Sri Lanka Institute of Information Technology

Pharmaceutical Management System

Project Report

Information Systems Project 2024

Project ID: **ISP_24_J_003**

Submitted by:

1. IT22319142 – WIJESINGHE A.G.T.
2. IT22310996 – THENNAKOON T.A.C.S.
3. IT22884138 – RATHNAYAKA R.M.T.D.
4. IT22364692 – KANDAGE K.T.S.

Submitted to:

……..

Ms. Narmada Gamage

Date of submission
14.06.2024

Abstract

A well-known pharmaceutical company, "Medicare Pharmaceuticals" is looking for an innovative solution to improve its overall business process, since it is currently using an ineffective and error-prone manual system. As the initiative for that revolution, this project presents a "Desktop Pharmaceutical Management Application" that automates the key business functions within the company. It will manage all the manual issues in Medicine Inventory & Stock Management, Supplier & Employee Management, Customer & Order Management, and Sales & Billing Management to minimize errors, improve data-driven decision-making and maximize productivity. Improved inventory & stock control, customer satisfaction, supplier chain management, and employee productivity are also promised by the suggested approach. At the same time, by automating & streamlining ordering & billing processes, this software seeks to promote financial accuracy and provide thorough reports. In the end, it is anticipated that the shift from manual to automated methods will culminate in substantial advantages for the inclusive business process of “Medicare Pharmaceuticals.”

Acknowledgement

We would like to express our sincere gratitude to every individual who helped us complete the "Desktop Pharmaceutical Management Application" project for Medicare Pharmaceuticals successfully.

We would also want to convey our gratitude to Ms. Narmada Gamage, our supervisor, whose persistent support, knowledgeable advice, and constructive feedback were significant in developing this project. She encouraged us toward striving for perfection and helped us overcome obstacles with her insightful guidance and constant support. We are grateful for her committed engagement with every phase of our project and greatly appreciate her mentorship.

Furthermore, we would like to extend our gratitude to our client, the main pharmacist and owner of Medicare Pharmaceuticals. Your specific expectations, frequent feedback, and explicit vision were essential to assisting us align our business goals with our development work. We value your trust in us and the opportunity you gave us to build a system that will revolutionize Medicare Pharmaceuticals' pharmaceutical management procedures. Your cooperation and encouragement were essential to this project's successful completion.

Lastly, we sincerely appreciate the support and resources provided by all the staff members of the Department of Computer Systems Engineering and our colleagues for their valuable guidance and motivation during the development process.

Declaration

We declare that this project report or part of it was not a copy of a document done by any organization, university any other institute or a previous student project group at SLIIT and was not copied from the Internet or other sources.

Project Details

Project Title	Pharmaceutical Management System
Project ID	ISP_24_J_003

Group Members

Reg. No	Name	Signature
IT22319142	WIJESINGHE A G T	
IT22310996	THENNAKOON T A C S	
IT22884138	RATHNAYAKA R M T D	
IT22364692	KANDAGE K T S	

Table of Contents

Abstract ..... **ii**

Acknowledgment ..... **iii**

Declaration ..... **iv**

Table of Contents ..... **v**

List of Figures ..... **vii**

List of Tables ..... **xiv**

1. Introduction **1**

1.1 Problem Statement 1

1.2 Product Scope 1

1.3 Project Report Structure 2

2. Methodology **3**

2.1 Requirements and Analysis 3

2.2 Design 12

2.3 Implementation 23

2.4 Testing 155

3. Evaluation **167**

3.1 Assessment of the Project Results 167

3.2 Lessons Learned 194

3.3 Future Work 195

4. Conclusion **196**

5. References **197**

Appendix A: Design Diagrams **198**

Appendix B: Test Results **199**

Appendix C: Selected Code Listings **200**

List of Figures

Figure 2.1: Use Case Diagram.……..... 6
Figure 2.2: Medicine Inventory & Stock Management Activity Diagram..... 7
Figure 2.3: Supplier Management Activity Diagram 8
Figure 2.4: Employee Management Activity Diagram 9
Figure 2.5: Customer & Order Management Activity Diagram..... 10
Figure 2.6: Sales and Billing Management Activity Diagram..... 11
Figure 2.7: Class Diagram……..... 12
Figure 2.8: ER Diagram……..... 13
Figure 2.9: Database Structure.……..... 14
Figure 2.10: Medicine Inventory Table Structure……..... 14
Figure 2.11: Stock Table Structure……..... 15
Figure 2.12: Supplier Table Structure……..... 15
Figure 2.13: Employee Table Structure……..... 16
Figure 2.14: Employee Attendance Table Structure……..... 16
Figure 2.15: Employee Salary Table Structure……..... 17
Figure 2.16: Customer Table Structure …….... 17
Figure 2.17: Order Table Structure……..... 18
Figure 2.18: Medicine Cart Table Structure……..... 18
Figure 2.19: Bill Table Structure……..... 19
Figure 2.20: Context Diagram..... 20
Figure 2.21: Level 1 Data Flow Diagram……..... 22
Figure 2.22: Structure Chart …….... 35
Figure 2.23: Company Logo……..... 23
Figure 2.24: Splash Form UI…….……..... 24
Figure 2.25: Splash Form Code…….……..... 24
Figure 2.26: DB Connection……..... 25
Figure 2.27: Login Page UI……..…….... ... 26
Figure 2.28: Login Page User Authentication Code..... 26
Figure 2.29: Dashboard UI…….....…….... 28

Figure 2.30: Dashboard Page Navigation Code……..... 28
Figure 2.31: User Profile UI……..... 29
Figure 2.32: Medicine Inventory Management Page UI…….... 30
Figure 2.33: Medicine Inventory Constructor and Class……..... 31
Figure 2.34: Medicine Inventory GUI Initialization…..…….... 31
Figure 2.35: Populate Medicine Type Combo Box..... 32
Figure 2.36: Event Handlers for Button and Other Components..... 32
Figure 2.37: Medicine Inventory Update delete Button Code……..... 33
Figure 2.38: Medicine Inventory Database Interaction …….... 34
Figure 2.39: Medicine Inventory Report Generation…..... 35
Figure 2.40: Use of Prepared Statements in Medicine Inventory Functionality..... 36
Figure 2.41: Stock Management UI…….…….... 37
Figure 2.42: Stock Management Class Declaration……..…..... 38
Figure 2.43: Stock Management Class Constructor..... 38
Figure 2.44: Stock Management Class Database Connection.....38
Figure 2.45: Medicine ID Combo Box Action Listener…….... 39
Figure 2.46: Get the name and storage location of the chosen medicine ID..... 39
Figure 2.47: Populate Medicine IDs in Combo Box..... 40
Figure 2.48: Action Listeners in Add Functionality..... 40
Figure 2.49: Code For update Stock Details..... 41
Figure 2.50: Code for Delete Stock Details …….....41
Figure 2.51: Code for Clear User Input fields of Stock Details…….. 42
Figure 2.52: Code of the Demo Button..... 42
Figure 2.53: Retrieve the details of the selected row of the Stock Table …….... 42
Figure 2.54: Search For a stock …….…….... 43
Figure 2.55: Refresh the Stock Table after every table data Manipulation..... 43
Figure 2.56: Prepared Statement related to Database connectivity……..... 44
Figure 2.57: Fetching data from the system database …….…….... 44
Figure 2.58: Formatting Dates in Stock Management…….... 45
Figure 2.59: Stock Management Page Components Initialization……..... 45
Figure 2.60: Fills Up Medical ID Combo Box with the existing medicine IDs..... 45

Figure 2.61: Insert Details of a new stock……..... 46
Figure 2.62: modify details of a Stock …….……..... 47
Figure 2.63: Delete a Stock……..... 48
Figure 2.64: Refresh Stock Table After a modification …….……..... 48
Figure 2.65: Supplier Management Page UI..... 49
Figure 2.66: Supplier Management Page Component Initialization Code..... 50
Figure 2.67: Supplier Management Page Contact No Validation Code……..... 52
Figure 2.68: Supplier Management Page Combo box Population Code..... 52
Figure 2.69: Adding a New Supplier to the System Database Functionality Code..... 53
Figure 2.70: Check for Duplicate Entries Functionality Code…….……..... 54
Figure 2.71: Auto Generate the Specific Medicine name of Supplied Medicine ID Functionality Code……. 55
Figure 2.72: Refresh the Supplier Table after Adding a new entry Functionality Code…. 56
Figure 2.73: Validate User Inputs for Contact No & Contact Person Functionality Code..... 56
Figure 2.74: Selecting a row from the Supplier Details Table Functionality Code…….. 57
Figure 2.75: Clearing all the User Input Fields Functionality Code..... 58
Figure 2.76: Updating a row from the Supplier Details Table Functionality Code……. 59
Figure 2.77: Deleting a row from the Supplier Details Table Functionality Code……... 60
Figure 2.78: Search for Registered Supplier Details from the Table Functionality Code…61
Figure 2.79: Customer Management Page UI ….……..... 64
Figure 2.80: Initialization of Customer Details Class …….... 65
Figure 2.81: Register a New Customer……. 66
Figure 2.82: Update Customer Details..... 68
Figure 2.83: Delete Customer Details…….... 69
Figure 2.84: Clear Customer Details..... 70
Figure 2.85: Customer Input Validation..... 70
Figure 2.86: Customer Email Validation.....……..... 70
Figure 2.87: Customer Registration ID Validation …..... 71
Figure 2.88: Customer Contact Number Length Validation….... 71
Figure 2.89: Customer first name and last name Character Input Validation..... 71
Figure 2.90: Create a database connection..... 72

Figure 2.91: Refresh Customer Table……..... 73
Figure 2.92: Customer Exception Handling……. 74
Figure 2.93: Customer Data Sorting Algorithm..... 77
Figure 2.94: Order Details Class Initialization.....……..... 78
Figure 2.95: Order Details Class Exit Button..... 79
Figure 2.96: Load Medicine Cart……..... 79
Figure 2.97: Retrieve Customer ID..... 81
Figure 2.98: Retrieve Order ID 82
Figure 2.99: Get Order Details..... 83
Figure 2.100: Logout Button..... 83
Figure 2.101: Back Button……..... 83
Figure 2.102: Medicine Cart Update Button……..... 84
Figure 2.103: Refresh Order Table..... 86
Figure 2.104: Add To Cart Button……..... 86
Figure 2.105: Calculate the total amount of the selected medicine…..……. 88
Figure 2.106: Get the Unit Price of the selected medicine…..…… 89
Figure 2.107: Validate the input for the Net Amount..... 90
Figure 2.108: Delete an Order Entry…..... 90
Figure 2.109: Get Medicine ID……..... 91
Figure 2.110: Get Total Amount ……..... 92
Figure 2.111: Search Orders……..... 92
Figure 2.112: Customer ID action performed method ……..... 93
Figure 2.113: Calculate the Total Amount……..... 93
Figure 2.114: Reorder Page Navigation…….……. 93
Figure 2.115: Add order Details……..... 94
Figure 2.116: Retrieve Medicine Details …….……..... 95
Figure 2.117: Calculates the total amount and updates the net total……96
Figure 2.118: Retrieves details of the selected row in the Cart table……... 97
Figure 2.119: Order Status Page UI……..... 98
Figure 2.120: Initialize Order Status Page Components …..... 99
Figure 2.121: Print Order Status Report..... 99

Figure 2.122: Retrieve Complete order Details..... 100
Figure 2.123: Retrieve Pending Order Details……..……..... 101
Figure 2.124: Employee Management Page …….……..... 102
Figure 2.125: Employee Management Page Components Initialization Code……..……. 103
Figure 2.126: Employee Management Page Demo Button Code…….. 104
Figure 2.127: Populate The Job Role Combo Box…… 105
Figure 2.128: Attendance Page Navigation Code ……..... 105
Figure 2.129: Salary Page Navigation Code..... 105
Figure 2.130: Dashboard Navigation Code..... 105
Figure 2.131: Code for Adding a New Employee……..... 106
Figure 2.132: Code for Retrieving the Gender Selection 107
Figure 2.133: Code for Loading the Modified Table ……..... 107
Figure 2.134: Code of Table Mouse Click Event..... 108
Figure 2.135: Code for Updating a Registered Employee..... 108
Figure 2.136: Code for Deleting a Registered Employee…….... 110
Figure 2.137: Code for Searching a Registered Employee..... 111
Figure 2.138: : Employee Attendance Tracking Page UI..... 114
Figure 2.139: Attendance Tracking Page Initial Code……..... 114
Figure 2.140: Employee ID Combo Box Code…… 115
Figure 2.141: Employee ID Combo Box Code…… 116
Figure 2.142: Get the Marked Attendance …… 116
Figure 2.143: Code for Adding a New Attendance Entry …….……..... 117
Figure 2.144: : Code for Selecting a row to update or delete an attendance record..... 118
Figure 2.145: Code for Modifying an Attendance record..... 119
Figure 2.146: Code for Deleting an Attendance record from the Database…… 120
Figure 2.147: Code for Searching an Attendance record from the Database ……..... 121
Figure 2.148: Employee Salary Calculation Page UI..... 123
Figure 2.149: Employee Salary Calculation Page Class Initialization Code……..... 124
Figure 2.150: Code to Load Employee IDs into the Combo Box ……..... 124
Figure 2.151: Get the Employee Name by the selected Employee ID…….……..... 125
Figure 2.152: Get the Daily Rate by the selected Employee ID…… 125

Figure 2.153: Enter a New Salary Record to the System Database 126
Figure 2.156: Code for retrieving the Monthly Attendance..... 127
Figure 2.157: Check for Duplicate Salary Entries…….....……..... 128
Figure 2.158: Selecting a row from the Salary Table ……..... 128
Figure 2.159: Update details of a row in the Salary Table…… 129
Figure 2.160: Delete a row from the Salary Table …….……..... 130
Figure 2.161: Search salary details in the Salary Table…… 131
Figure 2.162: Billing Page UI ……..... 132
Figure 2.163: Invoice UI…...……..... 132
Figure 2.164 Class Definitions of Billing Management..... 133
Figure 2.165: Fetching net amount of Billing Management ..……..... 133
Figure 2.166: Filling orderID of Billing Management …… 134
Figure 2.167: Apply discount & total amount of Billing Management …….…… 135
Figure 2.168: Enter key pressed of Billing Management ……..…….. 135
Figure 2.169: Add button of Billing Management ……..……. 136
Figure 2.170: print bill of Billing Management …….....…… 137
Figure 2.171: View button of Billing Management……..……138
Figure 2.172: Bill button of Billing Management ……...……..…… 139
Figure 2.173: Net amount calculation of Billing Management ……..….. 140
Figure 2.174: paid amount calculation of Billing Management…….....…… 141
Figure 2.175: Clear button code of Billing Management…….....…… 142
Figure 2.176: Refresh table code of Billing Management..…….....…… 143
Figure 2.177: update button code of Billing Management …….....…… 144
Figure 2.178: delete button code of Billing Management ……..…….....…… 146
Figure 2.179: Bill mouse clicked of Billing Management …….....…… 147
Figure 2.180: update paid amount code of Billing Management…….....…… 148
Figure 2.181: Sales Management Page UI..... 151
Figure 2.182: report generation of Sales management …….....…… 152
Figure 3.1: Splash Form UI…….....…… 166
Figure 3.2: Dashboard UI …….....…… 166
Figure 3.3: Medicine Inventory UI…….....…… 167

Figure 3.4: Medicine Inventory Adding Details 167
Figure 3.5: Medicine Inventory Update Details……..... 168
Figure 3.6: Medicine Inventory Delete Details..... 168
Figure 3.7: Medicine Inventory Report Generation……..... 169
Figure 3.8: Medicine Inventory Report……..... 169
Figure 3.9: Stock details UI 170
Figure 3.10: Stock Details Update……..... 170
Figure 3.11: Stock Details Delete..... 171
Figure 3.12: Stock Reordering page UI……..... 171
Figure 3.13: Add New Supplier ……..... 172
Figure 3.14: Modify Supplier Details ……..... 172
Figure 3.15: Delete a Supplier Record……..... 173
Figure 3.16: Search For Supplier 173
Figure 3.17: Add New Supplier……..... 174
Figure 3.18: Update a Registered Employee..... 174
Figure 3.19: Delete a Registered Employee……..... 175
Figure 3.20: Search details of a Registered Employee..... 175
Figure 3.21: Insert a New Attendance Record 176
Figure 3.22: Error message for Attendance Duplication..... 176
Figure 3.23: Updating the Attendance Status of an Employee..... 177
Figure 3.24: Deleting an Attendance Record ……..... 177
Figure 3.25: Searching Attendance of an Employee ……..... 178
Figure 3.26: Add New Salary Record..... 178
Figure 3.27: Add New Salary Record……..... 179
Figure 3.28: Update a Salary Record..... 179
Figure 3.29: Delete a Salary Record……..... 180
Figure 3.30: Search Salary Records of an Employee…… 180
Figure 3.31: Monthly Salary & Attendance Reports Generation..... 181
Figure 3.32: Add New Customer..... 182
Figure 3.33: Update the details of customer..... 182
Figure 3.34: Delete a customer from system database ……..... 183

Figure 3.35: Search for a Registered customer ……..... 183
Figure 3.36: Add New Order.....184
Figure 3.37: Modify Order Details……..... 184
Figure 3.38: Delete an Order..... 185
Figure 3.39: Search an Order……..... 185
Figure 3.40: Order Status Page……. 186
Figure 3.41: Order Status Reports Generation..... 186
Figure 3.42: Order Status Report..... 187
Figure 3.43: Add New Bill..... 188
Figure 3.44: View the Bill ……..... 188
Figure 3.45: Print the Bill ……..... 189
Figure 3.46: Selecting one of the invoices..... 189
Figure 3.47: Update the Invoice……..... 190
Figure 3.48: Delete an Invoice..... 190
Figure 3.49: Sales Page UI……..... 191
Figure 3.50: Sales Report Generation…… 191
Figure 3.51: Sales Report……. 192
Figure 6.1: High–Level Architecture Diagram……..... 197
Figure 8.1: Input Validations in Inventory ……..... 199
Figure 8.2: Interaction of Prepared Statements and Databases..... 199
Figure 8.3: Generate the Stock IDs ……..... 200
Figure 8.4: Search Functionality of Stocks..... 200
Figure 8.5: Quantity Update Stock……..... 201
Figure 8.6: Customer Search Algorithm..... 202
Figure 8.7: Customer Data Sharing Algorithms……..... 202
Figure 8.8: Employee Management Page Date Validation …..... 203
Figure 8.9: Employee NIC Validation..... 203
Figure 8.10: Employee Unique NIC Validations…..... 204
Figure 8.11: : Duplicate Attendance Record Validation..... 204
Figure 8.12: Refresh The Attendance Table…….. 205
Figure 8.13: Retrieve Daily Rate and the Employee Name..... 205

Figure 8.14: Validation Input in Salary Table ……..... 206

List of Tables

Table 2.1: Medicine Inventory Management Test Cases…….…. 155
Table 2.2: Medicine Stock Management Test Cases……..... 157
Table 2.3: Employee Management Test Cases..... 159
Table 2.4: Supplier Management Test Cases……..... 161
Table 2.5: Customer Management Test Cases …….. 163
Table 2.6: Order Management Test Cases……..... 164
Table 2.7: Sales And Billing Test Cases..... 165

1. Introduction

1.1 Problem Statement

Medicare Pharmaceuticals' current reliance on a manual system to manage medicine inventories and stocks, supplier and employee data, customer orders, and sales billing results in substantial inefficiencies and errors. Poor inventory management and stock monitoring, difficult interactions with suppliers, inaccurate orders, customer dissatisfaction, inadequate employee management, invoicing with human errors and a delay in decision-making are some of the main outcomes of these difficulties. This project suggests creating a thorough desktop pharmaceutical management application using Java to automate and optimize the overall business procedures to improve accuracy, productivity, and well-informed decision-making. [4]

1.2 Product Scope

The "Desktop Pharmaceutical Management Application" is being developed for "Medicare Pharmaceuticals" to streamline their operations and improve operational efficiency. The software aims to automate critical processes such as medicine inventory management, stock details, supplier and employee management, customer and order management, and sales and billing. It uses Java programming to offer real-time data accuracy, automated processes, and sophisticated functionalities across four main modules. The key benefits include effective inventory management, improved stock control, improved supplier relationships, enhanced customer service, increased employee productivity, order accuracy, data-driven decision-making capabilities, customer loyalty enhancement, financial accuracy, and advanced business analysis.

Some of the key benefits of the proposed system:

- Effective Inventory Management
- Financial Reporting
- Better Stock Management
- Improvement of Supplier Relationships
- Improved Customer Service
- Order accuracy.
- Business Analysis
- Financial Accuracy
- Employee Productivity
- Customer Loyalty
- Data-Driven Decision Making
- Effective Order Processing

1.3 Project Report Structure

In the remaining parts of this document, the methodology and the evaluation of the project will be presented. A use-case, ER, class, object, and context diagram are all included. The activity diagrams and sequence diagrams for the primary functions are included in the document. Funding is also available for the interfaces that include validations. You may discover the test cases, their outcomes, and a few unique codes that were utilized during implementation in this report.

IEEE standardized document conventions are followed by this project to guarantee consistency and clarity in the System Final Report. For ease of reading, the content uses Times New Roman font style and 12 font size. Main Heading uses 18 font size and Subheadings uses 14 font size. Those font sizes were used to highlight priority of the key features. The line spacing in the content is 1.5 and all the page numbers are at the right corner. All the diagrams in this final report are named in the same structure.

2. Methodology

2.1 Requirements and Analysis

During our extensive requirement elicitation phase with Medicare Pharmaceuticals' stakeholders, through direct interviews, several days of on-site observation and business document reviewing a detailed analysis was conducted to determine the business structure and current inefficiencies with their manual system as it is. As a result, significant flaws in medicine inventory management, stock control, supplier interaction, personnel administration, customer service, order processing, sales tracking, and billing accuracy were identified through this in-depth requirement analysis with the management and staff.

Therefore, this comprehensive requirement elicitation and analysis phase underscored the necessity for an automated desktop pharmaceutical management system which will be designed to reduce errors, support data-driven decision and optimize the overall operational efficiency of the business procedures. As we presented the need for an automated “Desktop Pharmaceutical Management System” while explaining the conceptual framework for the system development, our client accepted this change with optimism, stressing the expectation to have an automated software solution that meets high standards of quality and security.

The final analysis results were then assessed and summarized into the following essential business functions:

- Medicine Inventory Management
- Stock Management
- Supplier Management
- Employee Management
- Customer Management
- Order Management
- Sales Management
- Billing Management

The main functional requirements of each of the above key business functions are as follows:

- Medicine Inventory Management:
    - Add, update, and delete medicines in inventory.
    - Search for available medicines.
    - Calculate expiration dates for specific medicines.
    - View medicine inventory.
    - Generate daily medicine inventory report.
- Stock Management
    - Add, update, and delete Stock details.
    - Search for available stocks.
    - Real-time stock management based on data
    - Re-order stocks by sending emails to suppliers.
- Supplier Management
    - Add, update, and delete supplier details.
    - Search for registered suppliers.
    - View registered supplier details.
- Employee Management
    - Add, update, and delete employee details.
    - View the registered employee list.
    - Search for registered employee details.
    - Store daily attendance of employees.
    - Calculate monthly salary for employees based on monthly attendance.
    - Generate employee monthly salary reports based on daily attendance.
    - Add, update, and delete supplier details.
- Customer Management
    - Add, update, and delete customer details.
    - Search for registered customers.
    - Calculate loyalty points for customers.
    - View registered customer list.

- Order Management
    - Add, update, search, and delete orders.
    - View order list.
    - Generate order status.
    - Display error if the required medicine quantity is insufficient.
    - Generate unit price and total amount for orders.
    - Order return management.
    - Generate order status reports
- Sales Management
    - Analyze monthly sales of each medicine.
    - Calculate overall net sales, best-selling medicine, and monthly revenue.
    - Generate monthly sales reports.
- Billing Management
    - Add, update, and delete receipt details.
    - View payment receipt.
    - Print payment receipts as PDFs.
    - Calculate the net amount of the order.
    - Calculate the loyalty points that will be added to the order based on the net amount.
    - Generate discounts for loyal customers.
    - Calculate the net amount after discounts.
    - Generate the final balance after payment

And some of the essential non-functional requirements that were considered while developing the system which were highlighted by the client as follows: (The system was built with a specific focus on these non-functional requirements, because if these were not addressed properly, the entire system would have been a waste.)

- Performance Requirements
- Safety Requirements
- Security Requirements
- Software Quality Attributes
- Adhere to Business Rules

A use case was developed to provide a detailed visualization of the interactions and workflows within the system, enhancing the client's understanding. [1][2]

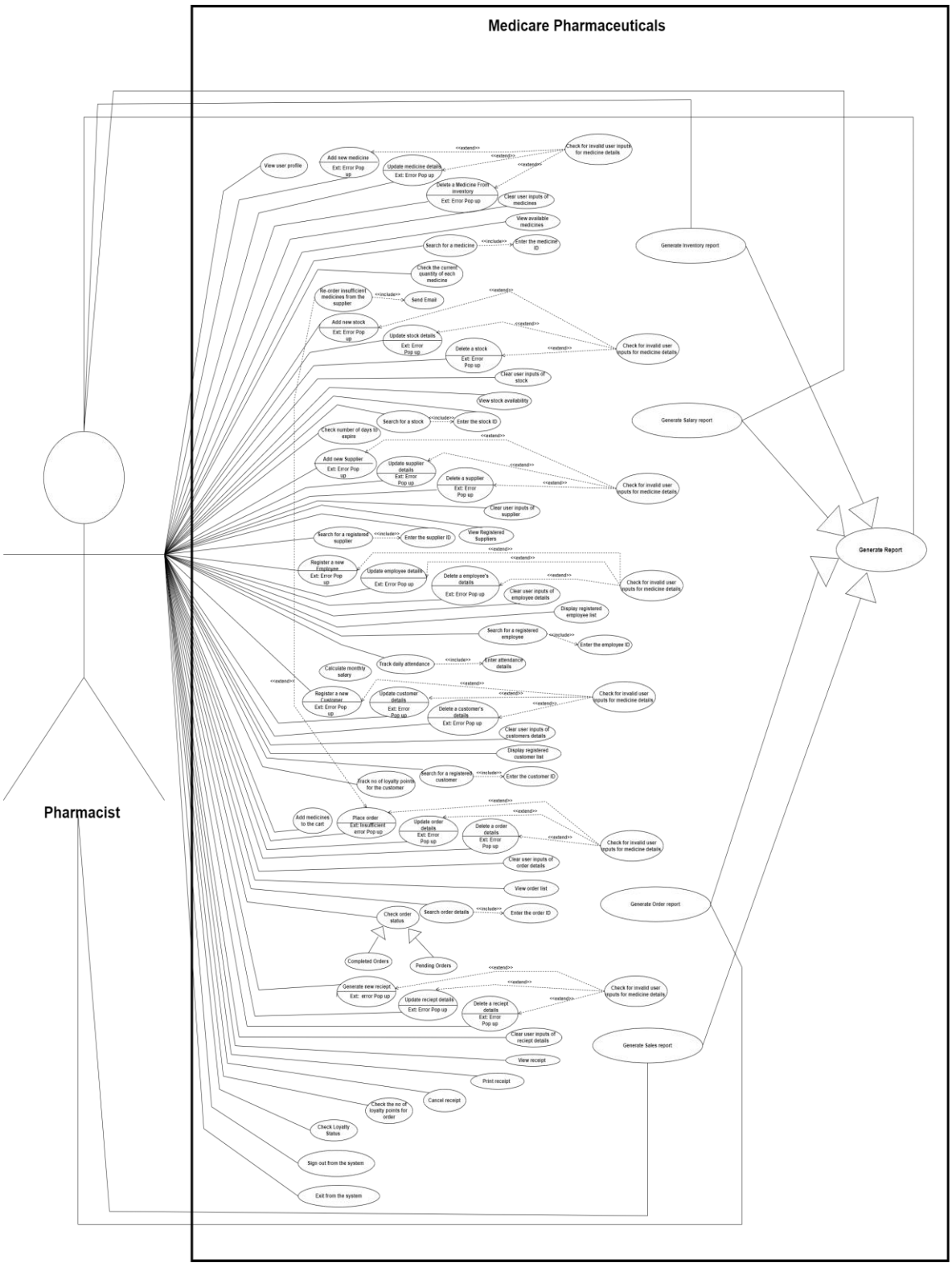

Figure 2.1: Use Case Diagram of the System

The following Activity diagrams provide a better representation of the selected key business processes while clarifying how users interact with the system to complete activities and demonstrate how it seamlessly switches from manual to automated processes.[1][2]

Medicine Inventory Management & Stock Management (IT22364692 – KANDAGE K.T.S)

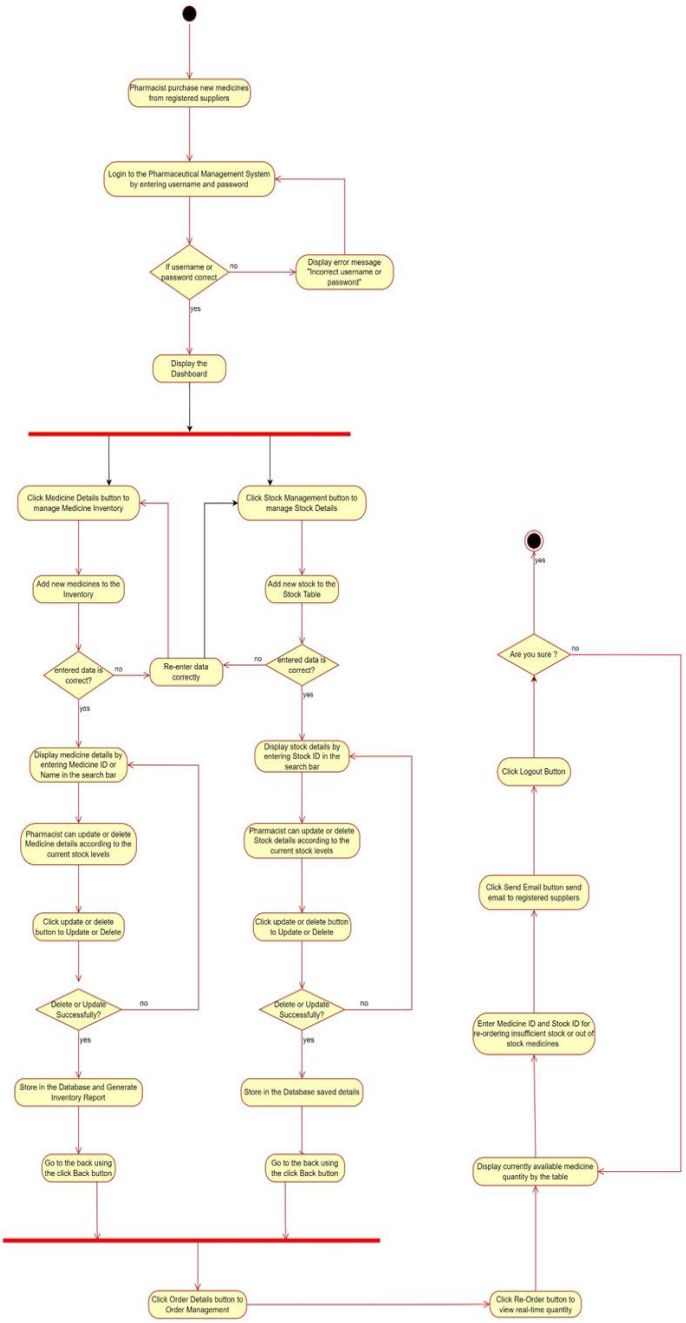

Figure 2.2: Activity Diagram – Medicine Inventory and Stock Management

Supplier Management (IT22319142 - WIJESINGHE A.G.T.)

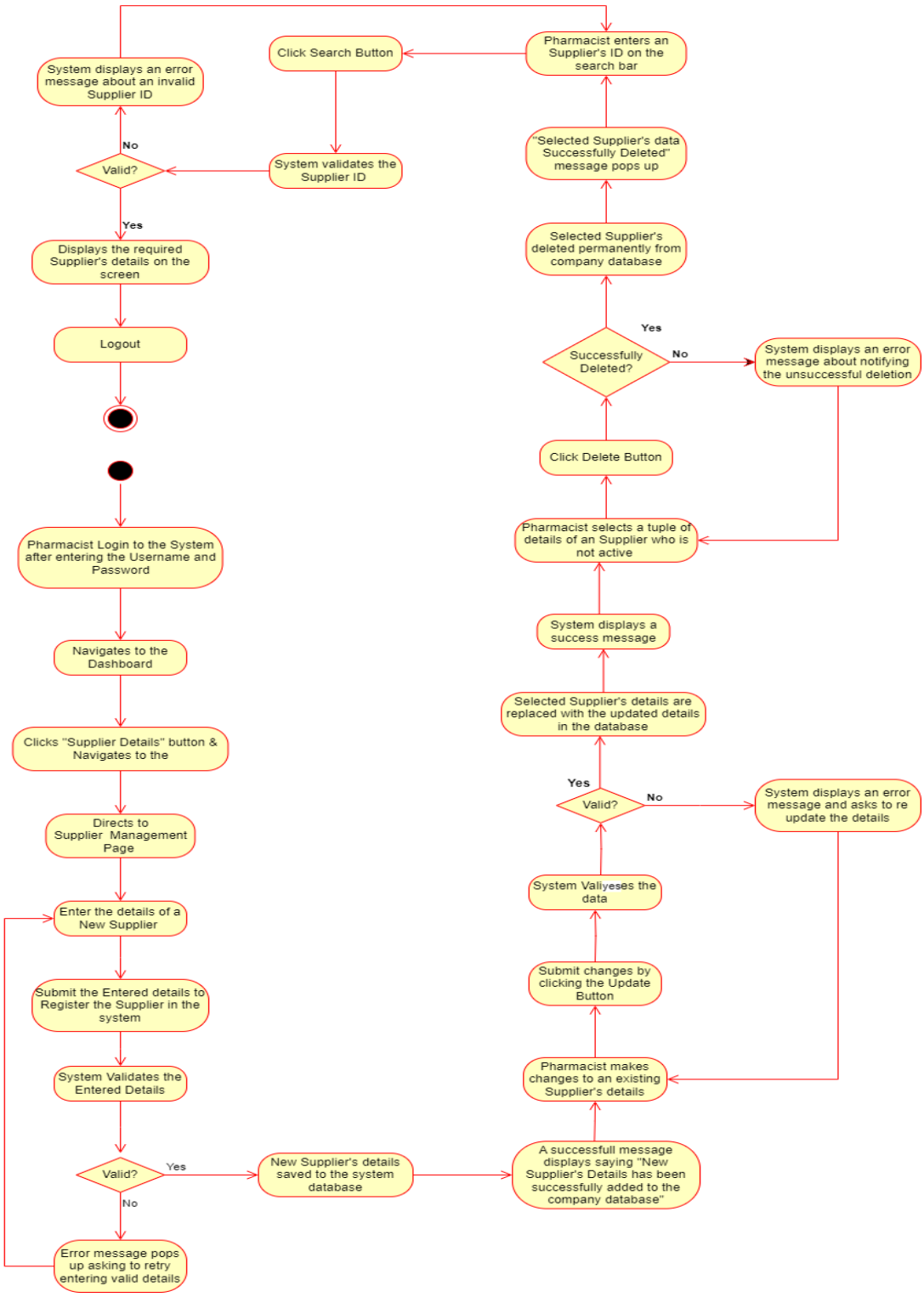

Figure 2.3: Activity Diagram – Supplier Management

Employee Management (IT22319142 - WIJESINGHE A.G.T.)

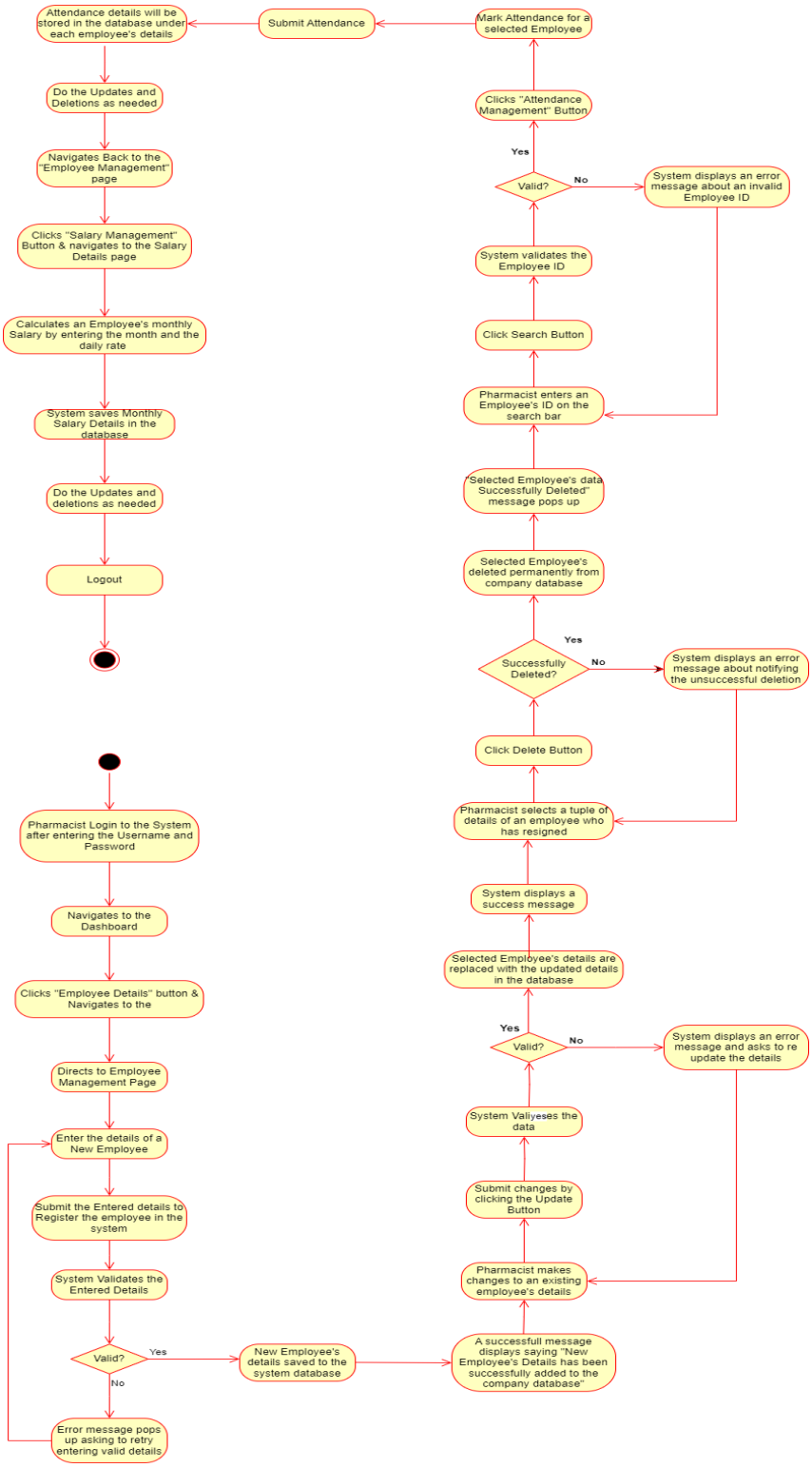

Figure 2.4: Activity Diagram – Employee Management

Customer Management (IT22884138 – RATHNAYAKA R.M.T.D)

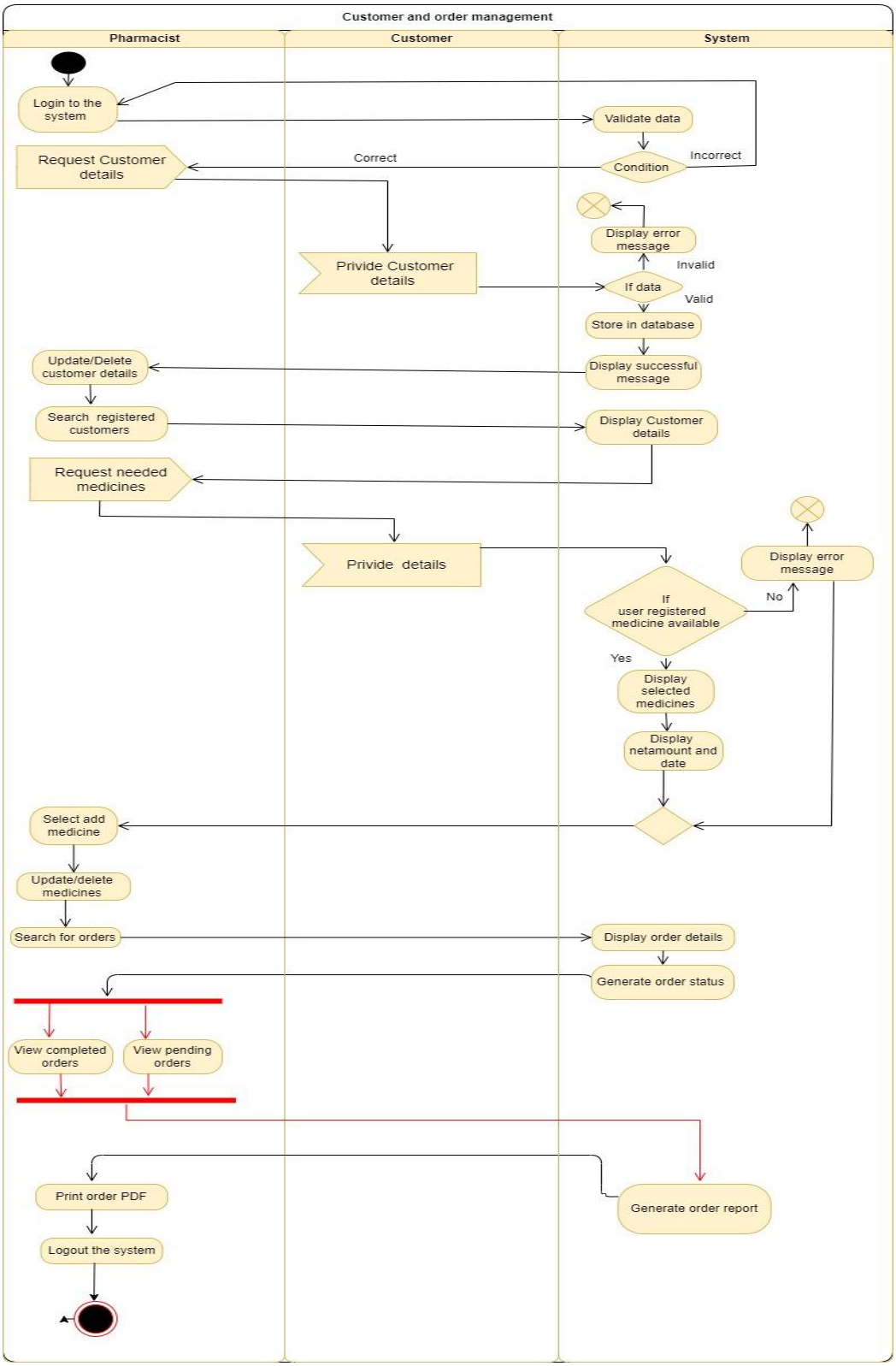

Figure 2.5: Activity Diagram – Customer & Order Management

Sales and Billing Management (IT22310996 – THENNAKOON T.A.C.S)

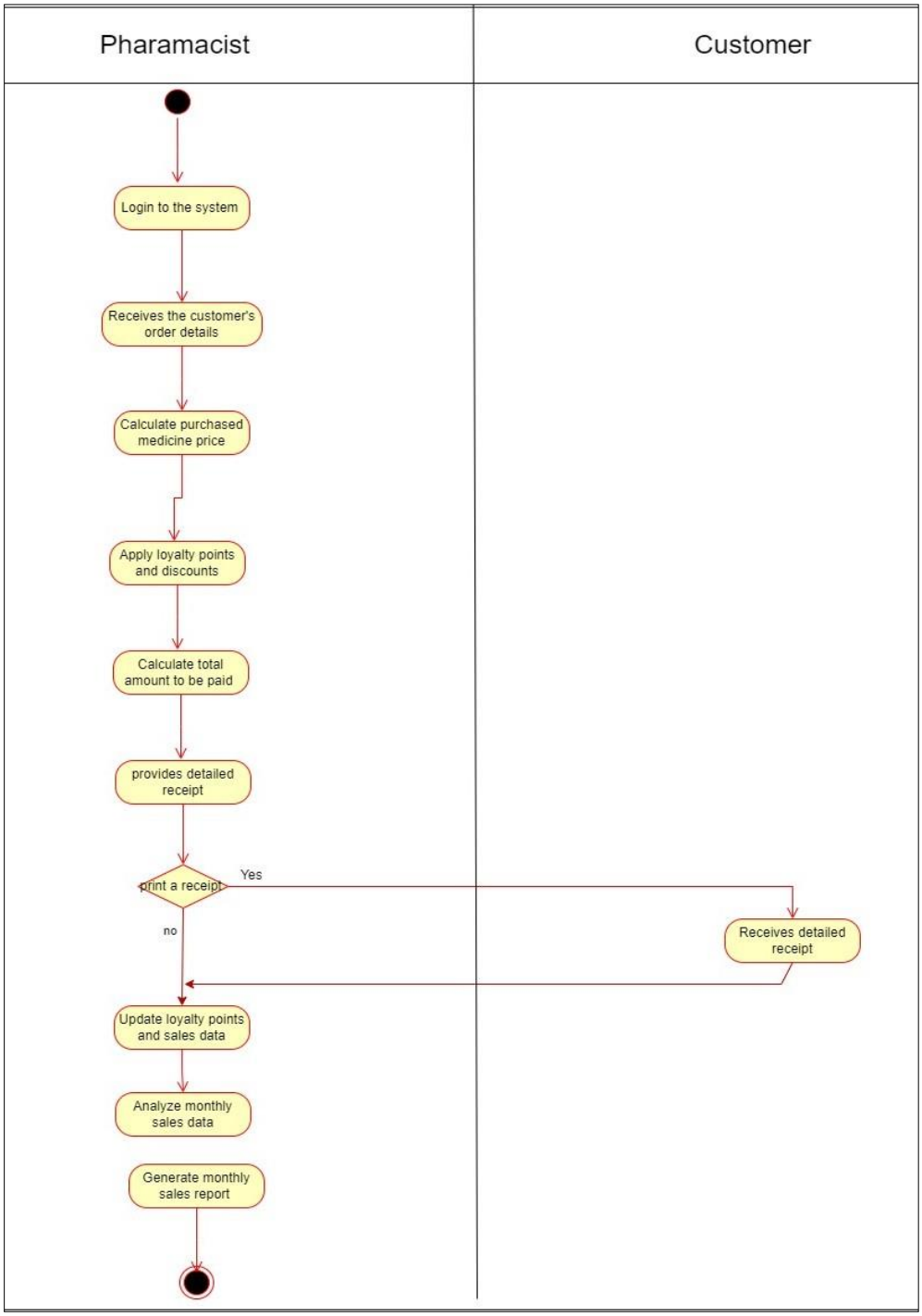

Figure 2.6: Activity Diagram – Sales and Billing Management

2.2 Design

This High-Level Architecture provides a holistic view of the Desktop Medicare Pharmaceutical Management system. The Pharmacist is the one who acts as the gatekeeper to the system, because in this system, the Pharmacist is the only person who has access to the desktop application. All the business functions are managed & controlled by the Pharmacist. The other stakeholders like Customers, Employees & Suppliers are accessing the system through the Pharmacist to interact with these functions. [1][2]

❖ Class Diagram

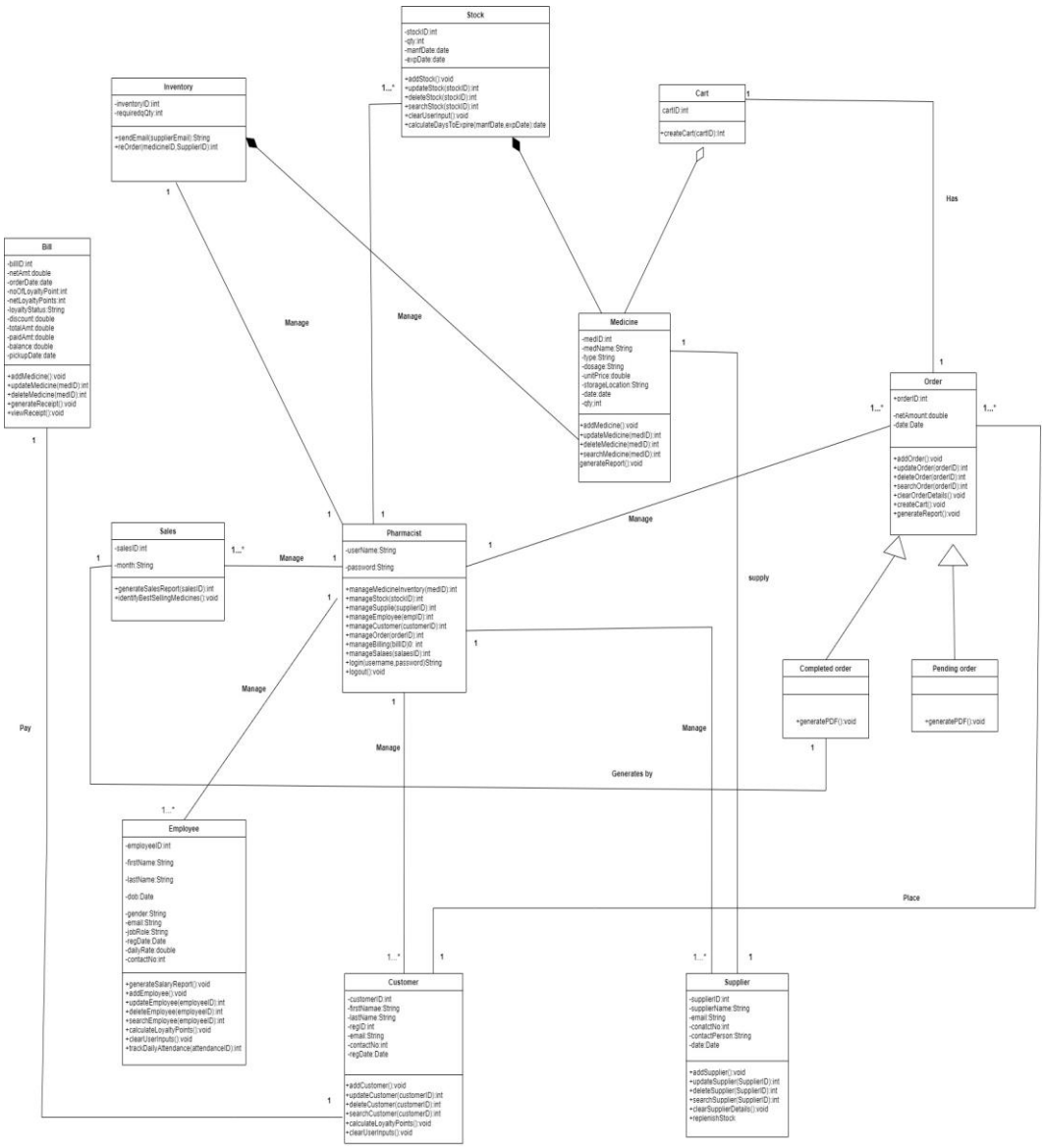

Figure 2.7: Class Diagram

❖ Entity Relationship Diagram

ER diagram of the system provides an intuitive understanding of data flow and preservation by showing the relationships between different entities and how they interact inside the system. It emphasizes the structural foundation required for reliable database interactions, guaranteeing data consistency and integrity throughout the system. [1][2]

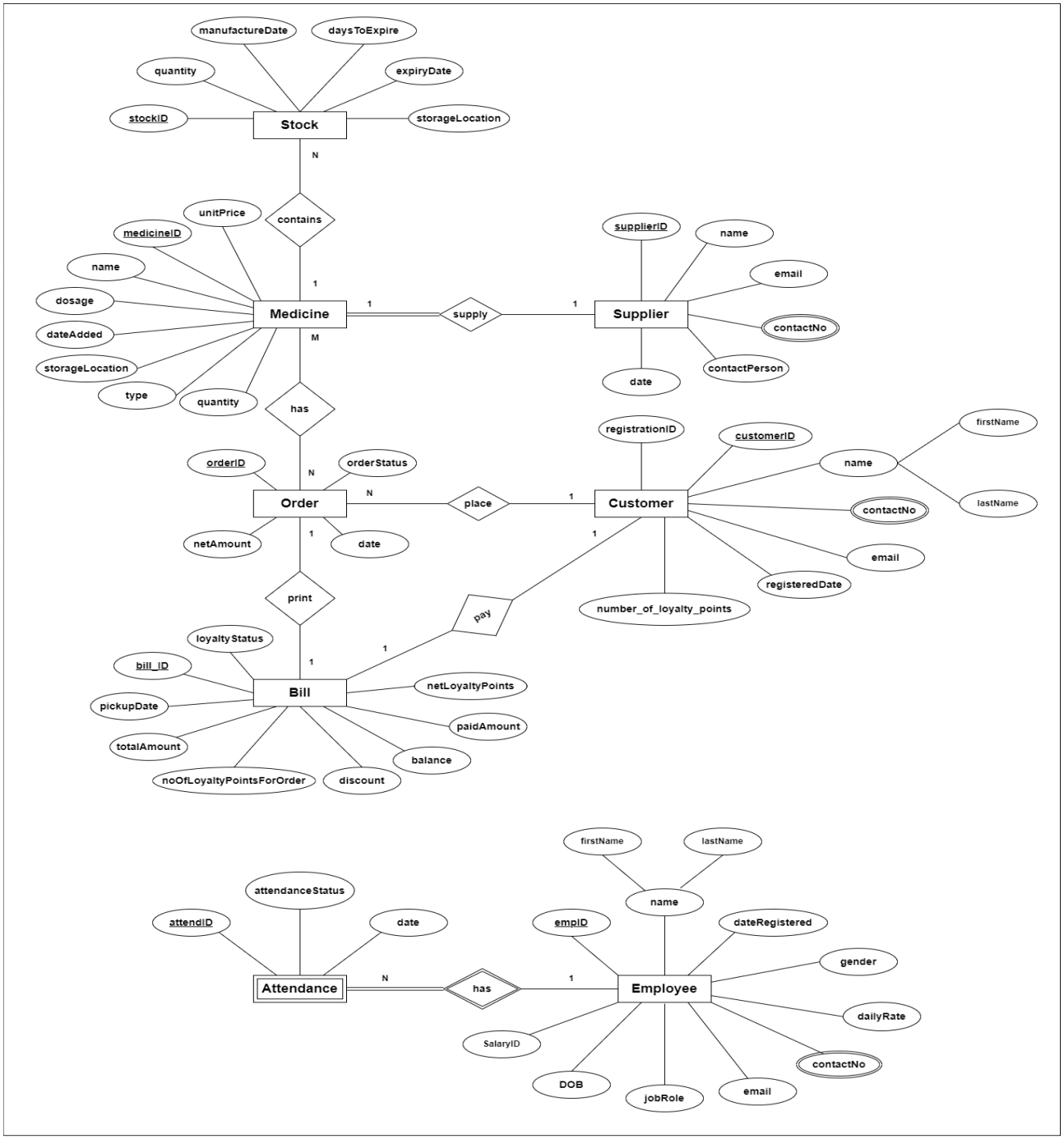

Figure 2.8: ER Diagram

❖ System Database

We planned and developed the system database based on the ER Design with all the required Primary Keys and Foreign Key Relations accurately. Entities in the ER Diagram represent the tables while the attributes represent the column names in each table.

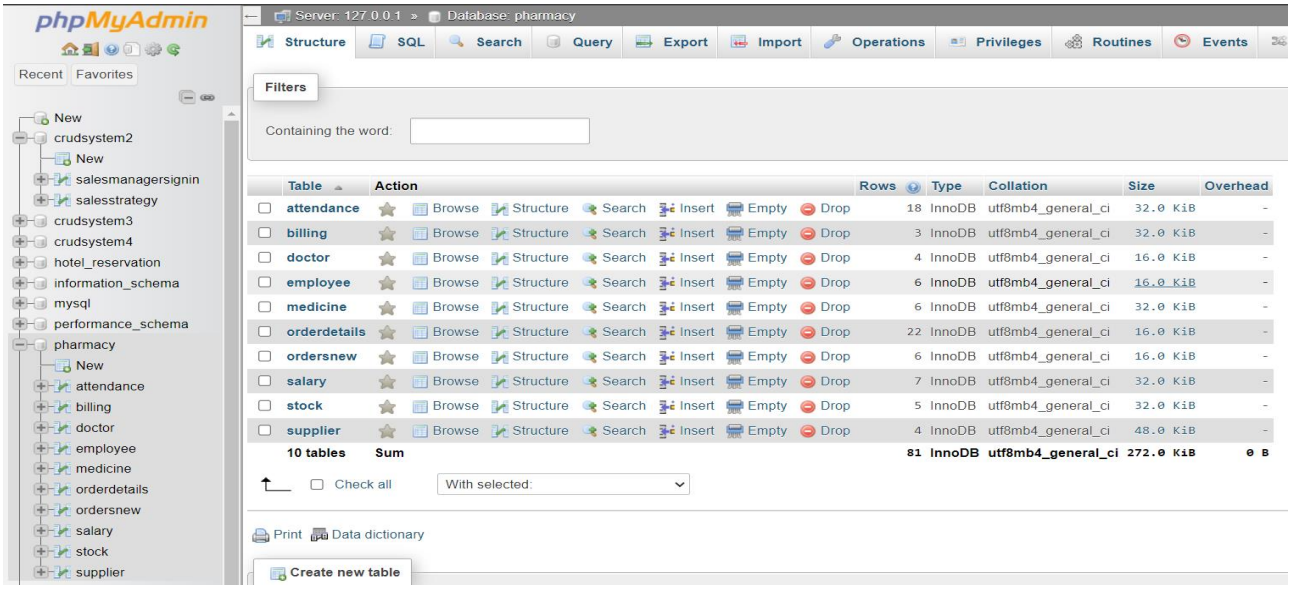

Figure 2.9: Database Structure

1) Medicine Inventory Management Function – (IT22364692 - KANDAGE K.T.S.)

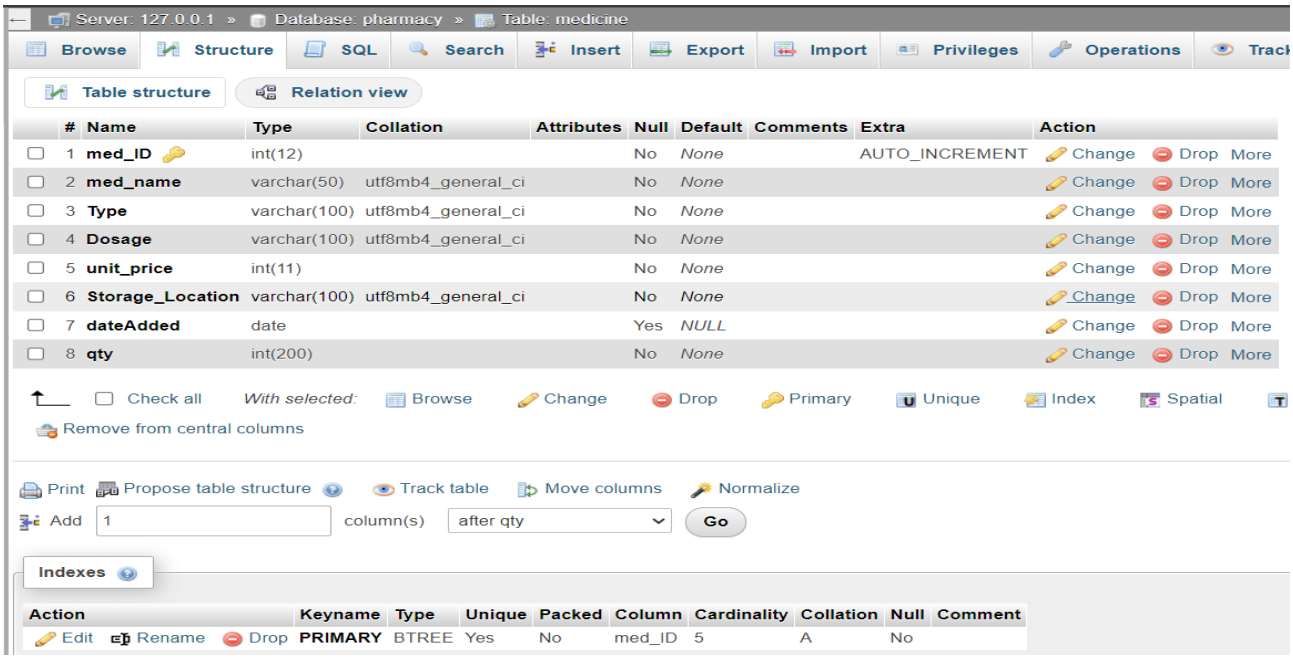

Figure 2.10: Medicine Inventory Table Structure

2) Stock Management Function – (IT22364692 - KANDAGE K.T.S.)

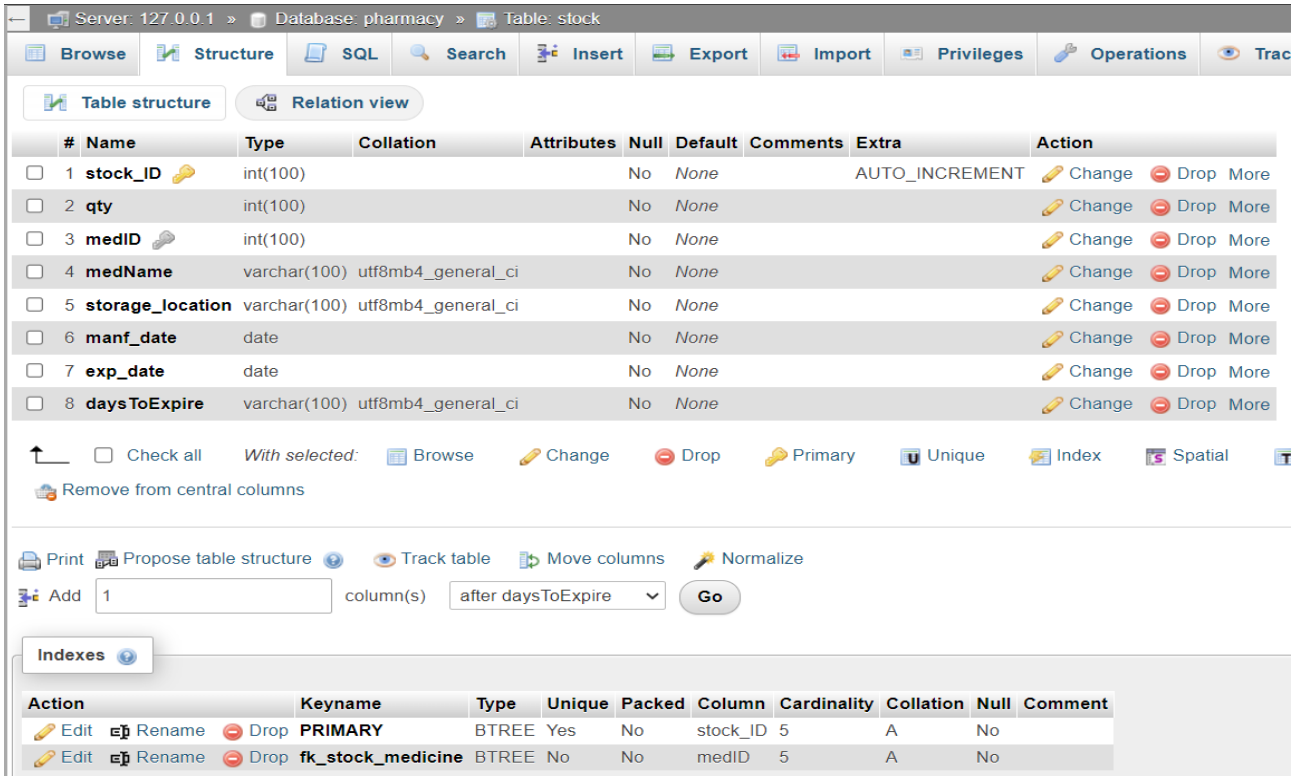

Figure 2.11: Stock Table Structure

3) Supplier Management Function – (IT22319142 – WIJESINGHE A.G.T)

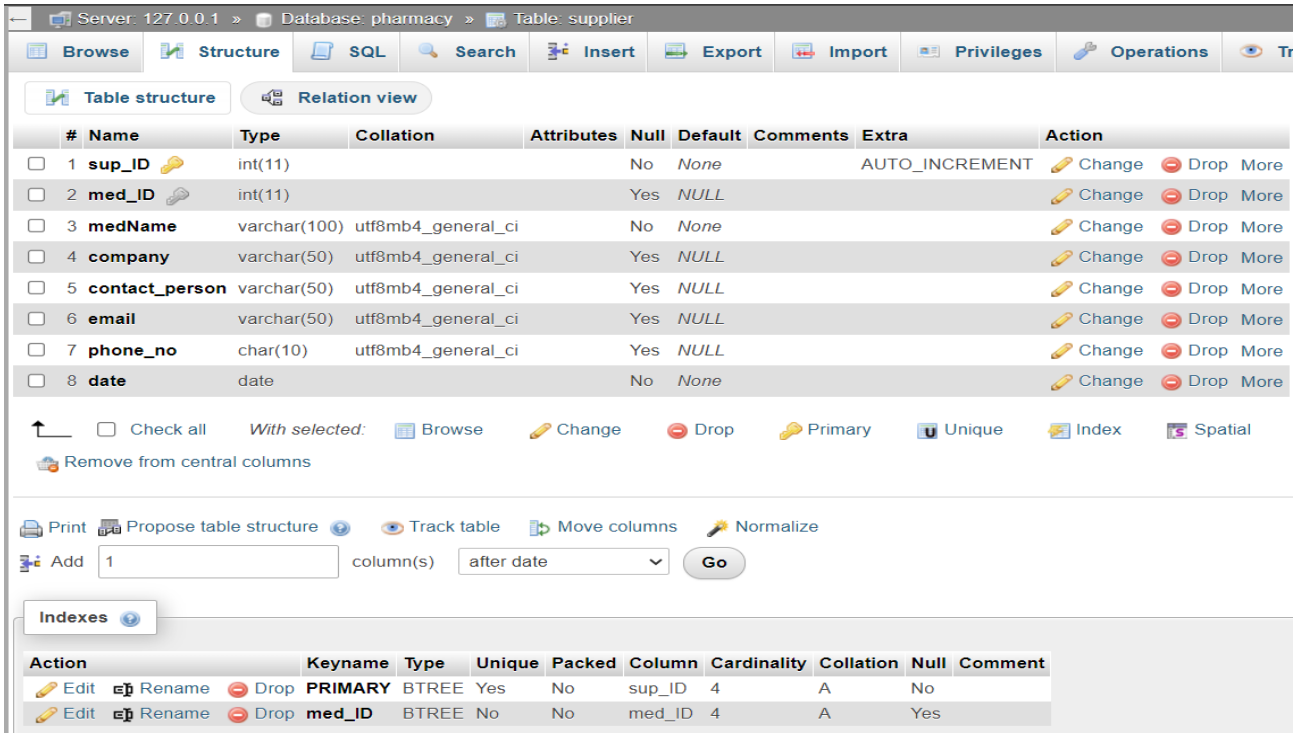

Figure 2.12: Supplier Table Structure

4) Employee Management Function – (IT22319142 – WIJESINGHE A.G.T)

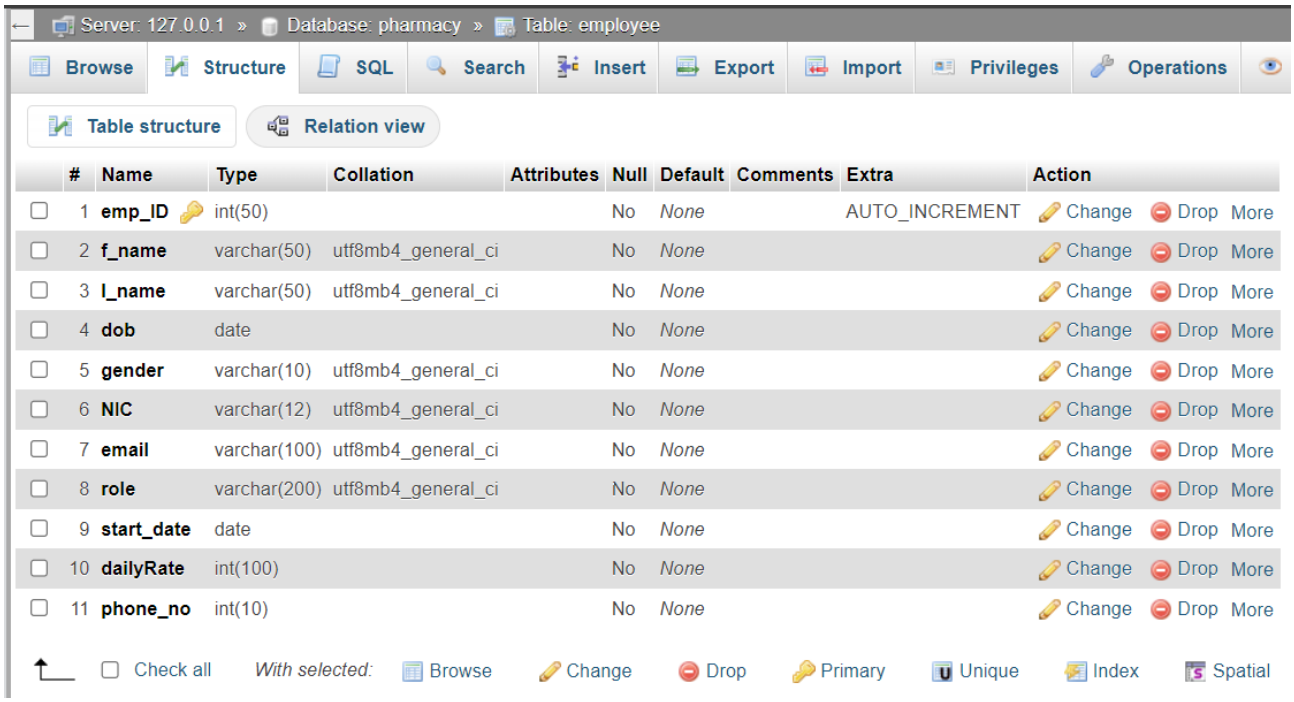

Figure 2.13: Employee Table Structure

5) Attendance Tracking: Employee Management Function

(IT22319142–WIJESINGHE A.G.T)

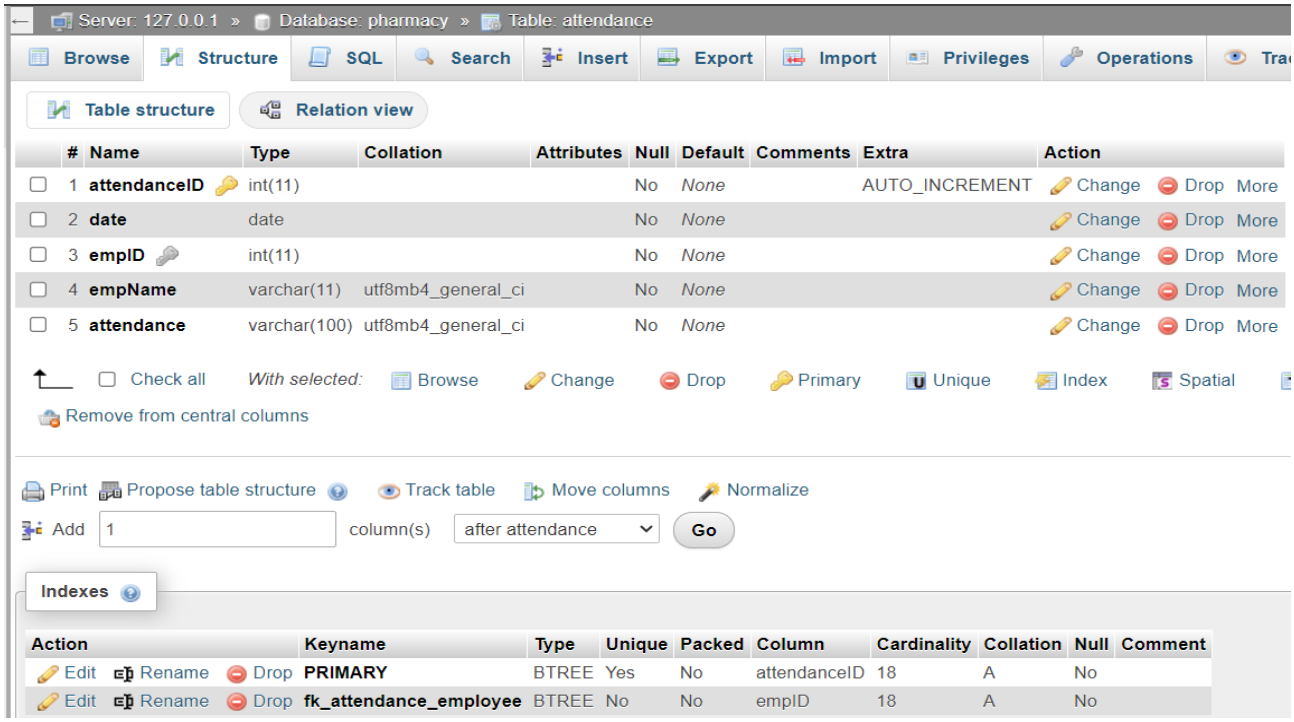

Figure 2.14: Employee Attendance Table Structure

6) Salary Calculation: Employee Management Function

(IT22319142–WIJESINGHE A.G.T)

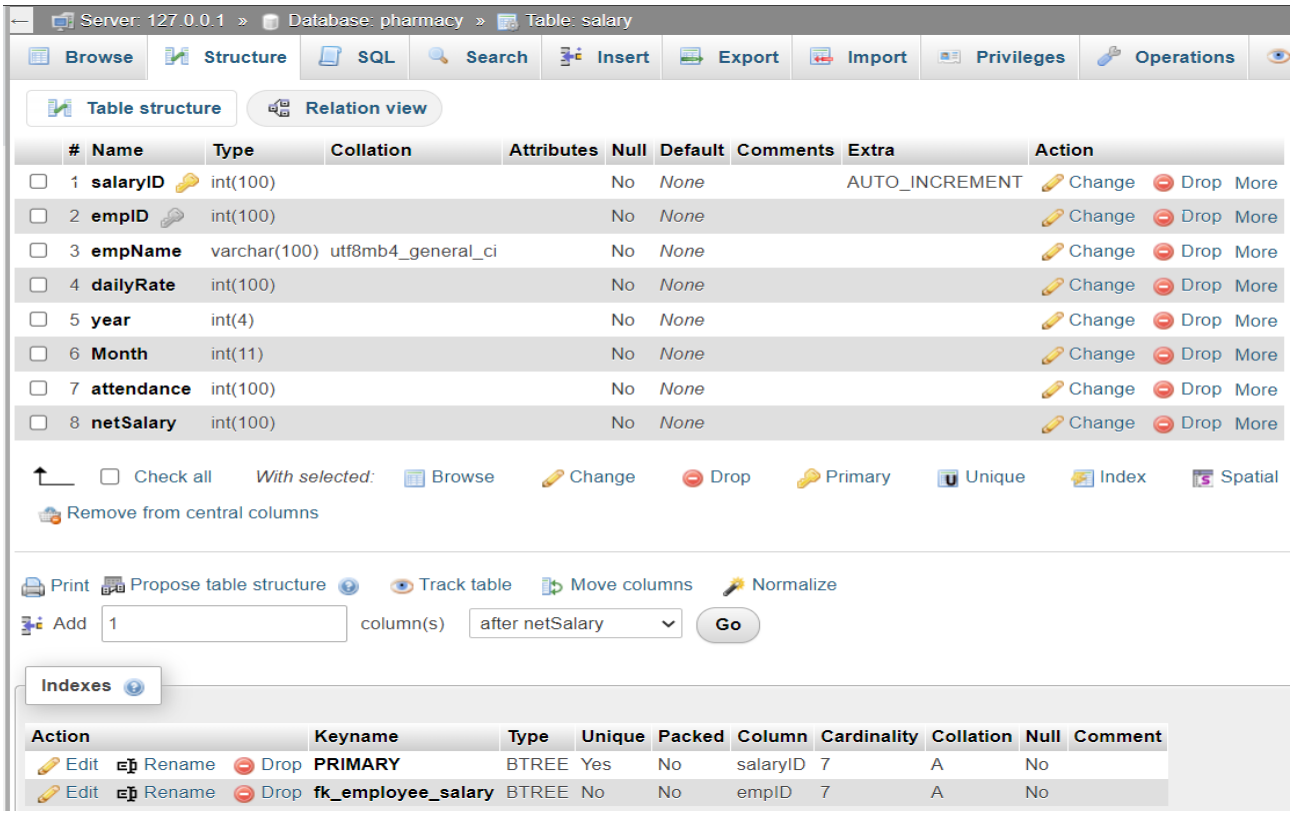

Figure 2.15: Employee Salary Table Structure

7) Customer Management Function – (IT22884138 – RATHNAYAKA R.M.T.D)

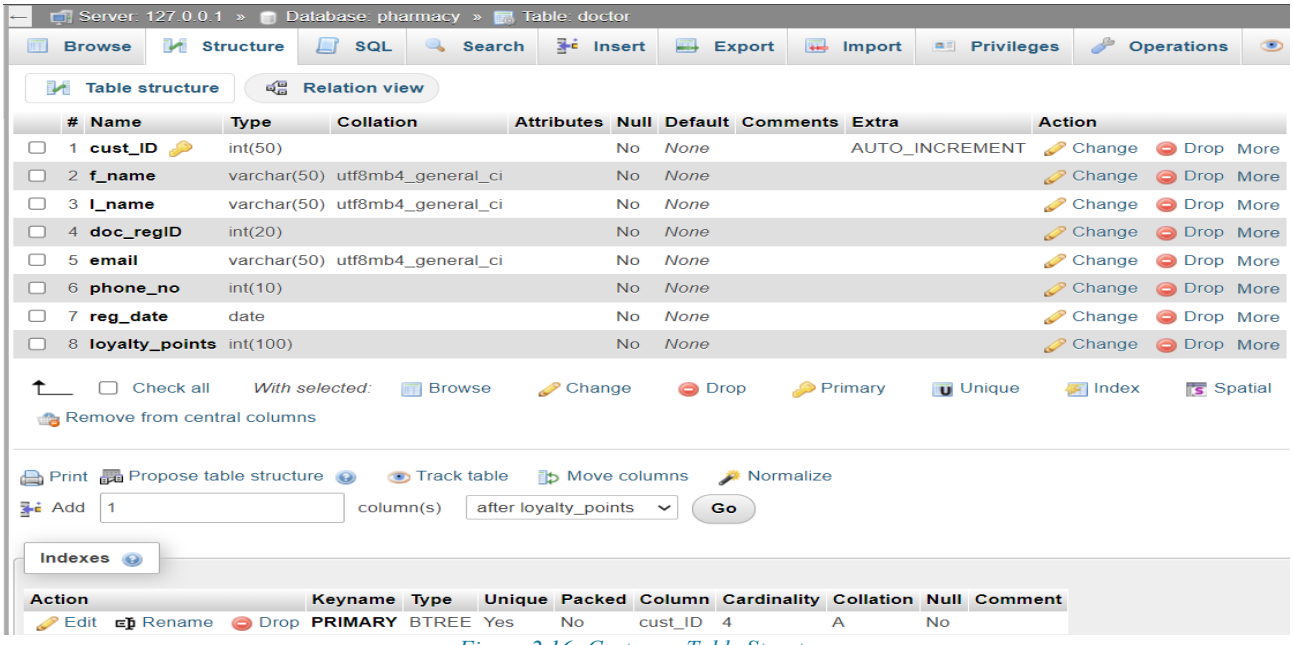

Figure 2.16: Customer Table Structure

8) Order Management Function – (IT22884138 – RATHNAYAKA R.M.T.D)

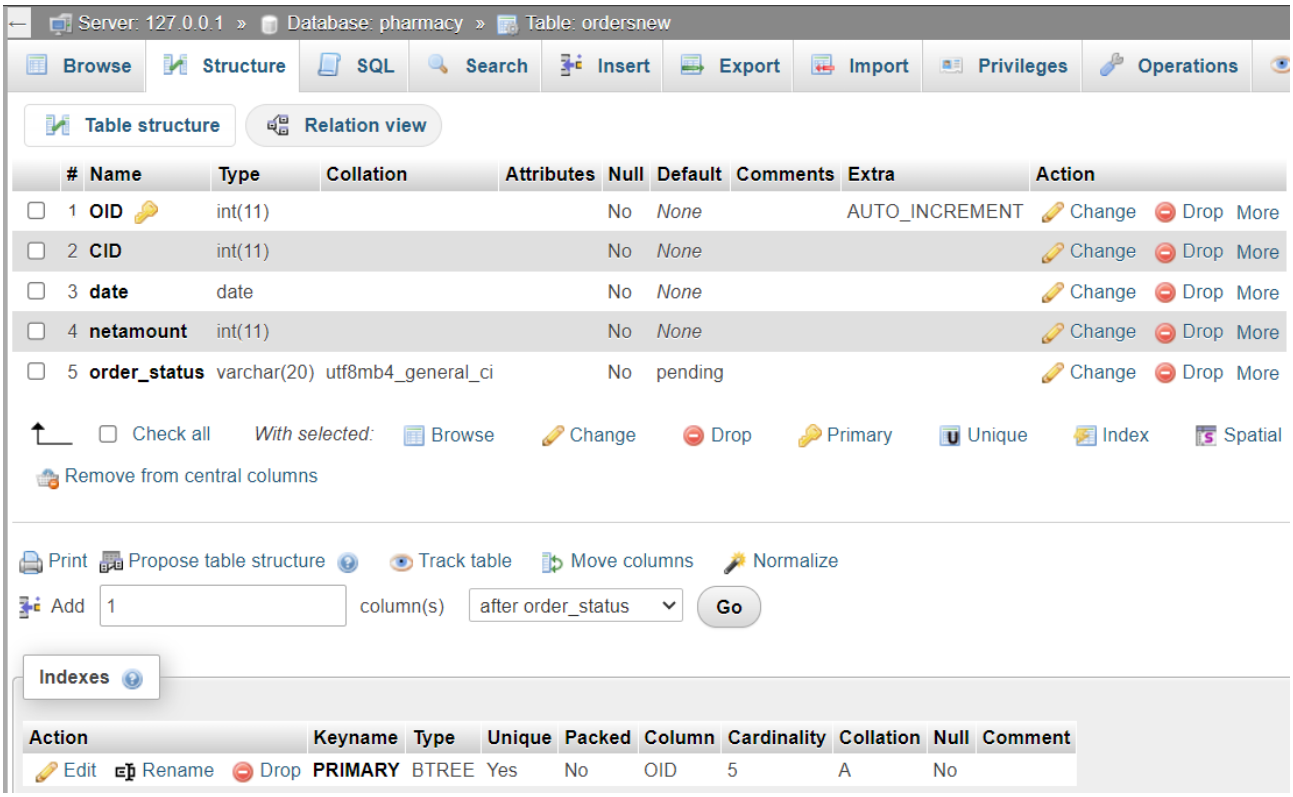

Figure 2.17: Order Table Structure

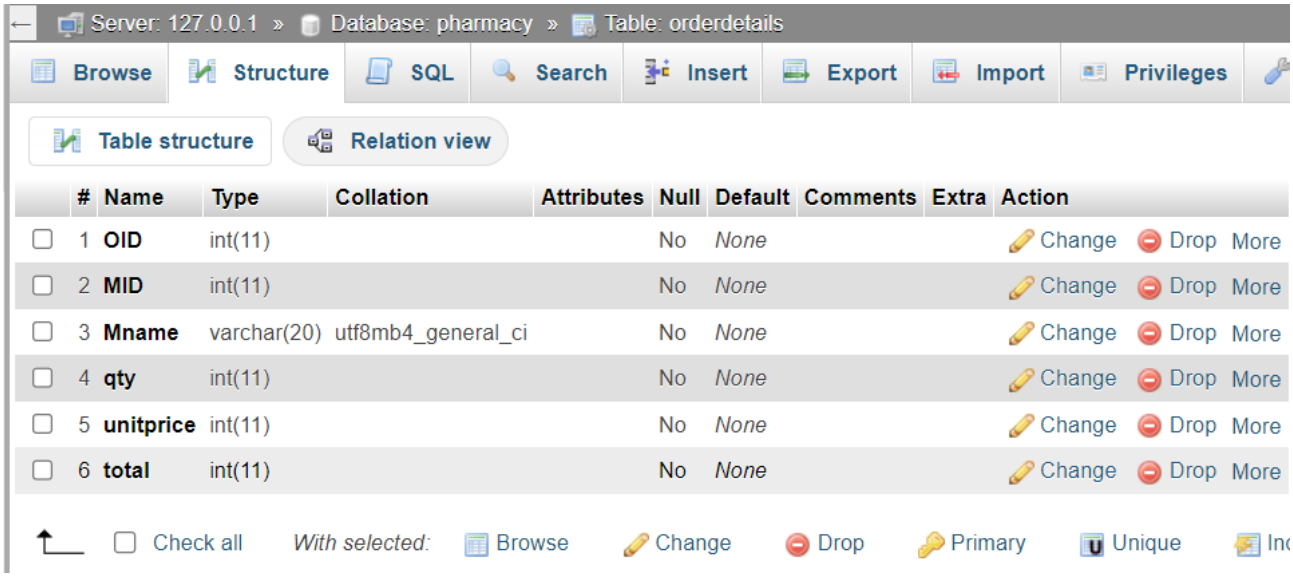

Figure 2.18: Medicine Cart Table Structure

9) Billing Management Function – (IT22310996 – THENNAKOON T.A.C.S)

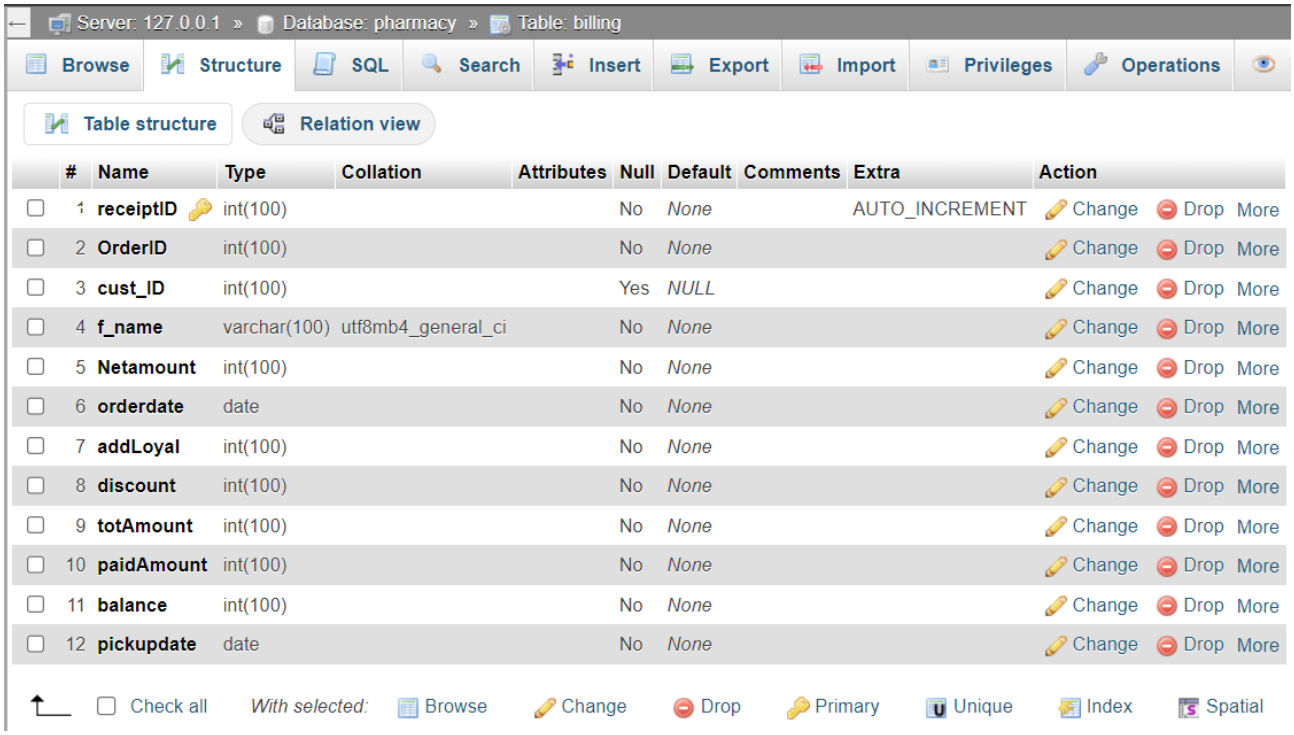

Figure 2. 19: Bill Table Structure

❖ Data Flow Diagrams

I. Context Diagram

To ensure that all significant external factors and interactions are taken into consideration during the development and implementation phases, the context diagram serves as a fundamental tool for understanding the system's environment. It illustrates how different stakeholders and internal departments interact with the system to accomplish key business goals by defining these external interactions.[8]

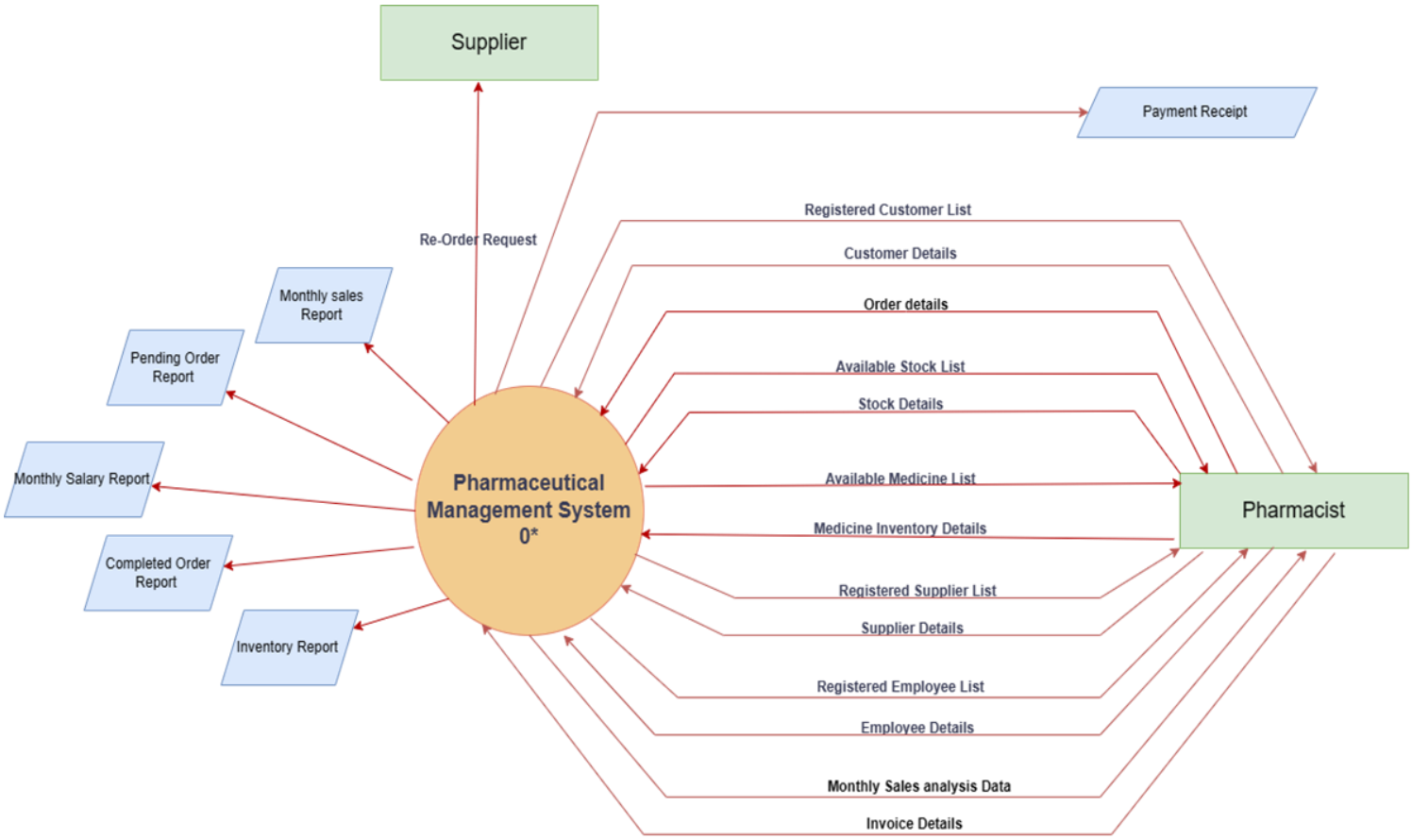

Figure 2.20: Context Diagram

II. Level 1 DFD Diagram

The DFD Level 1 Diagram depicts the organized movement and transformation of data between main business procedures inside the system by demonstrating the flow of information through those procedures. It highlights the crucial role that each process serves in obtaining seamless data management and operational efficiency by emphasizing the integration of multiple data sources and destinations, such as databases, user interfaces, and external systems.[8]

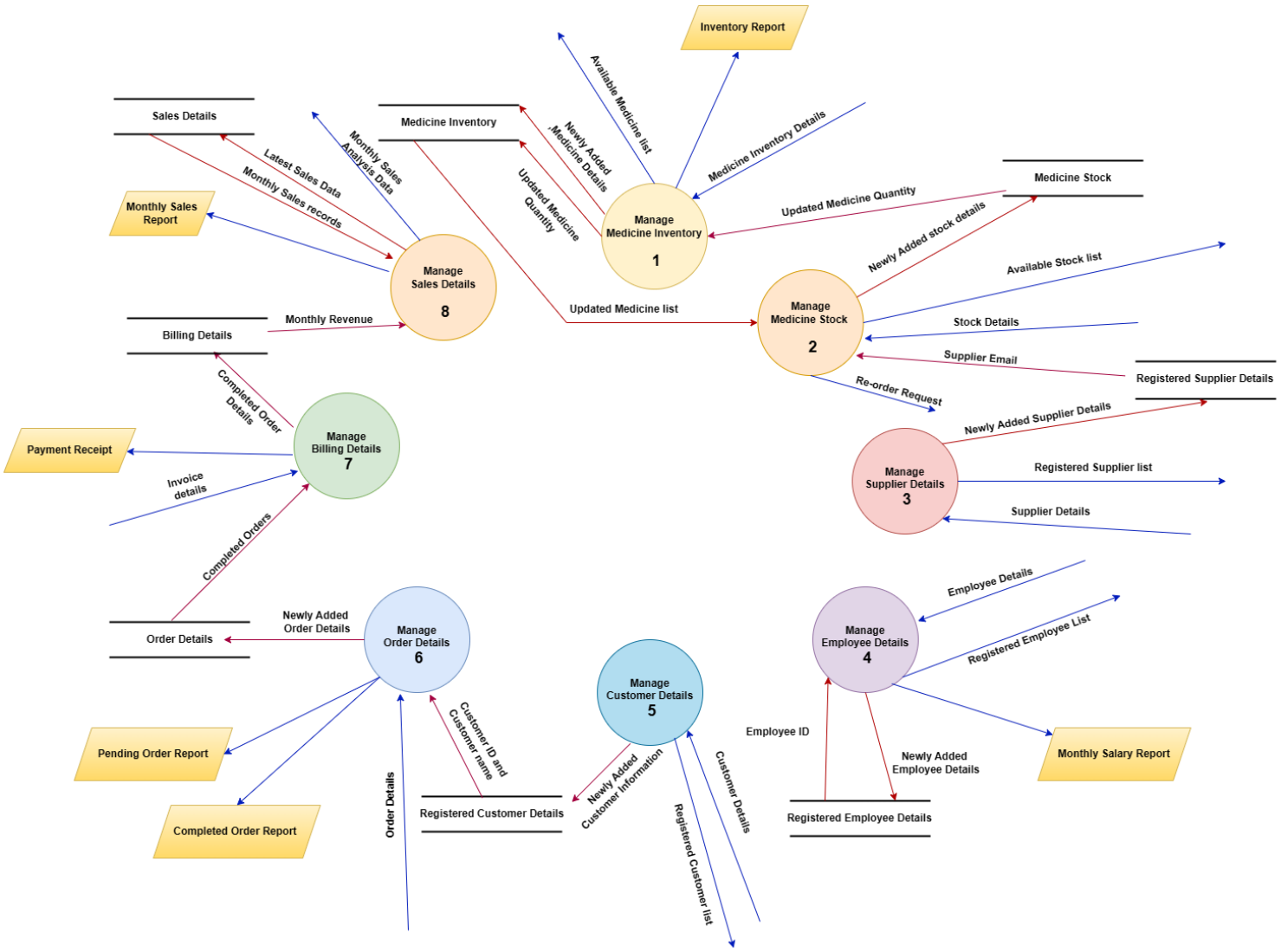

Figure 2.21: Level 1 Data Flow Diagram

❖ Structure Chart

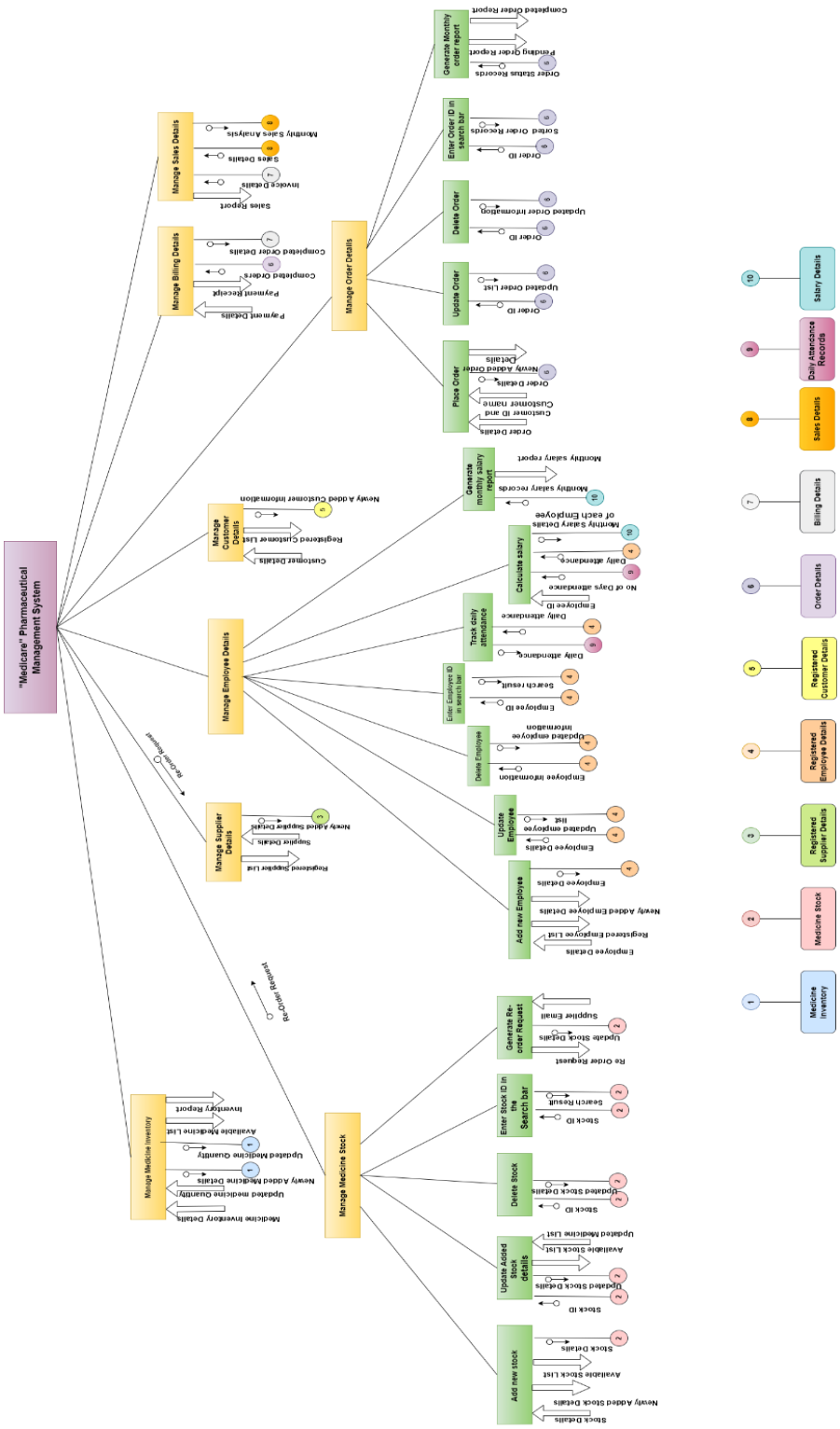

Figure 2.22: Structure Chart

2.3 Implementation

Medicare Pharmaceuticals' Java desktop pharmaceutical management system, which has been developed with the “NetBeans IDE 19”, incorporates extensive functionality to streamline a range of business procedures. The database that was used is structured to effectively manage all the essential modules such as inventory management for medicines, stock management, supplier management, employee management, customer management, order management, billing management, and sales management.

The architecture of the system is modular and reusable, utilizing powerful development tools and reusable code components to ensure scalability and maintainability. It was decided to use “phpMyAdmin” as the database management system (DBMS) because of its intuitive interface and robust SQL capabilities, which enable convenient database installation and administration. “Java” was chosen as the main programming language because of its wide range of libraries and platform independence, which made it possible to create a desktop application that was both secure and responsive. The appendices contain accurate information about the design and operation of special algorithms used in the system, such as those for producing sales reports and optimizing inventory levels.

Figure 2.23: Company Logo

❖ Splash Form

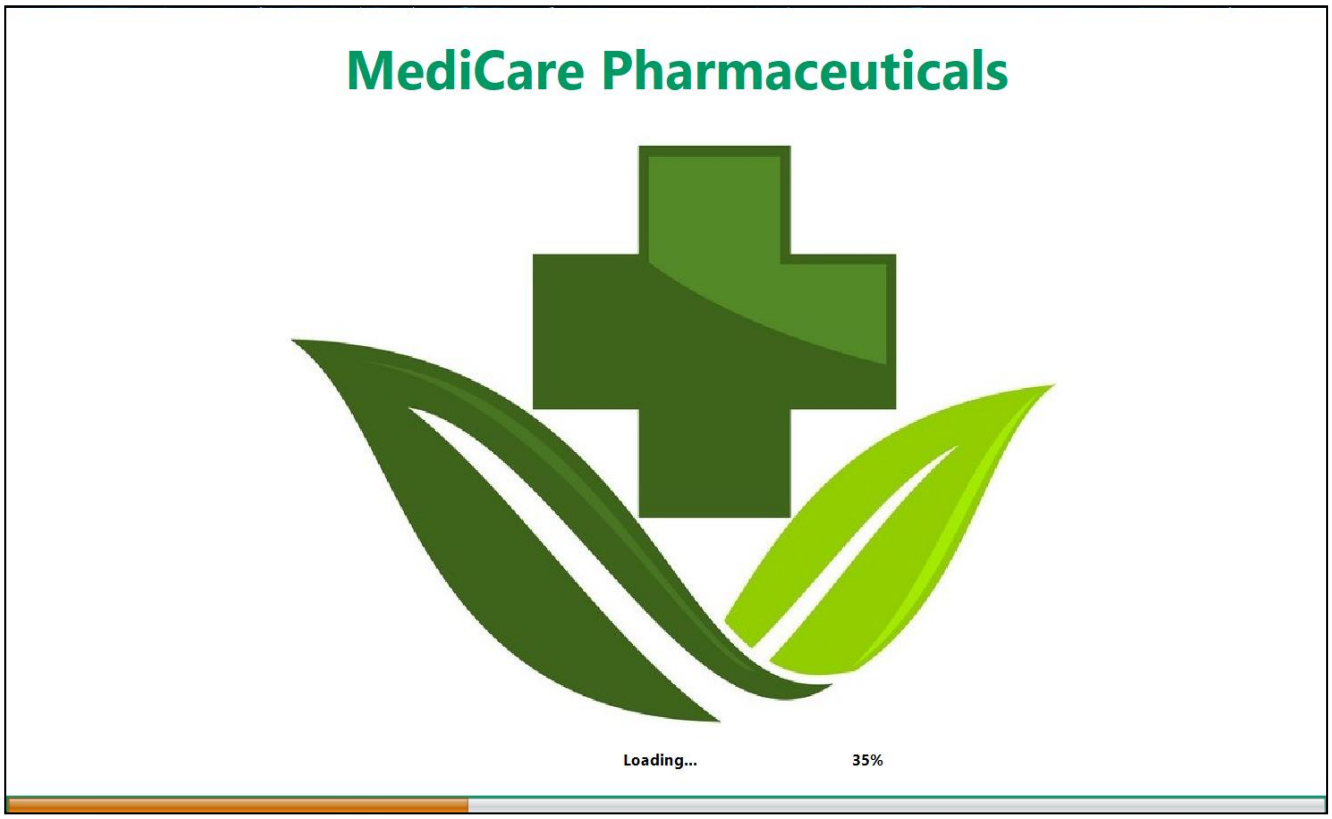

Figure 2.24: Splash Form UI

```
public static void main(String args[])
{
    /* Set the Nimbus look and feel */
     Look and feel setting code (optional)

    SplashForm obj = new SplashForm();
    obj.setVisible(b: true);

    try
    {
        for(int i=0;i<=100;i++)
        {
            Thread.sleep(millis:100);
            obj.Loading_value.setText(i+"%");
            obj.Loading_bar.setValue(n: i);

            if(i==5)
                obj.loading_label.setText(text: "Turning on...");
            if(i==30)
                obj.loading_label.setText(text: "Loading...");
            if(i==50)
                obj.loading_label.setText(text: "Connection Successfull...");
            if(i==80)
                obj.loading_label.setText(text: "Launching the Application...");
        }
        obj.dispose();
    }
    catch (Exception e)
    {

    }
```

Figure 2.25: Splash Form Code

❖ **Explanation of Splash Form Thread Code Functionality:**

The pharmacy package's “SplashForm” class extends “javax. swing. JFrame” and serves as the application's splash screen. An instance of “SplashForm” is generated and displayed visible upon the execution of the main method. This splash screen gives the user feedback while the application is starting by simulating a loading process with a background thread. The main purpose of the thread is to simulate loading by iterating from 0 to 100 and using “Thread.sleep(100)” to pause for 100 milliseconds at each step.

The loading bar (Loading_bar) and loading value label (Loading_value) with messages "Turning on..." appears at 5%, "Loading..." at 30%, "Connection Successful..." at 50%, and "Launching the Application..." at 80%, among other specific messages that are presented at significant milestones. n the splash screen is updated to reflect the current state of development at the end of each cycle. This shows the user where they are in the loading process, making the experience dynamic and engaging. The splash screen is eliminated from view using “obj. dispose ()” once the loading has reached 100%. After that, “java.awt. EventQueue.invokeLater” is used to create and display the login form, ensuring that it is created and displayed on the Event Dispatch Thread—a crucial component of thread safety in Swing applications. During the application startup phase, the “SplashForm” class takes advantage of Java Swing components and multithreading to produce a fluid and responsive user interface. By visually engaging the user during this time, the “SplashForm” class improves the overall user experience.

- **Database Connection Class:**

```
package pharmacy;
// @author thath

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;

public class DBconnection
{
    public static Connection con;

    public static Connection createDBconnection()
    {
        try
        {
            try
            {
                // Loading the MySQL JDBC driver
                Class.forName(className: "com.mysql.cj.jdbc.Driver");
                //Creating the Database Connection
                con = DriverManager.getConnection(url: "jdbc:mysql://localhost:3306/pharmacy?zeroDateTimeBehavior=CONVERT_TO_NULL",user: "root",password: "");
            }
            catch (ClassNotFoundException ex)
            {
                Logger.getLogger(name: DBconnection.class.getName()).log(level:Level.SEVERE, msg: null, thrown: ex);
            }
        }
        catch(SQLException ex)
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: ex);
        }
        return con;
    }
}
```

Figure 2.26: DB Connection

❖ Login Form

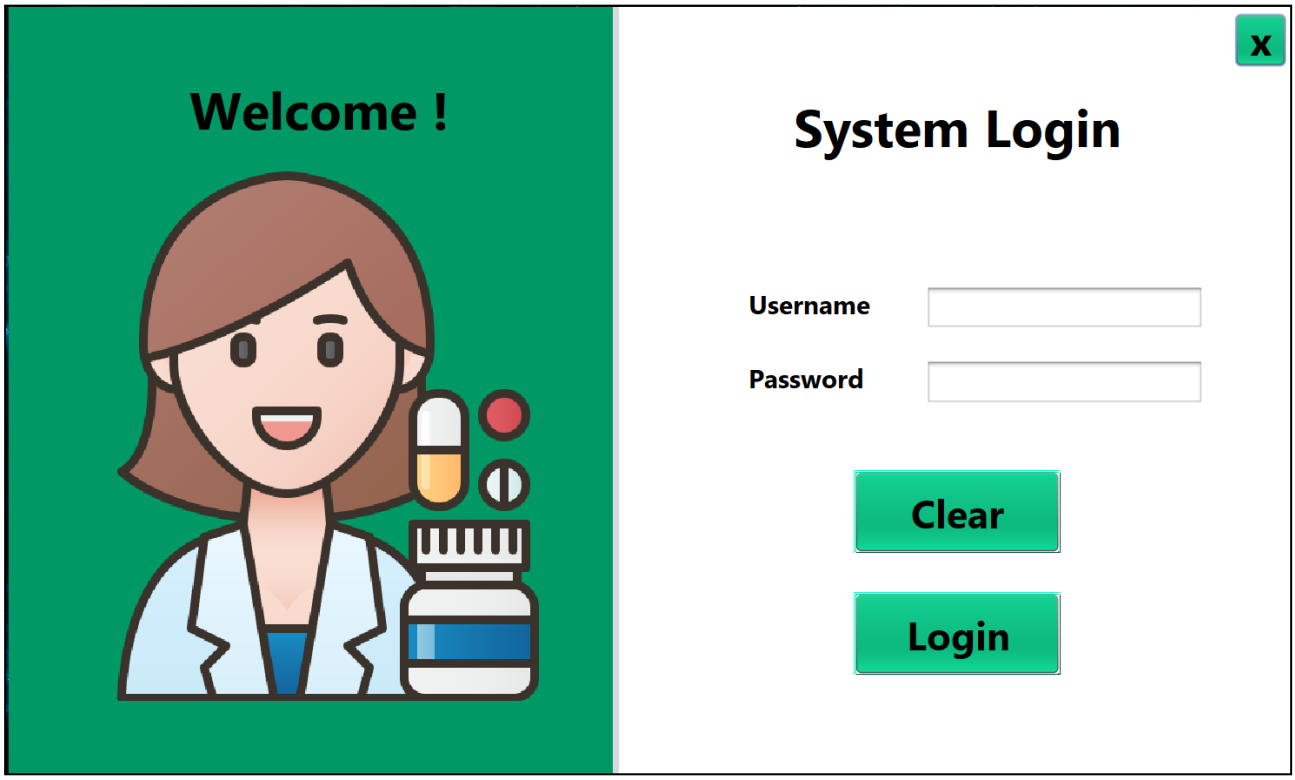

Figure 2.27: Login Page UI

```
public class Loginform extends javax.swing.JFrame
{
    // Creates Loginform
    // Define the correct username and password
    private static final String CORRECT_USERNAME = "zara123@gmail.com";
    private static final String CORRECT_PASSWORD = "123";

    public Loginform()
    {
        initComponents();
    }
    private void clearBtnActionPerformed(java.awt.event.ActionEvent evt) {
        // Clear button
        username.setText(t: "");
        pwd.setText(t: "");
    }

    private void loginBtnActionPerformed(java.awt.event.ActionEvent evt) {
        // Login button
        String enteredUsername = username.getText();
        String enteredPassword = pwd.getText();

        if (enteredUsername.equals(anObject: CORRECT_USERNAME) && enteredPassword.equals(anObject: CORRECT_PASSWORD))
        {
            // Successful login -> navigate to the Dashboard
            JOptionPane.showMessageDialog(parentComponent: this, message: "Login successful! Welcome Zara!", title: "Success", messageType: JOptionPane.INFORMATION_MESSAGE);
            Dashboard dashboard = new Dashboard();
            dashboard.setVisible(b: true);

            // Close the login form
            this.dispose();
        }
        else
        {
            // Invalid login -> display the error message
            JOptionPane.showMessageDialog(parentComponent: this, message: "Invalid username or password. Please try again.", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
        }
    }
```

Figure 2.28: Login Page User Authentication Code

❖ Explanation of Login Page User Credential Validation Code Functionality:

The “Loginform” class is a “top-level container” that offers a window in which the login interface is shown. It extends the “javax. swing. JFrame” class. A login form with the predefined, correct username and password has been assigned by the developer. (Since this system is only accessed by the Main Pharmacist) “zara123@gmail.com" is the username, and "123" is the password. These are both saved as private static final strings called “CORRECT_USERNAME” and “CORRECT_PASSWORD”. This assures that these values remain unchanged and cannot be modified during the execution of the program.

The “initComponents ()” function, which is normally in charge of configuring the GUI components including text fields and buttons, is called upon startup via the constructor public “Loginform ()”. The “loginBtnActionPerformed ()” method implements the action handling for the login button. The following actions are carried out by this method when the login button is clicked:

- Retrieving User Input: The password and username that the user gave are retrieved from the corresponding text boxes.
- Validation: the correct username and password are predefined, and the entered credentials are compared to those. The equals method, which checks for exact matches between the entered and correct values, is implemented to do this.
- Login Success: A message box indicating a successful login is displayed if the password and username entered match the correct credentials. (using “JOptionPane.showMessageDialog”) Then, calling dashboard creates and launches an instance of the Dashboard “dashboard. setVisible” to “true”. This is then employed to close the active login form. dispose () to hide the login page user interface and release resources.
- Login Failure: The user receives an error message saying that the username or password entered is incorrect if the credentials do not match.

This code segment enables a straightforward and effective user authentication procedure. Reusable code, like the Dashboard class and the “initComponents ()” method, improves code modularity and reusability by encapsulating particular functionalities that may be used in many portions of the program. Furthermore, the user interface for message dialogs is made smoother when “JOptionPane” is used.

❖ Dashboard

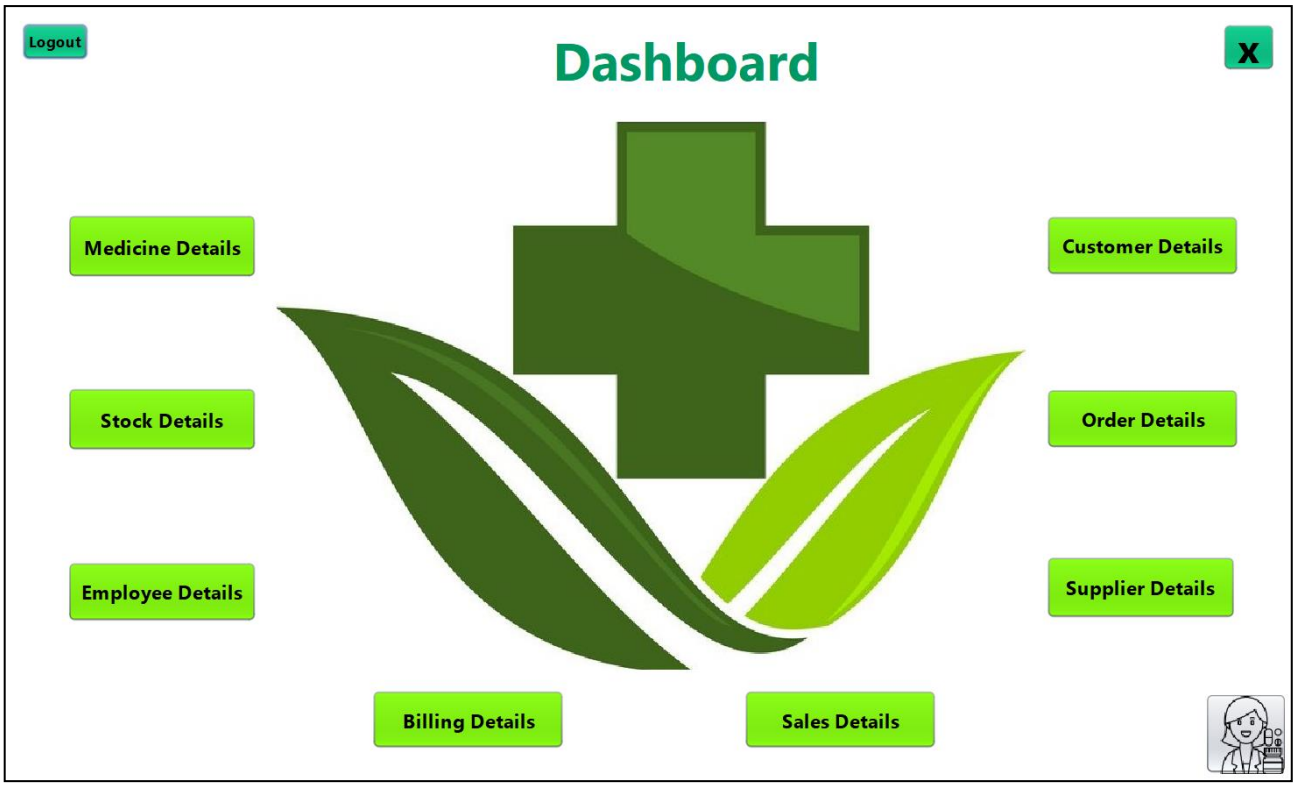

Figure 2.29: Dashboard UI

```
public class Dashboard extends javax.swing.JFrame
{

    public Dashboard()
    {
        initComponents();
    }

    private void medicineDetailsBtnActionPerformed(java.awt.event.ActionEvent evt) {
        // Medicine Details Button
        MedicineInventory obj = new MedicineInventory();
       obj.show();
       // Close the Dashboard
       this.dispose();
    }
private void employeeDetailsBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // Employee Details Button
    EmployeeDetails obj = new EmployeeDetails();
    obj.show();
    // Close the Dashboard
    this.dispose();
}

private void customerDetailsBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // Customer Details Button
    CustomerDetails obj = new CustomerDetails();
    obj.show();
    // Close the Dashboard
    this.dispose();
}
```

Figure 2.30: Dashboard Page Navigation Code

❖ Explanation of Dashboard Code's Page Navigation Functionality:

The Dashboard class extends "javax. swing. JFrame" and uses the Swing framework to create a graphical user interface (GUI) application. The main interface of the application is this Dashboard, from which users can click on different buttons to access other pages of the desktop application. Every button initiates an "ActionEvent", which is managed by the "ActionPerformed ()" method that corresponds to it. The associated method creates a new object of a particular class and displays its GUI when a button is clicked. For example, when you click the "Medicine Details" button, the "MedicineInventory class" is created, and the "show ()" function is used to display it. This is then used to close the current Dashboard from "this. dispose ()" method, which essentially switches the screen to the newly created interface.

The major modules, such as "MedicineInventory", "EmployeeDetails", "SupplierDetails", and others, are generally additional "JFrame sub-classes", each of which represents a distinct functional area of the program. Better code reuse could be achieved by abstracting the pattern of instantiating and displaying new user interfaces while discarding the current user interface. This is an example of reusable code.

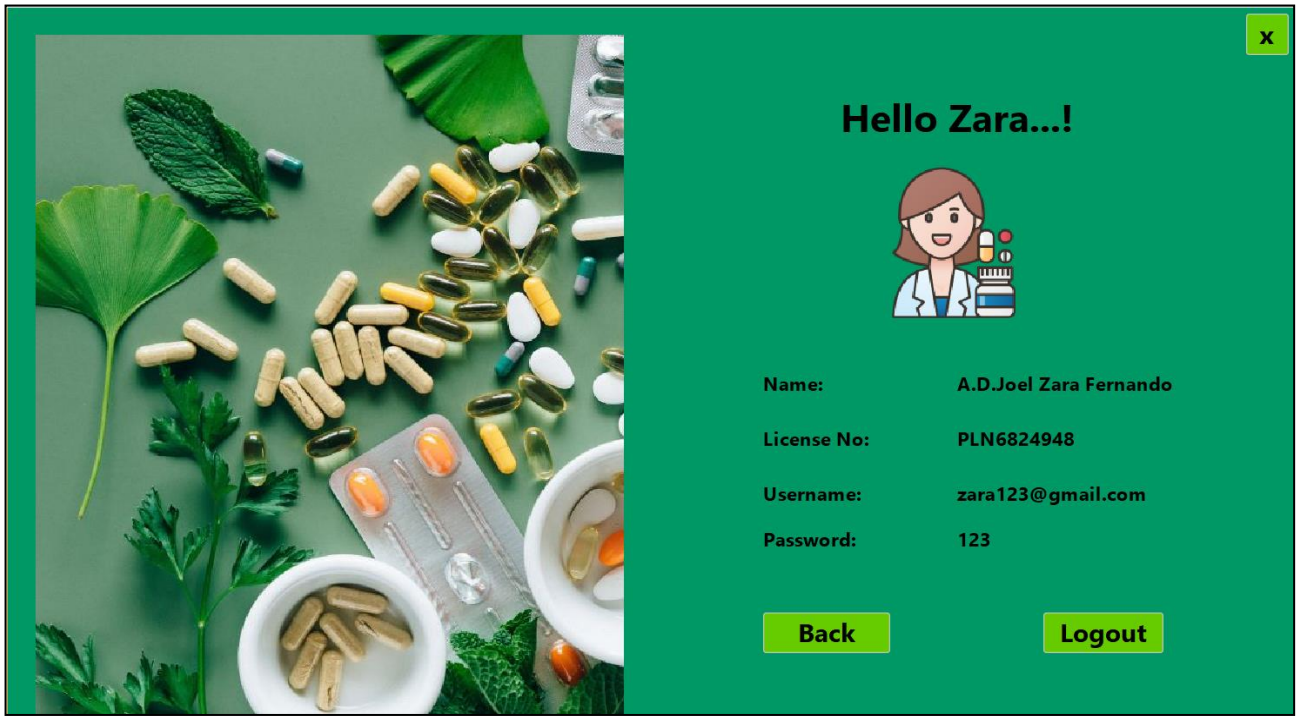

Figure 2.31: User Profile UI

1)Medicine Inventory Management Page (IT22364692 – KANDAGE K.T.S)

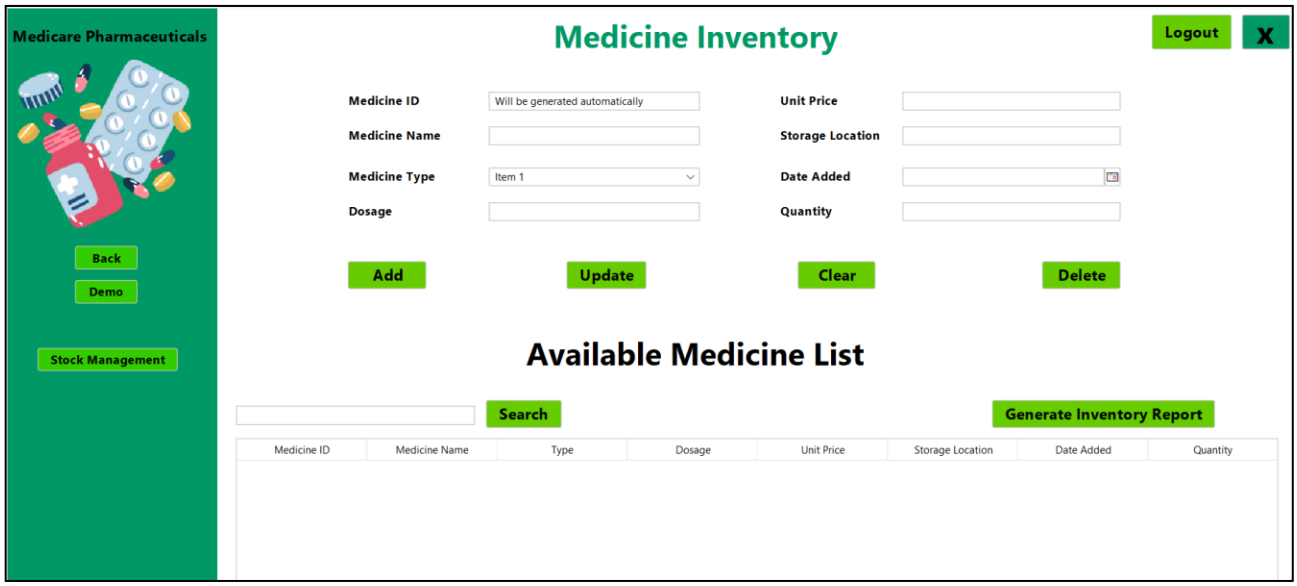

Figure 2.32: Medicine Inventory Management Page UI

This Medicine Inventory page functionalities aim to improve drug inventory management by addressing multiple key factors. Firstly, a great focus is placed on managing medicine inventories effectively, with the goal of creating streamlined processes for adding, modifying, and deleting medicines. The total efficiency of pharmaceutical item management is the focus of this strategy.

'MedicineInventory' presents a thorough graphical user interface (GUI) for controlling medicine inventory, which includes adding, updating, removing, searching, and printing reports. Important parts include methods for database CRUD operations, event handlers for user interactions, and utility methods for input validation and UI element filling. The application's maintainability and extensibility are guaranteed via reusable code patterns and consistent error handling.

Major Module Structures

❖ Constructor and Class

```
public class MedicineInventory extends javax.swing.JFrame {
//Creates new form MedicineInventory
        public MedicineInventory() {
        initComponents();
        refreshMedicineTable();
        //The method to populate the medicine type combo box
        getMedType();

        med_id.setText(t: "Will Be Added Automatically");
        medName.setText(t: "");
        med_dosage.setText(t: "");
        unitPrice.setText(t: "0");
        storageLocation.setText(t: "");
        date_added.setDate(date: Date.from(instant: Instant.now()));
        qty.setText(t: "1");
    }
```

Figure 2.33: Medicine Inventory Constructor and Class

This class acts as the medicine inventory's primary graphical user interface. Constructor Initializes the GUI elements, fills up the combo box and available medicine list table, and defines the fields' default values.

```
private void initComponents() {

    jPanel1 = new javax.swing.JPanel();
    jPanel2 = new javax.swing.JPanel();
    jLabel3 = new javax.swing.JLabel();
    medID = new javax.swing.JLabel();
    medID1 = new javax.swing.JLabel();
    medID4 = new javax.swing.JLabel();
    medID6 = new javax.swing.JLabel();
    medID7 = new javax.swing.JLabel();
    medID8 = new javax.swing.JLabel();
    medID9 = new javax.swing.JLabel();
    medID10 = new javax.swing.JLabel();
    med_id = new javax.swing.JTextField();
    medName = new javax.swing.JTextField();
    med_dosage = new javax.swing.JTextField();
    qty = new javax.swing.JTextField();
    unitPrice = new javax.swing.JTextField();
    storageLocation = new javax.swing.JTextField();
    updateBtn = new javax.swing.JButton();
    AddBtn = new javax.swing.JButton();
    deleteBtn = new javax.swing.JButton();
    clearBtn = new javax.swing.JButton();
    type = new javax.swing.JComboBox<>();
    logOutBtn = new javax.swing.JButton();
```

Figure 2.34: Medicine Inventory GUI Initialization

❖ Initializing the GUI (using initComponents): The GUI's text fields, tables, buttons, and other elements are configured via this method of configuration. The implementation is typically constructed automatically by a GUI builder program such as IntelliJ IDEA or NetBeans.

❖ Population of Combo Box: Fills the combo box for medicines with defined categories.

```
// Populate the medicine types combo box
private void getMedType() {
    DefaultComboBoxModel<String> model = new DefaultComboBoxModel<>();
    model.addElement(anObject: "Allergic");
    model.addElement(anObject: "Analgesics");
    model.addElement(anObject: "Angiotensin II Receptor");
    model.addElement(anObject: "Antibiotics");
    model.addElement(anObject: "Antidepressants");
    model.addElement(anObject: "Antidiabetics");
    model.addElement(anObject: "Antihypertensives");
    type.setModel(aModel:model);
}
```

Figure 2.35: Populate Medicine Type Combo Box

- **Reusable Code**: Anywhere a comparable combo box is required, this method may be used again.

❖ Supervisors of Events: Various event handlers for buttons and other components are included in the class. Here are few instances:

```
private void AddBtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Add Button
// Add Button action performed method

String med_Name, date_Added, medicineType, dosage, storage_Location;
int quantity, unit_Price;
int Newqty = 0;
med_Name = medName.getText();

// Get medicine type from the combo box
medicineType = (String) type.getSelectedItem();
dosage = med_dosage.getText();
storage_Location = storageLocation.getText();

//Validating the date formatting
SimpleDateFormat sdf = new SimpleDateFormat(pattern: "yyyy-MM-dd");
date_Added = sdf.format(date: date_added.getDate());

// User input validation
try {
    unit_Price = Integer.parseInt(s: unitPrice.getText());
    quantity = Integer.parseInt(s: qty.getText());

    // Confirmation message before adding
    int confirm = JOptionPane.showConfirmDialog(parentComponent: null, message: "Are you sure you want to 'Add' this medicine?", title: "Confirm Add", optionType: JOptionPane.YES_NO_OPTION);
    if (confirm != JOptionPane.YES_OPTION) {
        return; // If user clicks NO, do not proceed with the addition
    }

    // Validations of Database Interaction
    try {
        Statement st = pharmacy.DBconnection.createDBconnection().createStatement();

        // Fetch the next available Medicine ID from the Database
        ResultSet rs = st.executeQuery(string:"SELECT MAX(med_ID) FROM medicine");
        int nextMedID = 1;
```

Figure 2.36: Event handlers for buttons and other components

```
private void updateBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Update Button action performed method
    try {
        String med_Name, date_Added, medicineType, dosage, storage_Location;
        int quantity, unit_Price;
        int Newqty = 0;

        med_Name = medName.getText();
        // Get medicine type from the combo box
        medicineType = (String) type.getSelectedItem();
        dosage = med_dosage.getText();
        storage_Location = storageLocation.getText();

        //Validating the date formatting
        SimpleDateFormat sdf = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        date_Added = sdf.format(date: date_added.getDate());

        // User input validation
        quantity = Integer.parseInt(s: qty.getText());
        unit_Price = Integer.parseInt(s: unitPrice.getText());

        // Validations of Database Interaction
        try {
            Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
            int x = Integer.parseInt(s: med_id.getText());

            // Validations related to SQL
            String query = "UPDATE medicine SET med_name = ?, Type = ?, Dosage = ?, unit_price = ?, Storage_Location = ?, dateAdded = ?, qty = ? WHERE med_ID = ?";

            // Use PreparedStatement to avoid SQL injection
            java.sql.PreparedStatement prepSt = st.getConnection().prepareStatement(string:query);

            prepSt.setString(i: 1, string:med_Name);
            prepSt.setString(i: 2, string:medicineType);
            prepSt.setString(i: 3, string:dosage);
```

Figure 2.37: Medicine Inventory Update Button Code

```
private void deleteBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    int selectedRowIndex = medicineTable.getSelectedRow();
    if (selectedRowIndex != -1) {
        String medName = (String) medicineTable.getValueAt(row: selectedRowIndex, column: 1);

        int confirm = JOptionPane.showConfirmDialog(parentComponent: null, message: "Are you sure you want to 'Delete' this medicine?", title: "Confirm Deletion", optionType: JOptionPane.YES
        if (confirm == JOptionPane.YES_OPTION) {
            try {
                Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
                int count = st.executeUpdate("DELETE FROM medicine WHERE med_name='" + medName + "'");

                if (count > 0) {
                    JOptionPane.showMessageDialog(parentComponent: null, message: "Medicine deleted successfully.", title: "Success", messageType: JOptionPane.INFORMATION_MESSAGE);
                    refreshMedicineTable();
                }
            } catch (SQLException e) {
                JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while deleting medicine: " + e.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
            }
        }
    } else {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Please select a row to delete.", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }

}
```

Figure 2.37: Medicine Inventory Delete Button Code

These methods specify what happens, such adding, editing, or removing medicine records, when the relevant buttons are pressed.

❖ Interaction with Databases: The following code carries out CRUD (Create, Read, Update, Delete) actions by interacting with a database. Runs SQL queries with the intention of updating database entries.

```
// Validations of Database Interaction
try {
    Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
    // Fetch the next available Medicine ID from the Database
    ResultSet rs = st.executeQuery(string:"SELECT MAX(med_ID) FROM medicine");
    int nextMedID = 1;
    if (rs.next()) {
        nextMedID = rs.getInt(i: 1) + 1;)
    boolean isAlreadyExist = false;
    ResultSet rsExist = st.executeQuery("SELECT COUNT(*) AS count FROM `medicine` WHERE `med_name` = '" + med_Name + "' AND `Storage_Location` = '" + storage_Location + "'"
    if (rsExist.next()) {
        int countExist = rsExist.getInt(string:"count");
        if (countExist > 0) {
            isAlreadyExist = true;
        ))
    int count;
    if (!isAlreadyExist) {
        count = st.executeUpdate("INSERT INTO `medicine`(`med_ID`, `med_name`, `Type`,  `Dosage` , `unit_price`, `Storage_Location` , `dateAdded`, `qty`) VALUES ('" + nextM
    } else {
        ResultSet qtyResult = st.executeQuery("SELECT `qty` FROM `medicine` WHERE `med_name` = '" + med_Name + "' AND `Storage_Location` = '" + storage_Location + "'");
        if (qtyResult.next()) {
            Newqty = qtyResult.getInt(string:"qty");)
        Newqty += quantity;
        count = st.executeUpdate("UPDATE `medicine` SET `qty`='" + Newqty + "' WHERE `med_name` = '" + med_Name + "' AND `Storage_Location` = '" + storage_Location + "' ");
    } if (count > 0) {
        JOptionPane.showMessageDialog(parentComponent: null, message: "New Medicine Added Successfully", title: "Success", messageType: JOptionPane.INFORMATION_MESSAGE);
        // Refresh the table after adding the New Medicine Details
        refreshMedicineTable();
    }
} catch (SQLException ex) {
    JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while interacting with the database: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSA
}
} catch (NumberFormatException e) {
    JOptionPane.showMessageDialog(parentComponent: null, message: "Invalid Input for Quantity or Unit Price", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
}
```

Figure 2.38: Medicine Inventory Database Interaction

- **Reusable Code**: Several application components can make use of the same database connection and query execution process

❖ Report Preparation: A nested class called ‘ReportGenerator’ is in charge of producing printed reports. The ‘medicineTable's contents are printed to a printer for this purpose.

  - **Code Reusability**: Any table inside the program may be printed using the print function.

```
    private void InventoryReportActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        ReportGenerator.generatePrintedReport(table: medicineTable);
    }
public class ReportGenerator {

    public static void generatePrintedReport(JTable table) {
        PrinterJob printerJob = PrinterJob.getPrinterJob();

        if (printerJob.printDialog()) {
            printerJob.setJobName(jobName: "Medicine Inventory Report");
            printerJob.setPrintable(new Printable() {
                @Override
                public int print(Graphics graphics, PageFormat pageFormat, int pageIndex) throws PrinterException {
                    if (pageIndex > 0) {
                        return Printable.NO_SUCH_PAGE;
                    }

                    Graphics2D g2d = (Graphics2D) graphics;
                    g2d.translate(tx: pageFormat.getImageableX(), ty: pageFormat.getImageableY());

                    table.printAll(g: graphics);

                    return Printable.PAGE_EXISTS;
                }
            });
            try {
                printerJob.print();
                System.out.println(x: "Report sent to printer successfully!");
            } catch (PrinterException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

Figure 2.39: Medicine Inventory Report Generation

Reusable Codes

- A centralized way for generating database connections is 'pharmacy.DBconnection.createDBconnection()', which is reusable code for database connections.
- 'getMedType()' in the Combo Box Population can be used anywhere that choosing a medicine type is required.
- Table Refresh: 'refreshMedicineTable()' - This function may be used again to update any table that shows information regarding medicine.
- Validation and Error Management: Unified approaches for managing errors and validating input.

Tools for Development

- GUI Builder: The GUI is designed using programs like NetBeans or IntelliJ IDEA, which also generate many parts of the initComponents function automatically.
- JDBC: Java Database Connectivity, which allows you to communicate with SQL databases.
- Swing: The GUI component development framework in Java.
- Toedter Calendar: To choose a date, use the third-party library (com.toedter.calendar.JDateChooser).

Interaction of Prepared Statements and Databases

- Example: To avoid SQL injection in 'updateBtnActionPerformed', use 'PreparedStatement'

```
// Use PreparedStatement to avoid SQL injection
java.sql.PreparedStatement prepSt = st.getConnection().prepareStatement(string:query);

prepSt.setString(i: 1, string:med_Name);
prepSt.setString(i: 2, string:medicineType);
prepSt.setString(i: 3, string:dosage);
prepSt.setInt(i: 4, il: unit_Price);
prepSt.setString(i: 5, string:storage_Location);
prepSt.setString(i: 6, string:date_Added);
prepSt.setInt(i: 7, il: quantity);
prepSt.setInt(i: 8, il: x);

int count = prepSt.executeUpdate();
```

Figure 2.40: Use of Prepared Statements in Medicine Inventory Functionality

2) Stock Management Page (IT22364692 – KANDAGE K.T.S)

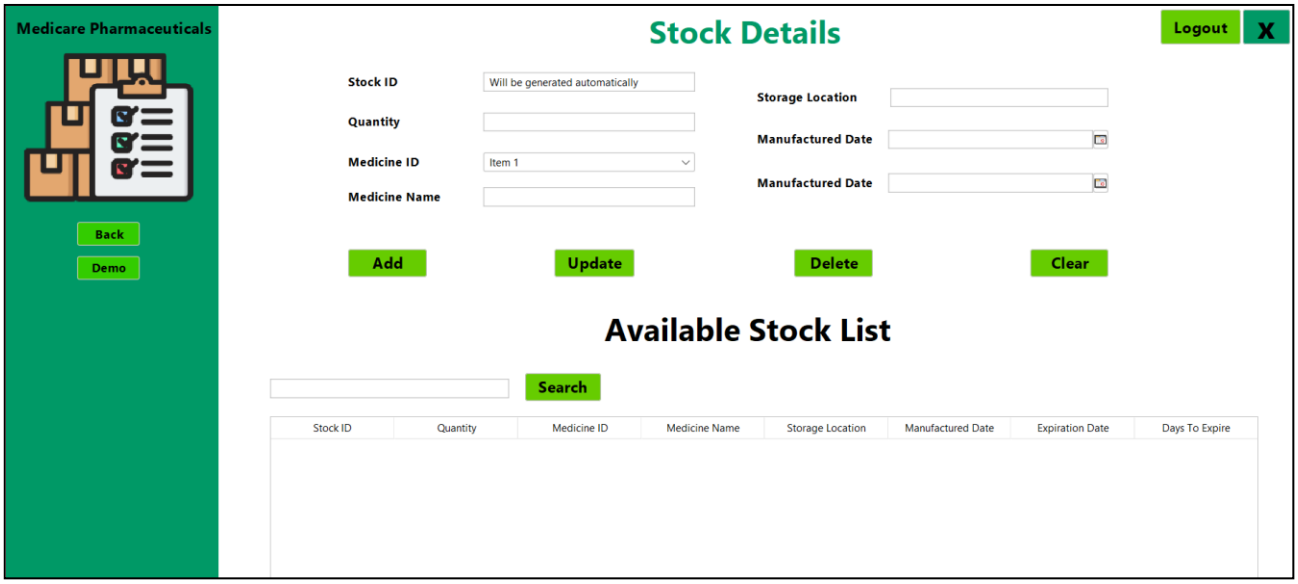

Figure 2.41: Stock Management Page UI

Stock Management page functionality offers features for adding, editing, deleting, and searching stock items in the inventory. It also shows stock data in a tabular style. The main objective of these functionalities is to reduce inconsistencies and improve the accuracy of stock information, the second shift in emphasis is to real-time data management. The project's goal is to put in place a flexible system that guarantees the availability of current data, enhancing decision-making procedures. Developing methods for reducing expenses and implementing waste-reduction, carrying-cost-cutting, and eventually profit-boosting policies in the drug inventory management system are important additional components. Finally, the sub-objectives focus on optimizing stock levels via effective management strategies to avoid overstocking and stockouts.

Major Module Structures

❖ Declaration of Class: The class generates a graphical user interface by extending 'javax.swing.JFrame'.

```
public class StocksDetails extends javax.swing.JFrame
```

Figure 2.42: Stock Management Class Declaration

❖ Constructor: The constructor provides default values for form fields, initializes the components, refreshes the stock database, and fills the medication ID combo box.

```
    public StocksDetails() {
    initComponents();
    refreshStockTable();

    //The method to populate the medicine ID combo box
    getMedicineIDs();
    addMedicineIDListener();


    stock_ID.setText(t: "Will Be Added Automatically");
    quantity.setText(t: "");
    medicine_name.setText(t: "0");
    storage_Location.setText(t: "");
    manufactured_date.setDate(new Date());
    expired_date.setDate(new Date());
    days_to_expire.setText(t: "");
}
```

Figure 2.43: Stock Management Class Constructor

❖ Database Connection: Creates a utility class ‘DBconnection’ connection in order to connect to the database.

```
Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
```

Figure 2.44: Stock Management Class Database Connection

- Create method to Add Medicine ID Listener : To change the medicine name and storage location when a new medicine ID is selected, add an action listener to the'medicine_ID' combo box.

```
    private void addMedicineIDListener() {
    medicine_ID.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            // When a new medicine ID is selected, populate the medicine name text field
            populateMedicineName();
            populateStorageLocation();
        }
    });
}
```

Figure 2.45: Medicine ID Combo Box Action Listener

- How to Fill in the Name and Storage Location of Medicines: The following methods do a database query to get the name and storage location of the chosen medicine ID.

```
    private void populateMedicineName() {
    try {
        int selectedMedicineID = Integer.parseInt((String) medicine_ID.getSelectedItem());
        // Query the database to retrieve the medicine name based on selected ID
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery("SELECT med_name FROM medicine WHERE med_ID = " + selectedMedicineID);
        if (rs.next()) {
            // Populate the medicine name text field with the retrieved name
            String medicineName = rs.getString(string:"med_name");
            medicine_name.setText(t: medicineName);
        }   rs.close();
            st.close();
        } catch (NumberFormatException | SQLException ex) {
            JOptionPane.showMessageDialog(parentComponent: null, "Error retrieving medicine name: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
        }
    }
    private void populateStorageLocation() {
try {
    int selectedMedicineID = Integer.parseInt(s: medicine_ID.getSelectedItem().toString());
    String storageLocation = ""; // Initialize storage location
    // Fetch storage location from the database based on the selected medicine ID
    String query = "SELECT Storage_Location FROM medicine WHERE med_ID = ?";
    PreparedStatement prepSt = DBconnection.createDBconnection().prepareStatement(string:query);
    prepSt.setInt(i: 1, i1: selectedMedicineID);
    ResultSet rs = prepSt.executeQuery();
    if (rs.next()) {
        storageLocation = rs.getString(string:"Storage_Location");
    }
    // Set the storage location in the storage_Location text field
    storage_Location.setText(t: storageLocation);
} catch (NumberFormatException | SQLException e) {
    // Handle exceptions
    JOptionPane.showMessageDialog(parentComponent: null, "Error fetching storage location: " + e.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
}
}
```

Figure 2.46: Get the name and storage location of the chosen medicine ID

- Create method for Populate Medicine IDs: Retrieves medicine IDs from the database and populates the ‘medicine_ID’ combo box.

```
// Populate the medicine IDs combo box
private void getMedicineIDs()
{
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery(string:"SELECT med_ID FROM medicine");

        //Clear existing items in the combo box
        medicine_ID.removeAllItems();

        //Populate combo box with medicine IDs
        while (rst.next())
        {
            //Store med_IDs in the combo box
            String item = String.valueOf(i: rst.getInt(string:"med_ID"));
            medicine_ID.addItem(item);
        }

        //Close resources
        rst.close();
        st.close();
        DBconnection.createDBconnection().close();
    }

    catch (SQLException ex)
    {
        //Handle exceptions
        JOptionPane.showMessageDialog(parentComponent: null, "Error fetching Medicine IDs: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.47: Populate Medicine IDs in Combo Box

- Methods of ActionListener

```
private void addBtn1ActionPerformed(java.awt.event.ActionEvent evt) {
// TODO add your handling code here:

String medicineName = medicine_name.getText();
String storageLocation = storage_Location.getText();
Date manufacturedDate = manufactured_date.getDate();
Date expiredDate = expired_date.getDate();
String daysToExpire = days_to_expire.getText();
int quantity = 0;
int medicineID = 0;

try {
    quantity = Integer.parseInt(s: this.quantity.getText());

    medicineID = Integer.parseInt(s: medicine_ID.getSelectedItem().toString());

    Statement st = pharmacy.DBconnection.createDBconnection().createStatement();

    ResultSet rs = st.executeQuery(string:"SELECT MAX(stock_ID) FROM stock");
    int nextStockID = 1;

    populateMedicineName();
    populateStorageLocation();
    if (rs.next()) {
        nextStockID = rs.getInt(i: 1) + 1;
    }

    boolean isAlreadyExist = false;
    ResultSet rsExist = st.executeQuery("SELECT COUNT(*) AS count FROM `medicine` WHERE `med_name` = '" + medicineName + "' AND `Storage_Location` = '" + storageLocation + "'");
    if (rsExist.next()) {
        int countExist = rsExist.getInt(string:"count");
        if (countExist > 0) {
            isAlreadyExist = true;
        }
    }
```

Figure 2.48: Action Listeners in Add Functionality

The user's button clicks and activities, including adding, updating, removing, and searching the stock table, are handled by these methods.

```
private void updateBtnActionPerformed(java.awt.event.ActionEvent evt) {
// TODO add your handling code here
        try {
        String med_Name, days_To_Expire, storageLocation, manufacturedDate, expiredDate;
        int Quantity, med_id;
        int Newqty = 0;
         int availableQTYinStock = 0;

        med_Name = medicine_name.getText();
        // Get medicine type from the combo box
        med_id = Integer.parseInt((String) medicine_ID.getSelectedItem());

        storageLocation = storage_Location.getText();

        populateMedicineName();
        populateStorageLocation();
        // Validating the date formatting
        SimpleDateFormat sdf = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        manufacturedDate = sdf.format(date: manufactured_date.getDate());

        SimpleDateFormat sdf1 = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        expiredDate = sdf1.format(date: expired_date.getDate());

        // User input validation
        Quantity = Integer.parseInt(s: quantity.getText());

        days_To_Expire = days_to_expire.getText();

        // Validations of Database Interaction
        try {
            // Create a connection and a prepared statement
            Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
            int x = Integer.parseInt(s: stock_ID.getText());

            String getBeforeValue = "select qty from stock where stock_ID ="+x+" ";
```

Figure 2.49: Code for Update Stock Details

```
private void deleteBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
     try {
    // Retrieve necessary information
    int med_id = Integer.parseInt((String) medicine_ID.getSelectedItem());
    int Quantity = Integer.parseInt(s: quantity.getText());
    int stock_id = Integer.parseInt(s: stock_ID.getText());
    int availableQTYinStock = 0;

    // Fetch current quantity in stock for the selected medicine
    String getBeforeValue = "SELECT qty FROM stock WHERE stock_ID = ?";
    PreparedStatement prepSt4 = pharmacy.DBconnection.createDBconnection().prepareStatement(string:getBeforeValue);
    prepSt4.setInt(i: 1, il: stock_id);
    ResultSet result4 = prepSt4.executeQuery();

    if(result4.next()){
        availableQTYinStock = result4.getInt(string:"qty");
    }

    // SQL query to delete the stock
    String deleteQuery = "DELETE FROM stock WHERE stock_ID = ?";
    PreparedStatement prepStDelete = pharmacy.DBconnection.createDBconnection().prepareStatement(string:deleteQuery);
    prepStDelete.setInt(i: 1, il: stock_id);

    // Execute the delete query
    int deleteCount = prepStDelete.executeUpdate();

    // If stock is deleted successfully, update medicine quantity
    if (deleteCount > 0) {
        // Calculate the new quantity in medicine table
        int newQty = availableQTYinStock - Quantity;

        // Update medicine table with new quantity
        String updateMedicineQuery = "UPDATE medicine SET qty = qty - ? WHERE med_ID = ?";
        PreparedStatement prepStUpdateMedicine = pharmacy.DBconnection.createDBconnection().prepareStatement(string:updateMedicineQuery);
        prepStUpdateMedicine.setInt(i: 1, il: Quantity);
```

Figure 2.50: Code for Delete Stock Details

```
    private void clearBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:

    // Clear all form fields and reset them to default values
    stock_ID.setText(t: "Will Be Added Automatically");
    quantity.setText(t: "");
    medicine_ID.setSelectedIndex(anIndex: -1); // Clear the selection
    medicine_name.setText(t: ""); // Clear the medicine name field
    storage_Location.setText(t: ""); // Clear the storage location field
    manufactured_date.setDate(new Date());
    expired_date.setDate(new Date());
    days_to_expire.setText(t: "");


    }
```

Figure 2.51: Code for Clear User Input fields of Stock Details

```
    private void demoBtnActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        quantity.setText(t: "150");
        storage_Location.setText(t: "A602");
        days_to_expire.setText(t: "365 days");

    }
```

Figure 2.52: Code of the Demo Button

```
private void stock_tableMouseClicked(java.awt.event.MouseEvent evt) {
    //Get the index of the selected row
    int selectedRow = stock_table.getSelectedRow();
    //Establish the table model that represents the structure & data of the table
    TableModel model = stock_table.getModel();
    //Validations to ensure that User interacts with the selected row data
    //Set the First Name,Last Name,Date Of Birth,Gender,Email,Job Position,Joined Date,Salary & Contact Number text fields to the value from the selected row
    quantity.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 1).toString());
    // Set selected job role in the combo box
    String selectedMediID = model.getValueAt(rowIndex: selectedRow, columnIndex: 2).toString();
    DefaultComboBoxModel<String> comboBoxModel = (DefaultComboBoxModel<String>) medicine_ID.getModel();
    if (comboBoxModel != null)
    {  medicine_ID.setSelectedItem(anObject: selectedMediID); }
    medicine_name.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 3).toString());
    storage_Location.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 4).toString());
    //Set the Date Format for ManufacturedDate
    SimpleDateFormat sdf = new SimpleDateFormat(pattern: "yyyy-MM-dd");
    String manufacturedDate = (String) model.getValueAt(rowIndex: selectedRow, columnIndex: 5);
    //Set the Date Format for ExpiredDate
    SimpleDateFormat sdf1 = new SimpleDateFormat(pattern: "yyyy-MM-dd");
    String expiredDate = (String) model.getValueAt(rowIndex: selectedRow, columnIndex: 6);
    //Retrieve the Stock ID from the First Column(index 0)of the selected row & store it
    stock_ID.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 0).toString());
        try { //Handle the parsing of the Birth Date
        Date manf_Date = sdf.parse(source:manufacturedDate);
        manufactured_date.setDate(date: manf_Date);
        //Handle the parsing of the Joined Date
        Date exp_Date = sdf1.parse(source:expiredDate);
        expired_date.setDate(date: exp_Date);
        // Disable Date of Birth and Start Date fields when user tries to update an existing data tuple
        manufactured_date.setEnabled(bln:false);
        expired_date.setEnabled(bln:false);  }
        catch (ParseException e)
        {  e.printStackTrace(); } // Handle the parsing exception
        days_to_expire.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 7).toString());
}
```

Figure 2.53: Retrieve the details of the selected row of the Stock Table

```
    private void stockSearchBarKeyPressed(java.awt.event.KeyEvent evt) {
        // TODO add your handling code here:
        // Get the text from the search bar
    String searchText = stockSearchBar.getText().trim();

    // Get the table model of your medicine table
    DefaultTableModel model = (DefaultTableModel) stock_table.getModel();

    // Create a row sorter for the table model
    TableRowSorter<DefaultTableModel> rowSorter = new TableRowSorter<>(model);

    // Set the row sorter to the table
    stock_table.setRowSorter(sorter:rowSorter);

    // Apply the filter to show rows that contain the search text
    if (searchText.length() == 0) {
        // If the search text is empty, show all rows
        rowSorter.setRowFilter(filter:null);
    } else {
        // Otherwise, show rows that contain the search text
        rowSorter.setRowFilter(filter:RowFilter.regexFilter("(?i)" + searchText)); // Case insensitive filter
    }
    }
```

Figure 2.54: Search for a Stock

- Method of Refresh Table: By requesting the most recent stock data from the database, the above method refreshes the data shown in the stock table.

```
// Assuming you have a method to populate your stock table
private void refreshStockTable()
{try
    {   // Fetch updated data from the Database
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery(string:" SELECT * FROM stock ");
        // Create a new DefaultTableModel with updated data
        DefaultTableModel model = new DefaultTableModel();
        model.addColumn(columnName:"stock_ID");
        model.addColumn(columnName:"qty");
        model.addColumn(columnName:"medID");
        model.addColumn(columnName:"medName");
        model.addColumn(columnName:"storage_location");
        model.addColumn(columnName:"manf_date");
        model.addColumn(columnName:"exp_date");
        model.addColumn(columnName:"daysToExpire");
            while (rs.next()) {
                Object[] row = {
                    rs.getInt(string:"stock_ID"),
                    rs.getInt(string:"qty"),
                    rs.getInt(string:"medID"),
                    rs.getString(string:"medName"),
                    rs.getString(string:"storage_location"),
                    rs.getString(string:"manf_date"),
                    rs.getString(string:"exp_date"),
                    rs.getString(string:"daysToExpire") };
            model.addRow(rowData:row);
        }
        // Set the updated model to the medicineTable
        stock_table.setModel(dataModel:model);
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while refreshing Table: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.55: Refresh the Stock Table after every table data Manipulation

Development Tools & Reusable Code

❖ Development Tools

- "Swing Framework": The graphical user interface is constructed using the "Java Swing Framework."
- "MySQL Database": The database management system used to store and retrieve stock data is called "MySQL."
- ‘PreparedStatement’:To stop SQL injection threats, parameterized SQL queries are executed using ‘PreparedStatements’.

```
// Use PreparedStatement to avoid SQL injection
java.sql.PreparedStatement prepSt = st.getConnection().prepareStatement(string:query);
```

Figure 2.56: Prepared Statement related to Database connectivity

❖ Reusable Codes

- ‘Database Connection’:

```
// Query the database to retrieve the medicine name based on selected ID
Statement st = DBconnection.createDBconnection().createStatement();

ResultSet rs = st.executeQuery("SELECT med_name FROM medicine WHERE med_ID = " + selectedMedicineID);
// Populate the medicine IDs combo box
   private void getMedicineIDs()
   {
       try
       {
           Statement st = DBconnection.createDBconnection().createStatement();
           ResultSet rst = st.executeQuery(string:"SELECT med_ID FROM medicine");
          Statement st = pharmacy.DBconnection.createDBconnection().createStatement();

          ResultSet rs = st.executeQuery(string:"SELECT MAX(stock_ID) FROM stock");
          private void refreshStockTable()

           {
              try
              {
                  // Fetch updated data from the Database
                  Statement st = DBconnection.createDBconnection().createStatement();
                  ResultSet rs = st.executeQuery(string:" SELECT * FROM stock ");
```

Figure 2.57: Fetching data from the system database

- Formatting Dates: When formatting date objects for database storage and retrieval, date formatting is applied uniformly across all methods.

```
// Validating the date formatting
SimpleDateFormat sdf = new SimpleDateFormat(pattern: "yyyy-MM-dd");
manufacturedDate = sdf.format(date: manufactured_date.getDate());

SimpleDateFormat sdf1 = new SimpleDateFormat(pattern: "yyyy-MM-dd");
expiredDate = sdf1.format(date: expired_date.getDate());
```

Figure 2.58: Formatting Dates in Stock Management

Detailed Description of the Code

❖ Components of Initializing the GUI: The GUI's basic attributes and components are built up using this method of construction.

```
@SuppressWarnings("unchecked")
// <editor-fold defaultstate="collapsed" desc="Generated Code">
private void initComponents() {

    jScrollPane1 = new javax.swing.JScrollPane();
    jTable1 = new javax.swing.JTable();
    jScrollPane2 = new javax.swing.JScrollPane();
    jTable2 = new javax.swing.JTable();
    jPanel1 = new javax.swing.JPanel();
    jPanel2 = new javax.swing.JPanel();
    jLabel3 = new javax.swing.JLabel();
```

Figure 2.59: Stock Management Page Components Initialization

❖ Filled Up Medical IDs: This procedure looks up medicine IDs in the database and adds them to the medicine_ID combo box.

```
// Populate the medicine IDs combo box
private void getMedicineIDs()
{
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery(string:"SELECT med_ID FROM medicine");

        //Clear existing items in the combo box
        medicine_ID.removeAllItems();

        //Populate combo box with medicine IDs
        while (rst.next())
        {
            //Store med_IDs in the combo box
            String item = String.valueOf(i: rst.getInt(string:"med_ID"));
            medicine_ID.addItem(item);
        }

        //Close resources
        rst.close();
        st.close();
        DBconnection.createDBconnection().close();
    }

    catch (SQLException ex)
    {
        //Handle exceptions
        JOptionPane.showMessageDialog(parentComponent: null, "Error fetching Medicine IDs: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.60: Fills Up Medical ID Combo Box with the existing medicine IDs

❖ Add a New Stock : Using this approach, a new stock item is added to the database, data is gathered from the form fields, and inputs are verified.

```
private void addBtnlActionPerformed(java.awt.event.ActionEvent evt) {
// TODO add your handling code here:

String medicineName = medicine_name.getText();
String storageLocation = storage_Location.getText();
Date manufacturedDate = manufactured_date.getDate();
Date expiredDate = expired_date.getDate();
String daysToExpire = days_to_expire.getText();
int quantity = 0;
int medicineID = 0;

try {
    quantity = Integer.parseInt(s: this.quantity.getText());

    medicineID = Integer.parseInt(s: medicine_ID.getSelectedItem().toString());

    Statement st = pharmacy.DBconnection.createDBconnection().createStatement();

    ResultSet rs = st.executeQuery(string:"SELECT MAX(stock_ID) FROM stock");

    int nextStockID = 1;

    populateMedicineName();
    populateStorageLocation();
    if (rs.next()) {
        nextStockID = rs.getInt(i: 1) + 1;
    }

    boolean isAlreadyExist = false;
    ResultSet rsExist = st.executeQuery("SELECT COUNT(*) AS count FROM `medicine` WHERE `med_name` = '" + medicineName + "' AND `Storage_Location` = '" + storageLocation + "'"
    if (rsExist.next()) {
        int countExist = rsExist.getInt(string:"count");
        if (countExist > 0) {
            isAlreadyExist = true;
        }
    }
```

Figure 2.61: Insert details of a New Stock

❖ Update already added stock: This function uses the user-provided inputs to change the details of an existing stock item.

```
private void updateBtnActionPerformed(java.awt.event.ActionEvent evt) {
// TODO add your handling code here
try {
    String med_Name, days_To_Expire, storageLocation, manufacturedDate, expiredDate;
    int Quantity, med_id;
    int Newqty = 0;
    int availableQTYinStock = 0;
    med_Name = medicine_name.getText();
    // Get medicine type from the combo box
    med_id = Integer.parseInt((String) medicine_ID.getSelectedItem());
    storageLocation = storage_Location.getText();
    populateMedicineName();
    populateStorageLocation();
    // Validating the date formatting
    SimpleDateFormat sdf = new SimpleDateFormat(pattern: "yyyy-MM-dd");
    manufacturedDate = sdf.format(date: manufactured_date.getDate());
    SimpleDateFormat sdf1 = new SimpleDateFormat(pattern: "yyyy-MM-dd");
    expiredDate = sdf1.format(date: expired_date.getDate());
    // User input validation
    Quantity = Integer.parseInt(s: quantity.getText());
    days_To_Expire = days_to_expire.getText();
    // Validations of Database Interaction
    try {
        // Create a connection and a prepared statement
        Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
        int x = Integer.parseInt(s: stock_ID.getText());
        String getBeforeValue = "select qty from stock where stock_ID ="+x+" ";
    java.sql.PreparedStatement prepSt4 = st.getConnection().prepareStatement(string:getBeforeValue);
    ResultSet result4 = prepSt4.executeQuery();
    if(result4.next()){
        availableQTYinStock = result4.getInt(string:"qty");
    }
        java.sql.PreparedStatement prepSt = st.getConnection().prepareStatement(string:query);
        prepSt.setInt(i: 1, i1: Quantity);
        prepSt.setInt(i: 2, i1: med_id);
        prepSt.setString(i: 3, string:med_Name);
        prepSt.setString(i: 4, string:storageLocation);
        prepSt.setString(i: 5, string:manufacturedDate);
        prepSt.setString(i: 6, string:expiredDate);
        prepSt.setString(i: 7, string:days_To_Expire);
        prepSt.setInt(i: 8, i1: x);
        int count = prepSt.executeUpdate();
    if (count > 0) {
    //JOptionPane.showMessageDialog(null, "Selected Stock Details Updated Successfully", "Success", JOptionPane.INFORMATION_MESSAGE);
    String selectQuery1 = "Select qty from medicine where med_ID = "+med_id+"";
    int availableQTY = 0;
     java.sql.PreparedStatement prepSt2 = st.getConnection().prepareStatement(string:selectQuery1);
    ResultSet result = prepSt2.executeQuery();
    if(result.next()){
        availableQTY = result.getInt(string:"qty");
    }
    int temp = (availableQTY-availableQTYinStock)+Quantity;
    String query1 = "UPDATE medicine SET qty = "+temp+" WHERE med_ID = "+med_id+" ";
    java.sql.PreparedStatement prepSt1 = st.getConnection().prepareStatement(string:query1);
    int count1 = prepSt1.executeUpdate();
    if (count1 > 0) {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Stock updated successfully and Medicine quantity updated", title: "Success", messageType: JOptionPane.INFORMATION_MESSAGE);
        refreshStockTable();
    }
    // Refresh the Stock Table after updating
    }
    else {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Failed to update Stock Details", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
        }
    } catch (SQLException e) {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while Updating the selected stock's details: " + e.getMessage(), title: "Error", messageType: JOptionPane.ERROR_ME
    }
    } catch (NumberFormatException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Invalid Input for Quantity or Medicine ID!", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
```

Figure 2.62: modify details of a Stock

- Deleting stock in the table: Using this method, a chosen stock item is removed from the database and the medicine amount is adjusted appropriately.

```
private void deleteBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
     try {
    // Retrieve necessary information
    int med_id = Integer.parseInt((String) medicine_ID.getSelectedItem());
    int Quantity = Integer.parseInt(s: quantity.getText());
    int stock_id = Integer.parseInt(s: stock_ID.getText());
    int availableQTYinStock = 0;
    // Fetch current quantity in stock for the selected medicine
    String getBeforeValue = "SELECT qty FROM stock WHERE stock_ID = ?";
    PreparedStatement prepSt4 = pharmacy.DBconnection.createDBconnection().prepareStatement(string:getBeforeValue);
    prepSt4.setInt(i: 1, il: stock_id);
    ResultSet result4 = prepSt4.executeQuery();
    if(result4.next()){
        availableQTYinStock = result4.getInt(string:"qty");
    }
    // SQL query to delete the stock
    String deleteQuery = "DELETE FROM stock WHERE stock_ID = ?";
    PreparedStatement prepStDelete = pharmacy.DBconnection.createDBconnection().prepareStatement(string:deleteQuery);
    prepStDelete.setInt(i: 1, il: stock_id);
    // Execute the delete query
    int deleteCount = prepStDelete.executeUpdate();
    // If stock is deleted successfully, update medicine quantity
    if (deleteCount > 0) {
        // Calculate the new quantity in medicine table
        int newQty = availableQTYinStock - Quantity;
        // Update medicine table with new quantity
        String updateMedicineQuery = "UPDATE medicine SET qty = qty - ? WHERE med_ID = ?";
        PreparedStatement prepStUpdateMedicine = pharmacy.DBconnection.createDBconnection().prepareStatement(string:updateMedicineQuery);
        prepStUpdateMedicine.setInt(i: 1, il: Quantity);
        prepStUpdateMedicine.setInt(i: 2, il: med_id);
        // Execute the update query for medicine table
        int updateMedicineCount = prepStUpdateMedicine.executeUpdate();
        // Refresh stock table after successful deletion
        if (updateMedicineCount > 0) {
            refreshStockTable();
            JOptionPane.showMessageDialog(parentComponent: null, message: "Stock deleted successfully and Medicine quantity updated.", title: "Success", messageType: JOptionPane.INFORMATION
```

Figure 2.63: Delete a Stock

- Refreshing the Stock Table: By changing the table model and retrieving the most recent data from the database, this method refreshes the stock table.

```
// Assuming you have a method to populate your stock table
private void refreshStockTable()
 {
    try
    {
        // Fetch updated data from the Database
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery(string:" SELECT * FROM stock ");
        // Create a new DefaultTableModel with updated data
        DefaultTableModel model = new DefaultTableModel();
        model.addColumn(columnName: "stock_ID");
        model.addColumn(columnName: "qty");
        model.addColumn(columnName: "medID");
        model.addColumn(columnName: "medName");
        model.addColumn(columnName: "storage_location");
        model.addColumn(columnName: "manf_date");
        model.addColumn(columnName: "exp_date");
        model.addColumn(columnName: "daysToExpire");
        while (rs.next()) {
            Object[] row = {
                    rs.getInt(string:"stock_ID"),
                    rs.getInt(string:"qty"),
                    rs.getInt(string:"medID"),
                    rs.getString(string:"medName"),
                    rs.getString(string:"storage_location"),
                    rs.getString(string:"manf_date"),
                    rs.getString(string:"exp_date"),
                    rs.getString(string:"daysToExpire")
            };
            model.addRow(rowData: row);
        }       // Set the updated model to the medicineTable
                stock_table.setModel(dataModel:model);  }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while refreshing Table: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.64: Refresh Stock Table After a modification

3) Supplier Management Page (IT22319142 – WIJESINGHE A.G.T)

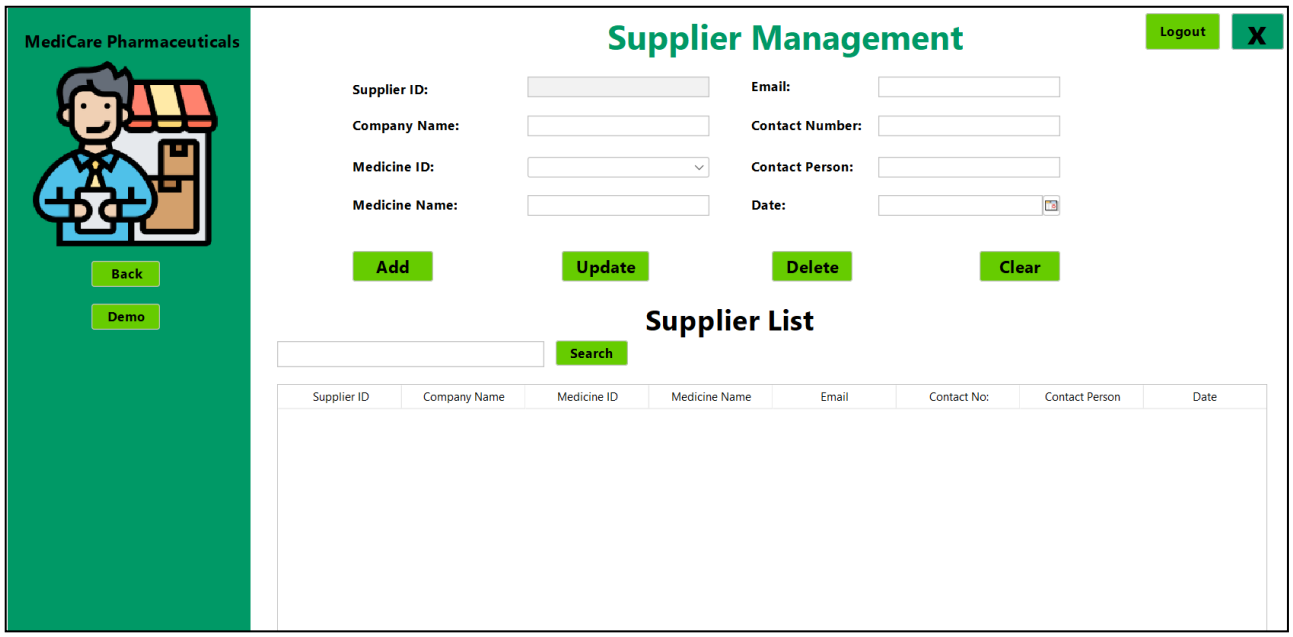

Figure 2.65: Supplier Management Page UI

The objective of this key business function, "Supplier Management function", is to improve operational efficiency. The pharmacist can easily register new suppliers and link them to certain medicine IDs that they are supplying, because in this company a specific supplier only supplies a specific medicine. The system provides an easy-to-use capability for updating supplier details and allows inactive suppliers to be permanently deleted from the system database. Pharmacists can type the supplier ID into a search field to filter and display specific supplier details.

This function;

- Improves supplier relationships by allowing users to do supplier registration, update details, search & delete the added details.
- Simplifies personnel administration, including keeping track of attendance and paying salaries.
- Introduces the ability to use automated comprehensive reports to analyze monthly salaries according to the monthly attendance.

❖ Supplier Management Page's Initial Code:

```
package pharmacy;
// @author thath

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.FocusAdapter;
import java.awt.event.FocusEvent;
import java.sql.Statement;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.swing.JOptionPane;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.text.ParseException;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableModel;

public class SupplierDetails extends javax.swing.JFrame
{
// Creates new form SupplierDetails

    public SupplierDetails()
    {
        initComponents();
        refreshSuppliersTable();

        //The method to populate the med_id combo box
        getMedicineIDs();

        // Setting today's date as the only selectable date in the JDateChooser
        Date today = new Date();
        Date.setDate(date: today);
        Date.setMinSelectableDate(date: today);
        Date.setMaxSelectableDate(date: today);

        // Adding a focus listener to contactPerson field
        contactPerson.addFocusListener(new FocusAdapter()
        {
            @Override
            public void focusLost(FocusEvent e)
            {
                validateContactNumber();
            }
        });
        med_id.addActionListener(new ActionListener()
        {
            @Override
            public void actionPerformed(ActionEvent e)
            {
                int selectedMedID = Integer.parseInt(s: med_id.getSelectedItem().toString());
                String medicineName = getMedicineNameByMedID(medicineID:selectedMedID);
                med_Name.setText(t: medicineName);
            }
        });

        //ActionListener to the Demo button
        demo_btn.addActionListener(new ActionListener()
        {
            @Override
            public void actionPerformed(ActionEvent e)
            {
                fillDemoData();
            }
    });

        id.setText(t: "Will Be Added Automatically");
        company.setText(t: "");
        med_Name.setText(t: "");
        email.setText(t: "@gmail.com");
        contactNo.setText(t: "");
        contactPerson.setText(t: "");
        Date.setDate(new java.util.Date());
    }
```

Figure 2.66: Supplier Management Page Component Initialization Code

Explanation of Supplier Management Page's Functionality:

The supplier details management user interface is provided by this class. It uses the "initComponents ()" function to initialize different components, including text fields, buttons, and date pickers, upon instantiation. It additionally performs a call to "refreshSuppliersTable ()" to load current supplier data into a table. In order to connect specific medicines with suppliers who supplies them, the function "getMedicineIDs ()" is called in order to fill a combo box (med_id) with medicine IDs.

The user is essentially prevented from selecting dates other than the current day by the configuration of the date picker (JDateChooser), which only allows the date of today to be chosen. (*To* ensure a user-friendly interface by restricting date selection.)

When attention is removed from the "contactPerson field", an event listener (FocusAdapter) is introduced to verify the contact number.

The "med_id combo box" has an additional event listener (ActionListener) associated with it. This listener causes an action to be triggered, fetching and displaying the name of the selected medicine in the "med_Name text field". (ensure user-friendliness of the interface by auto prefilling certain fields) Additionally, by clicking the "demo_btn" button, the action which has attached to it, causes the form to be filled with demo data.

Swing components are used by the class for its graphical user interface. It imports and uses classes from the "java.sql package" to handle database operations and run SQL queries, interacting with a database, for the purpose of storing and retrieving supplier information.

❖ Explanation of Supplier Management Page's Contact No Validation Functionality:

```
// Validate contact number and display error if not 10 digits
private void validateContactNumber()
{
    String contactNumber = contactNo.getText().trim();
    if (contactNumber.length() != 10)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Enter exactly 10 numerical values for the contact number",
                title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
        contactNo.requestFocus(); // Set focus back to contactNo field
    }
}
```

Figure 2.67: Supplier Management Page Contact No Validation Code

Using this method, when a contact number entered into a user interface it is validated. It specifically ensures the contact number entered has precisely ten digits. The user receives an error message, and the input field is refocused to allow them to correct their input if the contact number if not exactly ten digits.

❖ Explanation of Supplier Management Page's "med_id Combo Box Population" Functionality:

```
private void getMedicineIDs()
{
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery(string: "SELECT med_ID FROM medicine");

        // Clear existing items in the combo box
        med_id.removeAllItems();

        // Populate combo box with customer IDs
        while (rst.next())
        {
            // Store med_IDs in the combo box
            String item = String.valueOf(i: rst.getInt(string: "med_ID"));
            med_id.addItem(item);
        }

        // Close resources
        rst.close();
        st.close();
        DBconnection.createDBconnection().close();
    }
    catch (SQLException ex)
    {
        // Handle exceptions
        JOptionPane.showMessageDialog(parentComponent: null, "Error fetching Customer IDs: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.68: Supplier Management Page Combo box Population Code

This method selects all medicine IDs (med_ID) from the medicine table by creating a Statement object via a database connection and executing a SQL query on a "ResultSet" object. "Med_id. deleteAllItems ()" is used to clear the combo box before adding new items.

Each medicine ID is retrieved iteratively from the result set, transformed into a string, and then added to the combo box. In order to stop resource leaks, the database resources (ResultSet, Statement, and database connection) are closed at the end. In the event of a "SQLException", a "JOptionPane" is used to capture the exception and display an error message.

❖ Adding a New Supplier Functionality:

```
private void addBtnActionPerformed(java.awt.event.ActionEvent evt) {
    String companyName, medicineName, emailAddress, contact_Person, dateAdded;
    int medicineID, contactNumber;

    companyName = company.getText().trim().toLowerCase();
    emailAddress = email.getText().trim().toLowerCase();
    contact_Person = contactPerson.getText().trim();

    // Validate that all required user input fields are filled
    if (companyName.isEmpty() || emailAddress.isEmpty() || contact_Person.isEmpty() || med_id.getSelectedItem() == null)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Please fill all the required fields!",
            title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
        return; // Exit if any required user input field is blank
    }

    // Validate contact number before proceeding
    if (contactNo.getText().trim().length() != 10)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Enter exactly 10 numerical values for the contact number",
                title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
        return; // Exit method if contact number is invalid
    }

    // Validate email address
    if (!emailAddress.endsWith(suffix: "@gmail.com") || emailAddress.equals(anObject: "@gmail.com"))
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Enter a valid Gmail address", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
        return;
    }

    // Check if the company with the same name and email already exists
    if (isCompanyAlreadyRegistered(companyName, emailAddress))
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "The company you are trying to Add is already registered in the database!", title:"Duplicate Company", messageType: JOptionPane.ERROR_MESSAGE);

        // Exit the method
        return;
    }
    medicineID = Integer.parseInt(s: med_id.getSelectedItem().toString());
    contactNumber = Integer.parseInt(s: contactNo.getText());
    dateAdded = new SimpleDateFormat(pattern: "yyyy-MM-dd").format(date: Date.getDate());

    // Get the medicine name from the database
    medicineName = getMedicineNameByMedID(medicineID);
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery(string: "SELECT MAX(sup_ID) FROM supplier");
        int nextSupplierID = 1;

        if (rs.next())
        {
            nextSupplierID = rs.getInt(i: 1) + 1;
        }

        int count = st.executeUpdate("INSERT INTO `supplier`(`sup_ID`, `med_ID`, `medName`,  `company` , `contact_person`, `email`, `phone_no` , `date`) VALUES ('" + nextSupplierID + "','" + medicineID + "','"

        if (count > 0)
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "New Supplier's Details Added Successfully", title: "Success", messageType: JOptionPane.INFORMATION_MESSAGE);
            refreshSuppliersTable();
        }
    } catch (SQLException e)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error: " + e.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.69: Adding a New Supplier to the System Database Functionality Code

❖ Explanation of Adding a New Supplier Functionality:

Adding new supplier details to a database is handled by this method. When the user clicks the "Add" button, the "addBtnActionPerformed ()" method is called. It then operates the tasks as follows: it collects input data from multiple text fields, validates the input (making sure the contact number is exactly ten digits, the email is a valid Gmail address, and furthermore), and determines whether the company is already registered in the database. Then the new supplier details are inserted into the database if all the above validations are successful.

Additionally, a new supplier ID is generated by the process by incrementing the maximum existing supplier ID through a query. It retrieves from the database the medicine name that corresponds with the selected medicine ID. It displays a success message and updates the supplier table with the new entry when insertion is successful. The user receives the relevant error messages if any errors arise during these steps.

The other functions "isCompanyAlreadyRegistered ()", "getMedicineNameByMedID ()", and "refreshSuppliersTable ()" each perform a particular function, such as updating the suppliers table, retrieving the medicine name from the database, and detecting duplicate business entries as follows.

```
//Checking if the company with the same name and email already exists
private boolean isCompanyAlreadyRegistered(String companyName, String emailAddress)
{
    boolean companyExists = false;
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery("SELECT * FROM supplier WHERE LOWER(company) = '" + companyName + "' AND LOWER(email) = '" + emailAddress + "'");

        if (rst.next())
        {
            int count = rst.getInt(i: 1);
            companyExists = count > 0; // If a Company with the same name and email found
        }
        rst.close();
        st.close();
        DBconnection.createDBconnection().close();
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while checking for duplicate company: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }

    return companyExists;
}
```

Figure 2.70: Check for Duplicate Entries Functionality Code

This method determines whether a supplier company with a specific name and email address already exists in the supplier database table. It creates a connection to the database and executes a SQL query to search for records that have the provided companyName and emailAddress in the company and email fields (both lowercased for case-insensitive comparison).

The "ResultSet" containing rows (rst. next () returns true) indicates the presence of a matching record, and this sets companyExists to "true". The query terminates by closing the database connection, the Statement, and the "ResultSet".

This procedure uses "JOptionPane" to display an error message in the event that a "SQLException" occurs. Finally, a boolean value indicating if the company has already registered is returned by the method.

```
private String getMedicineNameByMedID(int medicineID)
{
    String medicineName = null;
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery("SELECT med_name FROM medicine WHERE med_ID = " + medicineID);
        if (rst.next())
        {
            medicineName = rst.getString(string: "med_name");
        }
        rst.close();
        st.close();
        DBconnection.createDBconnection().close();
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while fetching the medicine name: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
    return medicineName;
}
```

Figure 2.71: Auto Generate the Specific Medicine name of Supplied Medicine ID Functionality Code

Using a provided medicine ID, the "getMedicineNameByMedID ()" function obtains a medicine's name from the database. It proceeds to establish a database connection and execute a SQL query to retrieve the "med_name" from the "medicine table" where the "med_ID" matches the provided "medicineID" after initializing the medicineName variable to null. It retrieves the "med_name" out of the result set and assigns it to "medicineName" if a matching entry is discovered. The procedure ensures that, at completion of use, the database connection, statement, and result set are closed. The user is presented with an error message the moment that a "SQLException" occurs. This method handles possible SQL exceptions with a try-catch block and, if necessary, displays an error dialog. It uses JDBC for database communication.

```
// Refresh the supplierTable after updating/deleting the selected supplier details
private void refreshSuppliersTable()
{
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery(string: " SELECT * FROM supplier ");

        DefaultTableModel model = new DefaultTableModel();
        model.addColumn(columnName:"Supplier ID");
        model.addColumn(columnName:"Company Name");
        model.addColumn(columnName:"Medicine ID");
        model.addColumn(columnName:"Medicine Name");
        model.addColumn(columnName:"Email");
        model.addColumn(columnName:"Contact No");
        model.addColumn(columnName:"Contact Person");
        model.addColumn(columnName:"Date");

        while (rs.next())
        {
            Object[] row =
            {
                rs.getInt(string: "sup_ID"),
                rs.getString(string: "company"),
                rs.getInt(string: "med_ID"),
                rs.getString(string: "medName"),
                rs.getString(string: "email"),
                rs.getInt(string: "phone_no"),
                rs.getString(string: "contact_person"),
                rs.getString(string: "date")
            };
            model.addRow(rowData: row);
        }

        suppliersTable.setModel(dataModel: model);
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while refreshing Table: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.72: Refresh the Supplier Table after Adding a new entry Functionality Code

The suppliersTable is updated with the most recent supplier data from the database using the above "refreshSuppliersTable ()" method. Using a Statement object, a database connection is first established. After that, a SQL query will be executed to get all the records from the "supplier table". With columns selected for different supplier attributes such Supplier ID, Company Name, Medicine ID, Medicine Name, Email, Contact No, Contact Person, and Date, a new DefaultTableModel is constructed to store the data. The data is extracted and added to the table model as the "ResultSet" iterates over each row of the query result. The "suppliersTable" is the final destination for the updated model. The user is presented with an error message if any SQL issues arise throughout this operation.

```
private void contactNoKeyPressed(java.awt.event.KeyEvent evt) {
    // Ensures that the user enters only numerical values for the contact number
    char c = evt.getKeyChar();

    // Validating the user inputs for the contact number
    if (!Character.isDigit(ch:c) || contactNo.getText().length() >= 10)
    {
        evt.consume();
        // Consume the event to prevent the input
        JOptionPane.showMessageDialog(parentComponent: null, message: "Enter up to 10 numerical values for the contact number", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
private void contactPersonKeyPressed(java.awt.event.KeyEvent evt) {
    // Ensures that the user enters only alphabetic values for the contact person
    char c = evt.getKeyChar();

    // Validating the user inputs for the contact person
    if (!Character.isLetter(ch:c))
    {
        evt.consume(); // Consume the event to prevent the input

        // Display error message
        JOptionPane.showMessageDialog(parentComponent: null, message: "Please Enter Only Alphabetic Characters for the Contact Person", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.73: Validate User Inputs for Contact No & Contact Person Functionality Code

The above two methods are intended for input validation. In the "contactNoKeyPressed ()" method, the input length is limited to a maximum of 10 digits and only numeric values can be entered by the user in the contact number field. The key press event is tracked, and it determines whether the character entered is a digit or if the input is longer than ten characters. If the input is incorrect, it displays an error message and consumes the event to prevent any further input. The user input for a contact person's name field is limited to alphabetic letters only using the second method, "contactPersonKeyPressed ()". Every key press is verified, and if a character is entered that isn't a letter, the input terminates, and an error message appears. "JOptionPane" is used in both approaches to show error warnings.

❖ Updating an Existing Supplier's Details Functionality:

- **Selecting one of the rows from the Supplier Details Table**

```
private void suppliersTableMouseClicked(java.awt.event.MouseEvent evt) {
    //Get the index of the selected row
    int selectedRow = suppliersTable.getSelectedRow();

    //Establish the table model that represents the structure & data of the table
    TableModel model = suppliersTable.getModel();

    //Validationa to ensure that User interacts with the selected row data
    //Set the Company Name,Medicine ID,Medicine Name,Email,Contact No,Contact Person & Registered Date text fields to the value from the selected row
    company.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 1).toString());
    med_id.setSelectedItem(anObject: model.getValueAt(rowIndex: selectedRow, columnIndex: 2).toString());
    med_Name.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 3).toString());
    email.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 4).toString());
    contactNo.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 5).toString());
    contactPerson.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 6).toString());

    //Retrieve the data from the First Column(index 0)of the selected row & store it in supplierID
    id.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 0).toString());

    //Set the Date Format
    SimpleDateFormat sfd = new SimpleDateFormat(pattern: "yyyy-MM-dd");
    String added_Date = (String) model.getValueAt(rowIndex: selectedRow, columnIndex: 7);

    try
    {
      //Handle the parsing of the Registered Date
      Date register_Date = sfd.parse(source: added_Date);
      Date.setDate(date: register_Date);

      // Disable Registered Date when user tries to update an existing data tuple
      Date.setEnabled(bln: false);
    }
    catch (ParseException e)
    {
        // Handle the parsing exception,by displaying an error message.
        e.printStackTrace();
    }

}
```

Figure 2.74: Selecting a row from the Supplier Details Table Functionality Code

❖ **Explanation of selecting a row from the table:**

This “suppliersTableMouseClicked ()” method reacts to a mouse click event on a table (suppliersTable). Upon clicking a row within the database, the procedure receives the table's data model and returns the row's index. Subsequently, it retrieves information from designated columns of the chosen row and applies it to several text fields and a combo box. The firm name, medicine ID, medicine name, email, contact number, contact person, and registration date are among the fields that have been modified. After being parsed into a Date object and assigned to a date picker component, the registration date is hidden to avoid user change. An exception is caught, and the stack trace is reported in the event that a parsing issue arises during the date conversion process. This feature ensures that when a user selects a row in the table, the relevant data appears and is editable in the relevant fields, and that the registration date for already-existing entries remains the same.

```
private void clearBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // Clear Button
    id.setText(t: "Will Be Added Automatically");

    // Clear the selected item in the combo box if an item is selected
    if (med_id.getSelectedItem() != null)
    {
        med_id.setSelectedIndex(anIndex: -1);
    }

    company.setText(t: "");
    med_Name.setText(t: "");
    email.setText(t: "@gmail.com");
    contactNo.setText(t: "");
    contactPerson.setText(t: "");
    Date.setDate(new java.util.Date());
}
```

Figure 2.75: Clearing all the User Input Fields Functionality Code

This "clearBtnActionPerformed ()" method manages the "Clear" button's action event. The method sets the default values of different fields in the user interface when the button is clicked. It specifically clears the text in many text fields (company, med_Name, contactNo, and contactPerson) and sets the text of the id field to "Will Be Added Automatically." It also clears any selected item in the "med_id combo box" by setting its selected index to (-1). Additionally, it updates the date picker (Date) to the current date and resets the email field to "@gmail.com". By ensuring that every input field is empty and prepared for new data, this method gives the user a reliable place to start.

❖ Updating an existing Supplier's Details

```
private void updateBtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Update Button
    try
    {
        String companyName, medicineName, emailAddress, contact_Person, contactNumber;
        String medicineID; // Changing Int to String for parsing
        int supplierID;

        companyName = company.getText();
        medicineName = med_Name.getText();
        emailAddress = email.getText();
        contact_Person = contactPerson.getText();
        medicineID = (String) med_id.getSelectedItem();
        contactNumber = contactNo.getText();
        supplierID = Integer.parseInt(s: id.getText());

        // Validating empty fields
        if (companyName.isEmpty() || emailAddress.isEmpty() || contactNumber.isEmpty() || contact_Person.isEmpty())
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Please fill in all the required fields", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Validating contact number
        if (contactNumber.length() != 10)
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Enter exactly 10 numerical values for the contact number", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Validate email address
        if (!emailAddress.endsWith(suffix: "@gmail.com") || emailAddress.equals(anObject: "@gmail.com"))
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Enter a valid Gmail address", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Check for duplicate data
        if (RowisDuplicated(companyName, emailAddress, supplierID))
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Error: Please check the company details you're trying to update; There's another company in the system database with the same details", title:"Error",
            return;
        }
            String query = "UPDATE supplier SET med_ID = ?, medName = ?, company = ?, "
                         + "contact_person = ?, email = ?, phone_no = ?, date = ? "
                         + "WHERE sup_ID = ?";

            // Establish database connection and prepare the statement
            Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
            // Use PreparedStatement to avoid SQL injection
            java.sql.PreparedStatement prepSt = st.getConnection().prepareStatement(string: query);

            prepSt.setInt(i: 1, i1:Integer.parseInt(s: medicineID));
            prepSt.setString(i: 2, string: medicineName);
            prepSt.setString(i: 3, string: companyName);
            prepSt.setString(i: 4, string: contact_Person);
            prepSt.setString(i: 5, string: emailAddress);
            prepSt.setString(i: 6, string: contactNumber);
            prepSt.setDate(i: 7, date: java.sql.Date.valueOf(s: dateAdded));
            prepSt.setInt(i: 8, i1:supplierID);

            // Execute the update statement
            int rowCount = prepSt.executeUpdate();

            // Check if update was successful
            if (rowCount > 0)
            {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Supplier's Details Updated Successfully", title:"Success", messageType: JOptionPane.INFORMATION_MESSAGE);
                refreshSuppliersTable(); // Refresh the table after successful update
            }
            else
            {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Failed to update Supplier's Details", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
            }

    }
    catch (NumberFormatException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Invalid Input for Supplier ID", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
catch (SQLException ex)
{
    JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while updating Supplier's Details: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
}
}

private boolean RowisDuplicated(String companyName, String emailAddress, int supplierID)
{
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        String query = "SELECT * FROM supplier WHERE company = '" + companyName + "' AND email = '" + emailAddress + "' AND sup_ID != " + supplierID;
        ResultSet rs = st.executeQuery(string: query);
        return rs.next(); // If there's a result, it means duplicate data exists
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while checking for duplicate data: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        return false;
    }
}
```

Figure 2.76: Updating a row from the Supplier Details Table Functionality Code

❖ Explanation of updating details of an existing row from the supplier table:

This “UpdateBtnActionPerformed ()” action event handler aims to update supplier data in a database. Following clicking the "Update" button, input fields including supplier ID, company name, medicine name, email, contact person, and contact number are retrieved and verified. The code verifies that the email address is a working Gmail account, the contact number is exactly ten digits, and none of these fields are empty. Additionally, it uses the "RowisDuplicated ()" function to search the database for duplicate records (using the Supplier Company Name and its official Email Address). In order to prevent SQL injection, it formats the date after validation, creates a SQL update query, and runs it using a prepared statement. The supplier table is refreshed, and a success message is displayed if the update is successful; otherwise, an error notice displays. SQL exceptions and incorrect input formats are handled by error handling.

❖ Deleting an existing row from the Supplier’s Details Table

```
private void deleteBtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Delete Button
    int confirm = JOptionPane.showConfirmDialog(parentComponent: null, message: "Are you sure you want to 'Delete' this Supplier?", title:"Confirm Deletion", optionType:JOptionPane.YES_NO_OPTION);
    if (confirm == JOptionPane.YES_OPTION)
    {
        try
        {
            Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
            String query = "DELETE FROM `supplier` WHERE `sup_ID` = " + Integer.parseInt(s: suppliersTable.getValueAt(row: suppliersTable.getSelectedRow(), column: 0).toString());

            //executeUpdate Method call to execute the DELETE query & returns the No of rows deleted
            int count = st.executeUpdate(string: query);
            //System.out.println(query);

            //Data Validations
            if (count > 0)
            {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Selected Supplier's Details deleted successfully.", title:"Success", messageType: JOptionPane.INFORMATION_MESSAGE);

                // Refresh the suppliersTable after deleting
                refreshSuppliersTable();
            }
            //Deselect previous selected rows
            suppliersTable.clearSelection();
            refreshSuppliersTable();
        }
            catch (Exception e)
            {

            }
        }
}
```

Figure 2.77: Deleting a row from the Supplier Details Table Functionality Code

When the delete button is clicked, this “deleteBtnActionPerformed ()" method is called. The user is first presented with a confirmation dialog asking if they are certain they wish to remove the chosen supplier. The supplier is then removed from the database if the user provides confirmation. Using the database connection, the method creates a SQL DELETE query, extracts the supplier ID of the chosen record from the suppliersTable, and runs “createDBconnection ()” for “DBconnection”. The user receives a success message, and the suppliers table is refreshed to show the modifications if the deletion is successful. At last, the selected row in the table is deleted. The catch block implements error handling but doesn't go into depth about it.

❖ Searching for a Registered Supplier from the Supplier's Details Table

```
private void searchBtnActionPerformed(java.awt.event.ActionEvent evt) {
    String searchText = searchTxt.getText().trim();

    // Check if the search text is not empty
    if (!searchText.isEmpty())
    {
        try
        {
            Statement st = DBconnection.createDBconnection().createStatement();
            ResultSet rs = st.executeQuery("SELECT * FROM supplier WHERE (sup_ID LIKE '%" + searchText + "%') || (company LIKE '%" + searchText + "%') || (medName LIKE '%" + searchText + "%')");

            // Create a table model to hold the search results
            DefaultTableModel model = new DefaultTableModel();
            model.addColumn(columnName:"SupplierID");
            model.addColumn(columnName:"Company Name");
            model.addColumn(columnName:"Medicine ID");
            model.addColumn(columnName:"Medicine Name");
            model.addColumn(columnName:"Email");
            model.addColumn(columnName:"Contact No");
            model.addColumn(columnName:"Contact Person");
            model.addColumn(columnName:"Date");

             // Add the search results to the table model
            while (rs.next())
            {
                Object[] row =
                {
                    rs.getInt(string: "sup_ID"),
                    rs.getString(string: "company"),
                    rs.getInt(string: "med_ID"),
                    rs.getString(string: "medName"),
                    rs.getString(string: "email"),
                    rs.getString(string: "phone_no"),
                    rs.getString(string: "contact_person"),
                    rs.getString(string: "date")
                };
                model.addRow(rowData: row);
            }

            // Set the table model to the JTable
            suppliersTable.setModel(dataModel: model);

        }
        catch (SQLException e)
        {
            JOptionPane.showMessageDialog(parentComponent: null, "Error Occurred: " + e.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
            e.printStackTrace();
        }
    }
    else
    {
        // If search text is empty, refresh the table with all suppliers
        refreshSuppliersTable();
    }
}
```

Figure 2.78: Search for a Registered Supplier Details from the Table Functionality Code

The "searchBtnActionPerformed ()" the event handler for a Java Swing application's search button is defined by this method. After clicking the button, the text is extracted from the search text field (searchTxt) and any previous or following whitespace is removed. It uses "DBconnection" to create a database connection if the search text is not empty. The method "createDBconnection ()" generates a Statement object and runs a SQL query to find suppliers in the database that have the search phrase contained in their sup_ID, company, or medName fields. After processing and storing the data, a "DefaultTableModel" is utilized to update the "JTable" (suppliersTable). In the instance that the search bar is empty, all supplier records in the supplier table are displayed by calling the "refreshSuppliersTable ()" method. Any SQL exceptions that occur are detected and shown in the output.

Major module structures, reusable codes & Development tools used in Supplier Management Page's Functionality

Major Module Structures:

- User Interface is designed using Java Swing components like JFrame, JPanel, JTable, JButton, etc
- Used methods for interacting with a database using JDBC (java.sql package).
- Added Action listeners for handling user interactions and events (button clicks (ActionListener) and focus events (FocusAdapter)).
- Data Validation Methods for validating and verifying user input, such as contact numbers and email addresses.
- Used Data Population Methods to populate User Interface components with data retrieved from the database.
- Included Data Manipulation Methods for adding, updating, and deleting supplier details from the system database.
- Table Refreshing Method to refresh the supplier details table after data modification was added.
- Searching Ability to search for suppliers based on various criteria like supplier ID, Supplier Company Name, the Medicine name that is being supplied by the specific supplier.

Reusable Codes:

- "DBconnection.createDBconnection()" is being repeatedly called to establish the database connection.
- "getMedicineIDs ()" method is used to populate the medicine IDs in a combo box.
- getMedicineNameByMedID () fetches the medicine name by its medicine ID.
- "validateContactNumber ()", "isCompanyAlreadyRegistered ()", and "RowisDuplicated ()" methods validate contact number and check for duplicate entries.
- "refreshSuppliersTable ()" method is used to refreshe the supplier details table repeatedly.
- Used reusable event listeners for the components like "contactPerson" and "med_id".

Development tools

- Java Swing
- JDBC: Java Database Connectivity
- MySQL Database
- NetBeans or Eclipse IDE 19
- Simple Date Format
- JOptionPane

3) Customer Management Page (IT22884138 – RATHNAYAKA R.M.T.D)

The Customer Details module of the Pharmacy Management System is made to effectively manage and preserve customer data. The adding, updating, removing, and viewing of client records are all handled by this module. Because of its graphical user interface (GUI) and database interfaces, it guarantees reliable data administration and user-friendly procedures.

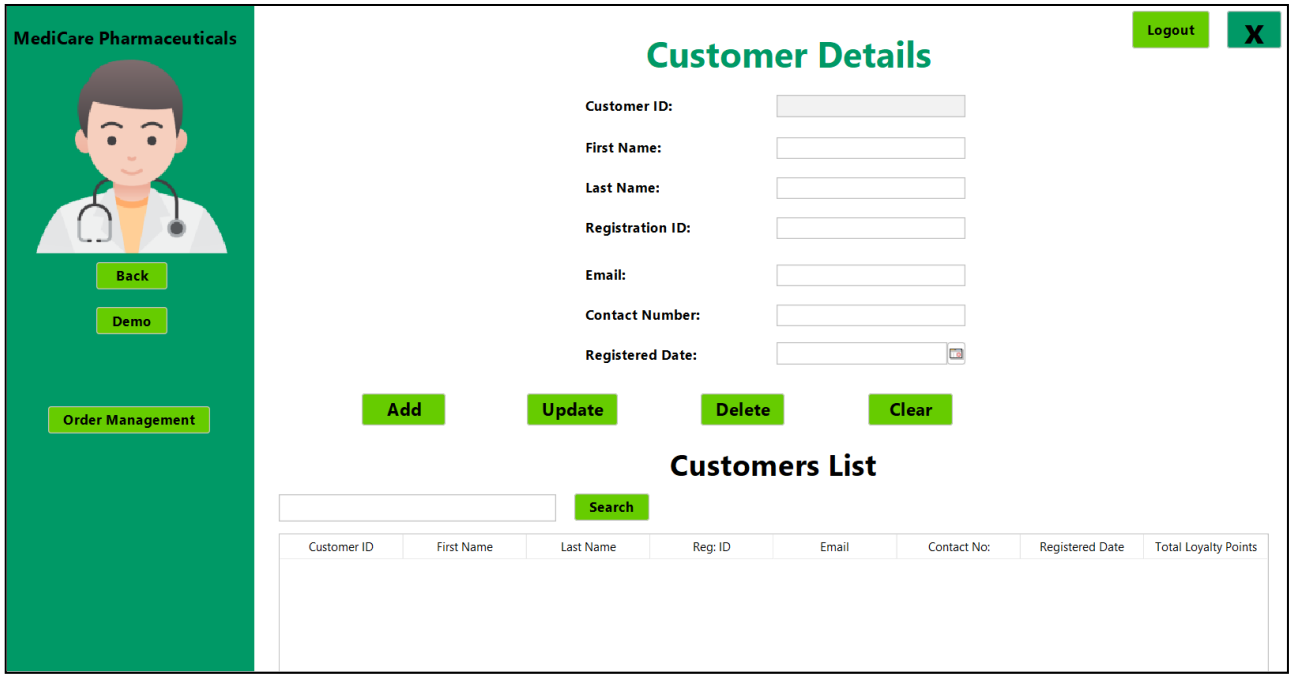

Figure 2.79: Customer Management Page UI

Major Module Structures in CustomerDetails Class

- Initialization and UI Setup: The constructor CustomerDetails () initializes the UI components and sets default values for the form field; it is used in the constructor and can be reused whenever you need to reset the form fields

```
public class CustomerDetails extends javax.swing.JFrame {
// Creates new form CustomerDetails

    public CustomerDetails()
    {
        initComponents();
        refreshCustomerTable();

        cust_id.setText(t: "Will Be Added Automatically");
        fname.setText(t: "");
        lname.setText(t: "");
        regID.setText(t: "");
        email.setText(t: "@gmail.com");
        contactNo.setText(t: "");
        date.setDate(date:Date.from(instant: Instant.now()));
        points.setText(t: "1");
    }
```

Figure 2.80: Initialization of CustomerDetails Class

The CustomerDetails class constructor initialises the GUI elements by initComponents (), ensuring the frame's layout matches the design. It also updates the customer table with the latest data from the database using refreshCustomerTable (), ensuring accuracy. Default values are set for various fields, preparing them for user input; for example, there is a placeholder for "automatic generation" in the customer ID column. The fields for the date and loyalty points have default values. This thorough initialization guarantees that the frame is ready for user interaction right out of the box when it is instantiated.

❖ Add New Customer

```
private void AddBtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Add Button
    String firstName, lastName, emailAddress, registeredDate;
    int registrationID, contactNumber, loyaltyPoints;

    firstName = fname.getText();
    lastName = lname.getText();
    emailAddress = email.getText();

    if (firstName.isEmpty() || lastName.isEmpty())
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Name cannot be empty !", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);

    }
    else if (emailAddress.length() < 11)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Enter valid email", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);

    }
    else
    {

        try
        {
            registrationID = Integer.parseInt(s: regID.getText());
            contactNumber = Integer.parseInt(s: contactNo.getText());
            loyaltyPoints = Integer.parseInt(s: points.getText());

            if (regID.getText().length() != 6)
            {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Enter valid registeration number", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);

            }
    else if(contactNo.getText().length() != 10)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Enter valid phone  number", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
    else
    {
        //Validating the date formatting
        SimpleDateFormat sfd = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        registeredDate = sfd.format(date:date.getDate());

        //Validations of Database Interaction
        try
        {
            Statement st = pharmacy.DBconnection.createDBconnection().createStatement();

            //Fetch the next avaialable Customer ID from the Database
            ResultSet rs = st.executeQuery(sql: "SELECT MAX(cust_ID) FROM doctor");
            int nextCustID = 1;

            if (rs.next())
            {
                nextCustID = rs.getInt(columnIndex: 1) + 1;
            }

            int count = st.executeUpdate("INSERT INTO doctor(cust_ID, f_name,l_name,doc_regID, email,phone_no,reg_date,loyalty_points) VALUES "
                    + ""
                    + "('" + nextCustID + "','" + firstName + "', '" + lastName + "', '" + registrationID + "', '" + emailAddress + "','" +
                    contactNumber + "', '" + registeredDate + "', '" + loyaltyPoints + "')");

            if (count > 0)
            {
                JOptionPane.showMessageDialog(parentComponent: null, message: "New Customer's Details Added Successfully", title: "Success",
                        messageType: JOptionPane.INFORMATION_MESSAGE);
                        // Refresh the customerTable after adding the new customer
                        refreshCustomerTable();
                    }
                } //Validations of Error Handling
                catch (NumberFormatException e)
                {
                    JOptionPane.showMessageDialog(parentComponent: null, "Invalid Input for Registration ID / Contact Number / Quantity /"
                            + "Price Or Total Amount...!", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
                }
                catch (Exception e)
                {
                    JOptionPane.showMessageDialog(parentComponent: null, message: "Something went wrong...!", title: "Error", messageType: JOptionPane.
                            ERROR_MESSAGE);
                }
            }
        }
        catch (NumberFormatException ex)
        {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Invalid input!", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
                return;
        }
    }

    //User input validation
}
```

Figure 2.81: Register a New Customer

Clicking the "Add" button adds a new customer, which is handled by this code segment. After retrieving user-entered customer information, it verifies its accuracy by looking for blank fields and appropriately spaced input values. After validation is successful, it formats the date, parses inputs into the relevant data types, and creates a database connection. After obtaining the next available customer ID, an INSERT SQL query is run to add the new client. After a successful addition, a success message appears, and the customer database is updated. A reliable method for adding customers while preserving data accuracy is ensured by the corresponding error messages that are displayed in the event of errors, such as invalid input or problems with the database connection.

- **Update Customer Details**

This code segment handles the action when the "Update" button is pressed. It retrieves, validates, and formats consumer information from input fields. Next, it establishes a database connection, constructs a SQL UPDATE query to modify the customer's data, and executes it using prepared statements. After a successful update, it displays a success message and changes the customer table. The user sees the appropriate error messages when something goes wrong. When everything is considered, it ensures that client data is updated in the database without interruption and that data integrity is maintained.

```
private void UpdateBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // Update Button
    try
    {
        String firstName, lastName, emailAddress, registeredDate;
        int registrationID, contactNumber;

        firstName = fname.getText();
        lastName = lname.getText();
        emailAddress = email.getText();

        //Validating the date formatting
        SimpleDateFormat sfd = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        registeredDate = sfd.format(date:date.getDate());

        //User input validation
        registrationID = Integer.parseInt(s: regID.getText());
        contactNumber = Integer.parseInt(s: contactNo.getText());

        //Validations of Database Interaction
        try
        {
            Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
            int x = Integer.parseInt(s: cust_id.getText());

            //Validations related to SQL
            String query = "UPDATE doctor SET f_name = ? , l_name = ? , doc_regID = ? , email = ? , phone_no = ? ,reg_date = ? WHERE cust_ID = ?";

            // Use PreparedStatement to avoid SQL injection
            java.sql.PreparedStatement prepSt = st.getConnection().prepareStatement(sql: query);

            prepSt.setString(parameterIndex: 1, x: firstName);
            prepSt.setString(parameterIndex: 2, x: lastName);
            prepSt.setInt(parameterIndex: 3, x: registrationID);
            prepSt.setString(parameterIndex: 4, x: emailAddress);
            prepSt.setInt(parameterIndex: 5, x: contactNumber);

            // Assuming pickupDate is a LocalDate
            prepSt.setString(parameterIndex: 6, x: registeredDate);
            prepSt.setInt(parameterIndex: 7, x);

            int count = prepSt.executeUpdate();

            if (count > 0)
            {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Selected Customer Details Updated Successfully", title: "Success",
                        messageType: JOptionPane.INFORMATION_MESSAGE);

                // Refresh the Customer Table after updating
                refreshCustomerTable();
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
            JOptionPane.showMessageDialog(parentComponent: null, message: "Error occurred while Updating the selected customer's details", title: "Error",
                    messageType: JOptionPane.ERROR_MESSAGE);
        }


        h (NumberFormatException e)

        JOptionPane.showMessageDialog(parentComponent: null, message: "Invalid Input for Registration ID / Contact Number / Quantity /Price Or Total Amount...!",
                title: "Error", messageType: JOptionPane.ERROR_MESSAGE);

        h (Exception e)
        `
          e.printStackTrace();
          JOptionPane.showMessageDialog(parentComponent: null, message: "Error occurred while Updating the selected customer's details",
                  title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
        }

    }
    catch (NumberFormatException e)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Invalid Input for Registration ID / Contact Number / Quantity /Price Or"
                + " Total Amount...!", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
    catch (Exception e)
    {
        //Generic Exception for any kind of exception
    }
```

Figure 2.82: Update Customer Details

❖ Delete Customer Details

```
private void DeleteBtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Delete Button
    try
    {
        Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
        String query = " DELETE FROM doctor WHERE cust_ID = " + Integer.parseInt(s: customerTable.getValueAt(row: customerTable.
                getSelectedRow(), column: 0).toString());

        //executeUpdate Method call to execute the DELETE query & returns the No of rows deleted
        int count = st.executeUpdate(sql: query);
        //System.out.println(query);

        //Data Validations
        if (count > 0)
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Selected Customer's Details deleted successfully.",
                    title: "Success", messageType: JOptionPane.INFORMATION_MESSAGE);

            // Refresh the customerTable after deleting
            refreshCustomerTable();
        }

        //Deselect previous selected rows
        customerTable.clearSelection();
        refreshCustomerTable();
    }
    catch (Exception e)
    {

    }
}
```

Figure 2.83: Delete Customer Details

The DeleteBtnActionPerformed method is an event handler for a button click. It uses a method call pharmacy to establish a database connection inside a try-catch block. DBconnection.createDBconnection(). createStatement () and creates a Statement object (st) for executing SQL queries. It constructs a SQL DELETE query to remove a customer record from the doctor table based on the customer's ID (cust_ID). The customer ID is retrieved from the selected row of a table (customerTable) using customerTable.getValueAt(customerTable.getSelectedRow(), 0). toString (), and it's parsed to an integer. The executeUpdate method of the Statement object is invoked with the DELETE query as an argument. This method executes the SQL query and returns the number of rows affected by the operation. If the count of affected rows is greater than 0, meaning the deletion was successful, a success message dialog is displayed using JOptionPane.showMessageDialog. The refreshCustomerTable () method is called to update the customer table display after deletion. Previous selected rows are cleared from the customerTable using customerTable.clearSelection(). Finally, the refreshCustomerTable () method is called again for additional table refreshing, and any exceptions encountered during the process are caught and handled in the catch block.

❖ Clear Customer Details

```
private void ClearBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // Clear Button
    cust_id.setText(t: "Will Be Added Automatically");
    fname.setText(t: "");
    lname.setText(t: "");
    regID.setText(t: "");
    email.setText(t: "@gmail.com");
    contactNo.setText(t: "");
    date.setDate(date:Date.from(instant: Instant.now()));
    points.setText(t: "1");
}
```

Figure 2.84: Clear Customer Details

In general, this code segment makes sure that when the "Clear" button is pressed, a number of customer information-related input fields are reset or cleared to make room for new customer details to be entered.

❖ Input Validation

Checks, if the first name or last name fields are empty. Display the error message

```
if (firstName.isEmpty() || lastName.isEmpty())
{
    JOptionPane.showMessageDialog(parentComponent: null, message: "Name cannot be empty !", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
```

Figure 2.85: Customer Input Validation

- Email validation: Verifies the validity of the email address. Checks to See whether the email address is fewer than 11 characters in length. whether it is, it displays an error message.

```
if (regID.getText().length() != 6)
{
    JOptionPane.showMessageDialog(parentComponent: null, message: "Enter valid registeration number", title: "Error",
            messageType: JOptionPane.ERROR_MESSAGE);

}
```

Figure 2.86: Customer Email Validation

- Registration ID length Validation: Verifies whether the registration's length is six digits or less. If the length is not correct, show an error message.

```
if (regID.getText().length() != 6)
{
    JOptionPane.showMessageDialog(parentComponent: null, message: "Enter valid registeration number", title: "Error",
            messageType: JOptionPane.ERROR_MESSAGE);

}
```

Figure 2.87: Customer Registration ID Validation

- Contact number length validation: Verifies whether the contact number's length is less than or equal to ten digits. If there is a mistake with the length, show an error message.

```
// Ensures that the user enters only numerical values for the contact number
char c = evt.getKeyChar();

// Validating the user inputs for the contact number
if (!Character.isDigit(ch: c))
{
    evt.consume();
    // Consume the event to prevent the input
    JOptionPane.showMessageDialog(parentComponent: null, message: "Enter up to 10 numerical values for the contact number", title: "Error",
            messageType: JOptionPane.ERROR_MESSAGE);
}
```

Figure 2.88: Customer Contact number length Validation

- Character Input validation for first name and last name: Make sure the first and last name sections are filled in with only alphabetical characters. If any numeric characters are inputted, an error notice is displayed.

```
char c = evt.getKeyChar();

if (Character.isDigit(ch: c))
{
    fname.setEditable(b: false);
    JOptionPane.showMessageDialog(parentComponent: null, message: "Enter letters only", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);

}
else
{
    fname.setEditable(b: true);
}
private void lnameKeyPressed(java.awt.event.KeyEvent evt) {
    char c = evt.getKeyChar();

    if (Character.isDigit(ch: c))
    {
        lname.setEditable(b: false);
        JOptionPane.showMessageDialog(parentComponent: null, message: "Enter letters only", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);

    }
    else
    {
        lname.setEditable(b: true);
    }
}
```

Figure 2.89: Customer first name and last name Character Input Validation

Reused code segments and Development tools

- Database Connection Management-The method to create a database connection

```
// Establish database connection and prepare the statement
Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
java.sql.PreparedStatement pst = st.getConnection().prepareStatement(sql);
```

Figure 2.90: create a database connection

The ability to utilise the same code in several application components without requiring changes is known as reusability. Reusability is increased by the encapsulated approach for various reasons:You can make sure that any modifications to the database connection logic only need to be made once in this function by having a single method for creating Statement objects.,Reduced Code Duplication Instead of writing Statement st = pharmacy.DBconnection.createDBconnection().createStatement(); multiple times throughout your codebase, st.getConnection(): This method is likely called on an instance of a statementobject (st). It retrieves the Connection object associated with the statement. The getConnection () method is typically provided by a JDBC driver and establishes a connection to a database. prepareStatement(sql): Once the Connection object is retrieved, the prepareStatement () method is invoked on it, passing a SQL query string (sql) as an argument. This method prepares the SQL statement for execution in the database. It returns a PreparedStatement object (pst), which represents a precompiled SQL statement that can be executed multiple times with different parameters.

- Table refreshing method to refresh the customer table

```
//Refresh the customerTable after updating/deleting the selected customer details
private void refreshCustomerTable()
{
    try
    {
        // Fetch updated data from the Database
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery(sql: " SELECT * FROM doctor ");

        // Create a new DefaultTableModel with updated data
        DefaultTableModel model = new DefaultTableModel();
        model.addColumn(columnName: "Customer ID");
        model.addColumn(columnName: "First Name");
        model.addColumn(columnName: "Last Name");
        model.addColumn(columnName: "Registration ID");
        model.addColumn(columnName: "Email");
        model.addColumn(columnName: "Contact No");
        model.addColumn(columnName: "Registered Date");
        model.addColumn(columnName: "Total Loyalty Points");

        while (rs.next())
        {
            Object[] row
                    = {
                        rs.getInt(columnLabel: "cust_ID"),
                        rs.getString(columnLabel: "f_name"),
                        rs.getString(columnLabel: "l_name"),
                        rs.getInt(columnLabel: "doc_regID"),
                        rs.getString(columnLabel: "email"),
                        rs.getInt(columnLabel: "phone_no"),
                        rs.getString(columnLabel: "reg_date"),
                        rs.getString(columnLabel: "loyalty_points")
                    };
            model.addRow(rowData: row);
            ,,
            model.addRow(rowData: row);
        }
        // Set the updated model to the customerTable
        customerTable.setModel(dataModel:model);
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while refreshing Table:"
                + " " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.91: Refresh Customer Table

The refreshCustomerTable method updates the data displayed in a JTable based on the latest information retrieved from the database. It utilizes a reusable method getDBConnection to establish a database connection and create a statement, ensuring code reusability and simplifying maintenance. The method executes a SQL query to fetch all rows from a specific table and structures the retrieved data into a new DefaultTableModel with corresponding columns. By iterating over the ResultSet, each row's data is extracted and added to the table model. Finally, the JTable's model is updated with the new data, reflecting the current database state. Error handling is included to handle SQL exceptions and display

user-friendly error messages. Overall, the method demonstrates effective database

```
        try
        {
            //Handle the parsing of the Pickup Date
            Date pickup_Date = sfd.parse(source: pickupDate);
            date.setDate(date:pickup_Date);
        }
        catch (ParseException e)
        {
            // Handle the parsing exception,by displaying an error message.
            e.printStackTrace();

        }
    }
    catch (Exception e)
    {
        System.out.println(x: e.getMessage());
    }
}
```

interaction and enhances application reliability.

- Exception handling
-

Figure 2.92: Customer Exception handling

//Exception handling if any unexpected error occurs during the process

```
Catch (Exception e)
{
    e. printStackTrace ();
}
```

This code segment catches any exceptions that may occur during the process of retrieving data from the table and parsing dates. By printing the stack trace (e. printStackTrace ()), It gives details about the exception, which is helpful for troubleshooting and figuring out where the mistake occurred. The application's exception handling method can be applied again to handle unforeseen failures in different areas of the codebase.

Development tools

Together, these technologies and development tools produce a Java application that is both useful and easy to use for handling customer information in a drugstore. In order to streamline the software development process and guarantee the product's dependability and quality, they offer the frameworks, libraries, and utilities that are required.

1)Java Swing:
The application's graphical user interface (GUI) is made with Java Swing. It offers a collection of UI elements, including text fields, buttons, tables, and other elements, that are utilized in the design of the user interface.

2)JDBC (Java Database Connectivity):
Java code can communicate with the database using JDBC. It offers a collection of classes and interfaces for establishing a database connection, running SQL queries, and getting the output. JDBC is probably utilized in this application to execute database activities including adding, removing, and querying customer information.

3)Java Standard Library:
The Java Standard Library offers features that are necessary for programming in Java. Date, SimpleDateFormat, Instant, and other standard library classes are utilised in this application to handle date and time related tasks.

4)Java IDE (Integrated Development Environment):
It's likely that an IDE like Eclipse or NetBeans is used to develop Java applications. Code editing, debugging, compilation, and deployment are just a few of the functions that these IDEs offer to improve development efficiency.

5)SQL Database (ex. MySQL):

To store client information, a relational database management system (RDBMS) is employed. This program may use MySQL or a related database system. Tables for keeping track of client data, including email addresses, phone numbers, and first and last names, are probably included in the database schema.

6)Version Control System (exGit):

The application's source code may be managed using version control systems such as Git. It enables developers to work together, monitor changes, and efficiently manage various codebase versions.

7)UI Design Tools:

Although not specifically stated in the code, the user interface can be designed and laid out using UI design tools like Adobe XD, Sketch, or Figma, and then Java Swing can be used to create it.

3) Order Management Page (IT22884138 – RATHNAYAKA R.M.T.D)

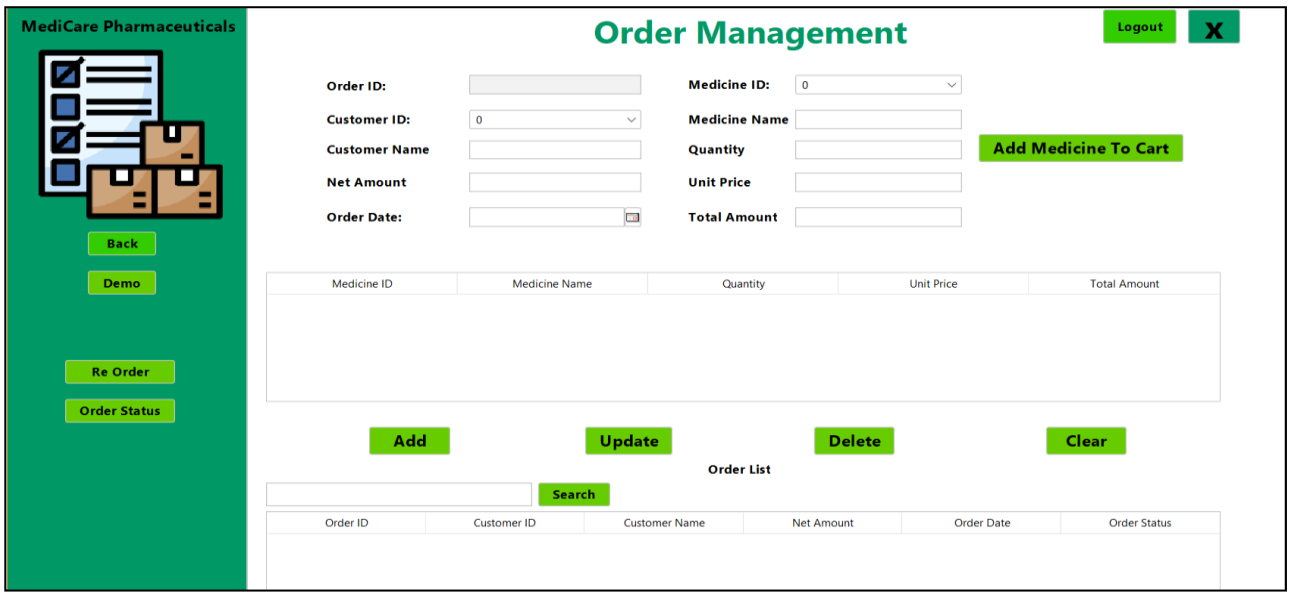

Figure 2.93: Customer Data Sorting Algorithm

Order management within a pharmaceutical system is made easier with the help of the Order Details page. Users can generally add new orders, edit existing orders, delete orders, and view/order drugs on this page, among other order-related tasks. characteristics like customer selection, medication selection, quantity specification, total computation, and order detail recording are typical characteristics. It might also have features like managing client information, creating reports, and checking order history.

Major Module Structures in OrderDetails Class

- Initialization and UI Setup: OrderDetails () is a constructor that may be reused whenever you need to reset the form fields. It initialises the UI components and sets default values for the form field.you need to reset the form fields.

```
public class OrderDetails extends javax.swing.JFrame {
// Creates new form OrderDetails

    public OrderDetails() {
        initComponents();
        getMedicineID();
        getCustomerID();
        getOrderID();
        getOrderDetails();
        date.setDate(date:Date.from(instant: Instant.now()));
    }
```

Figure 2.94: OrderDetails Class Initialization

- getMedicineID () - fills the drug ID dropdown with medicine IDs that are retrieved from the database. This technique appears to retrieve information from the database about medicines. getCustomerID (): Retrieves customer IDs from the database and populates the customer ID dropdown. Similar to getMedicineID (), this method fetches data related to customers. getOrderID (): Retrieves the maximum Order ID from the database and sets the next available Order ID. This method is responsible for getting the order ID, likely for the purpose of creating new orders. getOrderDetails (): Retrieves order details from the database and populates the ordersTable. This method fetches existing order details from the database and displays them in a table. date. setDate (Date.from(Instant.now ())): Sets the current date. This line sets the date field in the UI to the current date and time using the Instant.now () method. Overall, the constructor sets up the initial state of the OrderDetails frame by initializing UI components, fetching necessary data from the database, and setting the current date.

- Exit button action performed method- The Exit button's event handler is represented by the method in your code. This function is invoked when the Exit button is clicked.

```
private void ExitBtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Exit Button
    System.exit(status: 0);
}
```

Figure 2.95: OrderDetails Class Exit Button

- Load Medicine Cart

```
private void loadCartData() {
    ResultSet rst;

    try {
        String sql = "SELECT med_ID, med_name, qty, unit_price FROM medicine";

        // Establish database connection and prepare the statement
        Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
        java.sql.PreparedStatement pst = st.getConnection().prepareStatement(sql);

        rst = pst.executeQuery();

        DefaultTableModel model = (DefaultTableModel) Cart.getModel();
        model.setRowCount(rowCount:0); // clear the table

        while (rst.next()) {
            int medID = rst.getInt(columnLabel: "med_ID");
            String medName = rst.getString(columnLabel: "med_name");
            int qty = rst.getInt(columnLabel: "qty");
            int uPrice = rst.getInt(columnLabel: "unit_price");
            int tPrice = qty * uPrice;

            model.addRow(new Object[]{medID, medName, qty, uPrice, tPrice});
        }
    } catch (Exception e) {
        e.printStackTrace();
    }

    for (int row = 0; row < Cart.getRowCount(); row++) {
        int rowHeight = Cart.getRowHeight();

        for (int column = 0; column < Cart.getColumnCount(); column++) {
            Component comp = Cart.prepareRenderer(renderer:Cart.getCellRenderer(row, column), row, column);
            rowHeight = Math.max(a: rowHeight, comp.getPreferredSize().height + 12);

        }
        Cart.setRowHeight(row, rowHeight);
    }
}
```

Figure 2.96: Load Medicine Cart

The loadCartData method establishes a connection to the database, obtains information from the medication table, and adds this information to the Cart table. Additionally, it modifies the row heights to improve legibility. Using this technique guarantees that the Cart table always shows the most recent information available from the database.

- ResultSet Declaration: ResultSet rst-declares a ResultSet object that will hold the result of the SQL query.
- Try-Catch Block- The code inside the try block is used to connect to the database, execute the SQL query, and handle any exceptions that might occur during this process.

- SQL Query-String sql = "SELECT med_ID, med_name, qty, unit_price FROM medicine", This SQL query retrieves the med_ID, med_name, qty, and unit_price columns from the medicine table.
- Database Connection: Statement st = pharmacy. DBconnection.createDBconnection(). createStatement (); This line establishes a connection to the database and creates a Statement object to execute the SQL query.java.sql. PreparedStatement pst = st. getConnection (). prepareStatement(sql); This line prepares the SQL query for execution.
- Execute Query: rst = pst. executeQuery (); This line executes the SQL query and stores the result in the ResultSet object rst.
- Get Table Model: DefaultTableModel model = (DefaultTableModel) Cart.getModel(); This line gets the table model for the Cart table, which is used to manage the data displayed in the table.
- Clear Table: model. setRowCount (0); This line clears the existing data in the table before loading new data.
- Iterate Through ResultSet: The while loop iterates through the ResultSet and retrieves the data for each row. For each row, it retrieves the med_ID, med_name, qty, and unit_price values. It calculates the total price (tPrice) as qty * uPrice.
- Add Rows to Table Model: model. addRow (new Object [] {medID, medName, qty, uPrice, tPrice}); This line adds a new row to the table model with the retrieved data.
- Adjust Row Heights: The for loop iterates through each row in the Cart table. It adjusts the row height based on the preferred height of the cell components, ensuring better readability
- . Cart.setRowHeight(row, rowHeight); This line sets the adjusted row height for each row

➢ Get customer ID: fills the customer ID option with customer IDs that are retrieved from the database's doctor table.

```
// get customer ID
public void getCustomerID() {
    try {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery(sql: "SELECT cust_ID FROM doctor");

        while (rst.next()) {
            int custID = rst.getInt(columnLabel: "cust_ID");
            cust_id.addItem(item:String.valueOf(i: custID));
        }

    } catch (SQLException ex) {
        Logger.getLogger(name:OrderDetails.class.getName()).log(level:Level.SEVERE, msg: null, thrown: ex);
    }
}
```

Figure 2.97: Retrieve Customer ID

- Statement st = DBconnection.createDBconnection(). createStatement (); This line establishes a database connection and creates a Statement object st for executing SQL queries.
- ResultSet rst = st. executeQuery ("SELECT cust_ID FROM doctor"); This line executes a SQL SELECT query to retrieve all customer IDs (cust_ID) from the doctor table. The results are stored in a ResultSet object rst. while (rst. next ()) This loop iterates through each row in the result set.int custID = rst. getInt("cust_ID"); Inside the loop, it retrieves the customer ID from the current row of the result set and stores it in an integer variable
- custID.cust_id. addItem (String.valueOf(custID)); It adds the retrieved customer ID (converted to a string) to the GUI component cust_id, presumably a drop-down list or combo box, using the addItem method.
- The try-catch block handles any SQLException that might occur during the database operation. If an exception occurs, it is logged using a Logger object, associating it with the class OrderDetails.

- Get order ID- fills the customer ID dropdown with customer IDs that are retrieved from the database. By getting the maximum number of order IDs that are currently in the database and increasing it by one, this technique makes sure that every new order that is created in the system has a unique order ID.

```
// get order ID
public void getOrderID() {
    try {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery(sql: "SELECT MAX(OID) FROM ordersnew");

        while (rst.next()) {
            int maxOID = rst.getInt(columnIndex: 1); // Get the first column (column index 1) from the result set
            id.setText(t: String.valueOf(maxOID + 1)); // Set the text of id with the maxOID
        }
    } catch (SQLException ex) {
        Logger.getLogger(name:OrderDetails.class.getName()).log(level:Level.SEVERE, msg: null, thrown: ex);
    }
```

Figure 2.98: Retrieve Order ID

- This method, getOrderID (), retrieves the maximum value of the OID column from the ordersnew table in the database and sets it as the text of a GUI component, likely a text field or label named id.
- ResultSet rst = st. executeQuery ("SELECT MAX(OID) FROM ordersnew"); This line executes a SQL query to retrieve the maximum value of the OID column from the ordersnew table. The MAX(OID) function returns the highest value in the OID column. The result is stored in a ResultSet object rst. while (rst. next ()) {This loop iterates through each row in the result set.
- int maxOID = rst. getInt (1); Inside the loop, it retrieves the value of the first column (column index 1) from the current row of the result set and stores it in an integer variable maxOID. Since there's only one column in the result set (the maximum OID value), we retrieve it directly using getInt (1).
- id. setText (String.valueOf(maxOID + 1)); It sets the text of a GUI component named id, presumably a text field or label, to the value of maxOID incremented by 1. This likely prepares the GUI component to display the next available order ID.
- The try-catch block handles any SQLException that might occur during the database operation. If an exception occurs, it is logged using a Logger object, associating it with the class OrderDetails.

- Get order details- This procedure makes sure that the user has access to the most recent data about orders and the customers who are linked with them by dynamically retrieving order details and customer names from the database and displaying them in a table within the application's graphical user interface.

```
public void getOrderDetails() {
    try {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery("SELECT onew.OID,onew.CID,doc.f_name AS CustomerName,onew.date,onew.netamount,onew.order_status FROM "
                + "ordersnew onew INNER JOIN doctor doc ON onew.CID = doc.cust_id;");

        DefaultTableModel tblmodel = (DefaultTableModel) ordersTable.getModel();

        tblmodel.setRowCount(rowCount:0);
        while (rst.next()) {
            String OID = String.valueOf(obj: rst.getString(columnLabel: "OID"));
            String CID = String.valueOf(obj: rst.getString(columnLabel: "CID"));
            String MID = String.valueOf(obj: rst.getString(columnLabel: "CustomerName"));
            String Net = String.valueOf(obj: rst.getString(columnLabel: "netamount"));
            String Date = String.valueOf(obj: rst.getString(columnLabel: "date"));
            String status = String.valueOf(obj: rst.getString(columnLabel: "order_status"));

            String tbData[] = {OID, CID, MID, Net, Date, status};

            tblmodel.addRow(rowData: tbData);
        }
    } catch (SQLException ex) {
        Logger.getLogger(name:OrderDetails.class.getName()).log(level:Level.SEVERE, msg: null, thrown: ex);
    }
}
```

Figure 2.99: Get order details

- Logout button-Closes the current window and opens the “loginform” window

```
private void LogoutBtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Logout Button
    this.dispose();
    Loginform obj = new Loginform();
    obj.show();
}
```

Figure 2.100: Logout button

- Back button-Closes the current window and opens the dashboard window

```
private void backBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // Back Button -> navigates to the Dashboard
    this.dispose();
    Dashboard obj = new Dashboard();
    obj.show();
}
```

Figure 2.101: Back button

- Update button-Updated the selected row in the cart table

```
private void updateBtnActionPerformed(java.awt.event.ActionEvent evt) {
    String medID = String.valueOf(obj: medicine_ID.getSelectedItem());
     String medName = med_Name.getText();
     int qty = Integer.parseInt(s: quantity.getText());
     int unitPrice = Integer.parseInt(s: unit_Price.getText());
     int total = Integer.parseInt(s: tot.getText());

     String qtyString = String.valueOf(i: qty);
     String unitPriceString = String.valueOf(i: unitPrice);
     String totalString = String.valueOf(i: total);

     String data[] = {medID, medName, qtyString, unitPriceString, totalString};

     DefaultTableModel tblModel = (DefaultTableModel) Cart.getModel();
     int selectedRow = Cart.getSelectedRow();
     tblModel.removeRow(row: selectedRow);
     tblModel.addRow(rowData: data);
}
```

Figure 2.102: Medicine Cart Update Button

- String medID = String.valueOf(medicine_ID. getSelectedItem ()); Retrieves the selected item from a drop-down list (medicine_ID) containing medicine IDs and converts it to a string.
- String medName = med_Name.getText(); Retrieves text from a text field named med_Name, presumably containing the name of the medicine.
- int qty = Integer.parseInt(quantity. getText ()); Retrieves text from a text field named quantity and converts it to an integer, representing the quantity of the selected medicine.
- int unitPrice = Integer.parseInt(unit_Price.getText()); Retrieves text from a text field named unit_Price and converts it to an integer, representing the unit price of the selected medicine.

- int total = Integer.parseInt(tot. getText ()); Retrieves text from a text field named tot and converts it to an integer, representing the total price of the selected medicine. Creates string variables qtyString, unitPriceString, and totalString to convert the integer values back to strings for storing in the data array. Creates an array data containing the collected information about the selected medicine. Retrieves the table model associated with a table component named Cart.Retrieves the index of the currently selected row in the Cart table. Removes the currently selected row from the table model. Adds a new row to the table model with the updated information, effectively updating the details of the selected medicine in the cart display.

➢ Refresh order tables- this method fetches the latest order details from the database, constructs a new data model based on this data, and updates the display of the ordersTable with the refreshed data. This ensures that the user, interface displays the most current data that is accessible in the database.In order to run SQL queries, it builds a Statement object and establishes a database connection.It runs a SQL query to retrieve every entry from the orders table.Iterating through the result set, the while loop creates an object array row with the information for each row retrieved from the database. Order ID, Customer ID, Medicine ID, Quantity, Total Amount, and Order Date are all included in each row. It adds the built object array row to the model for each row in the result set, which represents the table's data model. After iterating through all rows in the result set, it sets the updated model containing the fetched data to the ordersTable, effectively refreshing the table display with the latest data. In case of any SQLException occurring during the database operation, it catches the exception, displays an error message using JOptionPane.showMessageDialog(), and logs the error message for debugging purposes

Figure 2.103: Refresh Order Table

- Add to cart action performed- This section of code makes it easier to add products to a shopping cart or view order details by obtaining pertinent data from user input fields and showing it in the graphical user interface of the application.

```
private void AddToCartBtnActionPerformed(java.awt.event.ActionEvent evt) {
  //Add Medicine To The Cart Button
    isupdate = false;
    String medID = String.valueOf(obj: medicine_ID.getSelectedItem());
    String medName = med_Name.getText();
    int qty = Integer.parseInt(s: quantity.getText());
    int unitPrice = Integer.parseInt(s: unit_Price.getText());
    int total = Integer.parseInt(s: tot.getText());

    String qtyString = String.valueOf(i: qty);
    String unitPriceString = String.valueOf(i: unitPrice);
    String totalString = String.valueOf(i: total);

    String data[] = {medID, medName, qtyString, unitPriceString, totalString};

    DefaultTableModel tblModel = (DefaultTableModel) Cart.getModel();
    tblModel.addRow(rowData: data);

    clear();
}
```

Figure 2.104: Add to Cart Button

- isupdate = false; This line sets a boolean variable isupdate to false, indicating that this action is not an update of an existing item in the cart, but rather adding a new item. It retrieves various information about the selected medicine and quantity entered by the user

- String medID = String.valueOf(medicine_ID. getSelectedItem ()); Retrieves the selected item from a drop-down list (medicine_ID) containing medicine IDs and converts it to a string
- String medName = med_Name.getText(); Retrieves text from a text field named med_Name, presumably containing the name of the medicine
- int qty = Retrieves text from a text field named quantity and converts it to an integer, representing the quantity of the selected medicine.
- int unitPrice = Integer.parseInt(unit_Price.getText()); Retrieves text from a text field named unit_Price and converts it to an integer, representing the unit price of the selected medicine.
- int total = Integer.parseInt(tot. getText ()); Retrieves text from a text field named tot and converts it to an integer, representing the total price of the selected medicine. It converts the integer values for quantity, unit price, and total back to strings for storing in an array:String qtyString = String.valueOf(qty);,String unitPriceString = String.valueOf(unitPrice);,String totalString = String.valueOf(total);It constructs an array data containing the collected information about the selected medicine.It retrieves the table model connected to the Cart table component.The details of the chosen medication are essentially added to the cart display by creating a new row in the table model using the previously gathered data.Lastly, it invokes the clear() function, which is probably in charge of clearing certain fields or states in the application in order to get ready to add more items or conduct additional interactions.

➢ Calculate total amount- This method, calculateTotalAmount (), is responsible for calculating the total amount of an order based on the quantity and unit price of the selected medicine

```
private void calculateTotalAmount() {
    try {
        int quantity = Integer.parseInt(s: net_Amt.getText());
        String selectedItem = (String) cust_id.getSelectedItem();

        // Check whether the selectedItem is null
        if (selectedItem != null) {
            String parts[] = selectedItem.split(regex: " - ");
            int medID = Integer.parseInt(parts[1]);

            // Fetch the unit price from the database using medID
            double unitPrice = getUnitPrice(medID);

            double totalAmount = quantity * unitPrice;
            tot.setText(t: String.valueOf(d: totalAmount));
        } else {
            tot.setText(t: "0");
        }
    } catch (NumberFormatException e) {
        tot.setText(t: "0");
    }
}
```

Figure 2.105: Calculate total amount of the selected medicine

- int quantity = Integer.parseInt(net_Amt.getText()); Retrieves the quantity entered by the user from a GUI component, presumably a text field named net_Amt, and parses it into an integer.
- String selectedItem = (String) cust_id. getSelectedItem (); Retrieves the selected item from a drop-down list or combo box named cust_id, which likely contains information about the selected medicine, such as its ID and name. It checks if the selected item is not null: If the selected item is not null, it splits the string to extract the medicine ID. The drug ID is then passed as a parameter to the getUnitPrice () method, which retrieves the unit price of the chosen medication from the database. By multiplying the quantity by the unit price, the total amount is determined. It sets the computed order total as the text of a GUI component called tot, which is likely to show the order total. In the event that a NumberFormatException arises when converting a quantity or unit pricing, the error is handled gently by setting the total amount to "0".

➢ Get unit price- Retrieves the unit price of a medicine from the database. This code segment defines a method named getUnitPrice that retrieves the unit price of a medicine from a database table (medicine) based on the provided medicine ID (medID).

```
private double getUnitPrice(int medID) {
    try {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery("SELECT unit_price FROM medicine WHERE med_ID = " + medID);

        if (rs.next()) {
            return rs.getDouble(columnLabel: "unit_price");
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
    return 0;
}
```

Figure 2.106: Get Unit Price of the selected medicine

private double getUnitPrice (int medID) {: This line defines the method signature. It specifies that the method returns a double value and takes an integer parameter medID, representing the ID of the medicine for which the unit price needs to be retrieved. Inside the method, a try-catch block is used to handle potential SQLExceptions that might occur during database operations.Within the try block:It establishes a database connection and creates a Statement object (st) for executing SQL queries.It executes a SQL query to fetch the unit price (unit_price) from the medicine table based on the provided medicine ID (medID).If the query returns a result (if rs.next() evaluates to true), it retrieves the unit price from the result set (rs) using rs.getDouble("unit_price") and returns it. If any SQLException occurs during the database operation, it's caught by the catch block, and the exception is printed using ex. printStackTrace () for debugging purposes. If no result is found or if an exception occurs, the method returns 0, indicating that the unit price could not be retrieved

- Net amount key pressed- this code ensures that the user can only input numerical values for the quantity field (net_Amt) by preventing input of alphabetical characters and displaying an error message if such characters are entered

```
private void net_AmtKeyPressed(java.awt.event.KeyEvent evt) {
    //Ensures that the user enters only the numerical values for quantity
    char c = evt.getKeyChar();
    //Validating the user inputs for the quantity
    if (Character.isLetter(ch: c)) {
        net_Amt.setEditable(b: false);
        JOptionPane.showMessageDialog(parentComponent: null, message: "Enter Numerical Values for the Quantity",
                title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
        net_Amt.setEditable(b: true);

    }
}
```

Figure 2.107: Validate the input for the Net Amount

- Delete button action performed-Delete the selected row from the cart table

```
private void DeleteBtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Delete Button
    DefaultTableModel tblmodel = (DefaultTableModel) Cart.getModel();

    if (Cart.getSelectedRowCount() > 0) {
        int selectedRow = Cart.getSelectedRow();
        String total = (String) tblmodel.getValueAt(row: selectedRow, column: 4); // assuming the total amount is in the 6th column (index 5)

        int totalAmount = Integer.parseInt(s: total);

        tblmodel.removeRow(row: selectedRow);

        if (Cart.getRowCount() == 0) {
            net_Amt.setText(t: "");
        } else {
            int net = nettotal - totalAmount;
            net_Amt.setText(t: String.valueOf(i: net));
        }
        System.out.println(x: totalAmount);
    } else if (Cart.getRowCount() == 0) {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Error: No rows selected.");
    } else {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Error: Please select a row to delete.");
    }

}
```

Figure 2.108: Delete an Order Entry

DefaultTableModel tblmodel = (DefaultTableModel) Cart.getModel(); Retrieves the table model associated with a table component named Cart, presumably representing a shopping cart or order details. Checks if any row is selected in the Cart table:If at least one row is selected (Cart.getSelectedRowCount() > 0), it proceeds to delete the selected row(s).If no row is selected but the table is empty (Cart.getRowCount() == 0), it displays an error message indicating that no rows are selected.If the table is not empty but no row is selected, it displays an error message prompting the user to select a row to delete.If a row is selected:It retrieves the index of the selected row (selectedRow).It retrieves the total amount from the selected row in the table (total) assuming it's in the 5th column (index 4).

It parses the total amount from a string to an integer (totalAmount).It removes the selected row from the table model using tblmodel.removeRow(selectedRow).If the table becomes empty after deleting the row, it clears the text of a GUI component named net_Amt.Otherwise, it calculates the new net total by subtracting the deleted total amount from the previous net total (nettotal) and updates the text of the net_Amt component with the new value.If no row is selected:It prints an error message to the console indicating the total amount of the deleted rows.

- Get medicine Id-Retrieves medicine IDs from the database and populates the medicine ID dropdown.

```
// GET MEDICINE ID FROM DATABSE
public int medicineID;
public String medicineName;

// get medicine ID
public void getMedicineID() {
    try {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery(sql: "SELECT med_ID FROM medicine");

        while (rst.next()) {
            int medID = rst.getInt(columnLabel: "med_ID");
            medicine_ID.addItem(item:String.valueOf(i: medID));
        }

    } catch (SQLException ex) {
        Logger.getLogger(name:OrderDetails.class.getName()).log(level:Level.SEVERE, msg: null, thrown: ex);
    }
}
```

Figure 2.109: Get medicine Id

Inside a try catch block, it establishes a database connection using a method DBconnection.createDBconnection() and creates a Statement object st for executing SQL queries. It executes a SQL SELECT query "SELECT med_ID FROM medicine" to fetch all medicine IDs from the medicine table. Using a while loop, it iterates through the result set rst containing the retrieved medicine IDs. For each medicine ID retrieved from the result set, it retrieves the integer value using rst. getInt("med_ID").

It adds each medicine ID to the medicine_ID GUI component using addItem () method, converting the integer value to a string. Any SQLException that occurs during the database operation is caught and logged for error handling using a logger associated with the OrderDetails class.

- Get total-Calculate the total amount based on quality and unit price.

The getTotal () method retrieves the quantity and unit price from user input, calculates the total price based on these values, and updates a GUI component to display the calculated total. total = Quantity * unit_price; This line calculates the total price by multiplying the quantity (Quantity) by the unit price (unit_price) and stores the result in the total variable.

```
public int Quantity, unit_price, total;

private void getTotal() {
    Quantity = Integer.parseInt(s: quantity.getText());
    unit_price = Integer.parseInt(s: unit_Price.getText());

    total = Quantity * unit_price;
    tot.setText(t: String.valueOf(i: total));
}
```

Figure 2.110: Get total amount

- Search by the OrderID key pressed-Filter the orderTable based on the input order ID

```
private void SearchByTheOrderIDKeyPressed(java.awt.event.KeyEvent evt) {
    DefaultTableModel obj = (DefaultTableModel) ordersTable.getModel();
    TableRowSorter<DefaultTableModel> obj1 = new TableRowSorter<>(model: obj);
    ordersTable.setRowSorter(sorter: obj1);
    obj1.setRowFilter(filter: RowFilter.regexFilter(regex: SearchByTheOrderID.getText()));
}
```

Figure 2.111: Search Orders

- Customer ID action performed- This code segment manages the event that occurs when a customer ID is chosen from the combo box cust_id. It then extracts the medicine ID from the selected item and may carry out additional operations based on this data, like assigning the extracted medicine ID to a text field.

```
private void cust_idActionPerformed(java.awt.event.ActionEvent evt)
    // Extract med_ID from the selected combo box item
    String selectedCustomerID = (String) cust_id.getSelectedItem();

    // Check if the selectedCustomerID is not null before splitting
    if (selectedCustomerID != null) {
        String medID = selectedCustomerID.split(regex: " - ")[1];
        // Set med_ID to the respective JTextField
    }
}
```

Figure 2.112: Customer ID action performed method

- Netamount keyReleased- based on the net amount, determines the total amount.

```
private void net_AmtKeyReleased(java.awt.event.KeyEvent evt) {
    calculateTotalAmount();
}
```

Figure 2.113: Calculates the total amount

- Reorder button action performed- The current window is closed, and the ReOrdering window is opened.

```
private void ReOrderBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // Re Order Page Button
    ReOrdering obj = new ReOrdering();
    obj.show();
    // Close the Order Management Page
    this.dispose();
}
```

Figure 2.114: Reorder Page Navigation

- Add button action performed-Add order details to the Database

```
private void AddBtnActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        int cid = cust_id.getSelectedIndex();
        int netamount = Integer.parseInt(s: net_Amt.getText());
        SimpleDateFormat sfd = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        String orderDate = sfd.format(date:date.getDate());

        Statement st = pharmacy.DBconnection.createDBconnection().createStatement();

        int count = st.executeUpdate("INSERT INTO ordersnew (CID, date, netamount) VALUES ('" + cid + "', '" + orderDate + "', '" + netamount + "')");

        if (count > 0) {
            JOptionPane.showMessageDialog(parentComponent: null, message: "New Order's Details Added Successfully", title: "Success", messageType: JOptionPane.
                    INFORMATION_MESSAGE);
        }
    } catch (SQLException ex) {
        Logger.getLogger(name:OrderDetails.class.getName()).log(level:Level.SEVERE, msg: null, thrown: ex);
    }
    try {
        int oid = Integer.parseInt(s: id.getText());
        Connection conn = pharmacy.DBconnection.createDBconnection(); // Establish database connection
        String insertQuery = "INSERT INTO orderdetails (OID, MID, Mname, qty, unitprice, total) VALUES ('" + oid + "', ?, ?, ?, ?, ?)";

        PreparedStatement pstmt = conn.prepareStatement(sql: insertQuery);

        for (int row = 0; row < Cart.getRowCount(); row++) {

            String MID = Cart.getValueAt(row, column: 0).toString(); // Assuming column index 1 for MID
            String Mname = Cart.getValueAt(row, column: 1).toString(); // Assuming column index 2 for Mname
            String qty = Cart.getValueAt(row, column: 2).toString(); // Assuming column index 3 for qty
            String unitPrice = Cart.getValueAt(row, column: 3).toString(); // Assuming column index 4 for unitprice
            String total = Cart.getValueAt(row, column: 4).toString(); // Assuming column index 5 for total
```

```
private void AddBtnActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        int cid = cust_id.getSelectedIndex();
        int netamount = Integer.parseInt(s: net_Amt.getText());
        SimpleDateFormat sfd = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        String orderDate = sfd.format(date:date.getDate());

        Statement st = pharmacy.DBconnection.createDBconnection().createStatement();

        int count = st.executeUpdate("INSERT INTO ordersnew (CID, date, netamount) VALUES ('" + cid + "', '" + orderDate + "', '" + netamount + "')");

        if (count > 0) {
            JOptionPane.showMessageDialog(parentComponent: null, message: "New Order's Details Added Successfully", title: "Success", messageType: JOptionPane.
                    INFORMATION_MESSAGE);
        }
    } catch (SQLException ex) {
        Logger.getLogger(name:OrderDetails.class.getName()).log(level:Level.SEVERE, msg: null, thrown: ex);
    }
    try {
        int oid = Integer.parseInt(s: id.getText());
        Connection conn = pharmacy.DBconnection.createDBconnection(); // Establish database connection
        String insertQuery = "INSERT INTO orderdetails (OID, MID, Mname, qty, unitprice, total) VALUES ('" + oid + "', ?, ?, ?, ?, ?)";

        PreparedStatement pstmt = conn.prepareStatement(sql: insertQuery);

        for (int row = 0; row < Cart.getRowCount(); row++) {

            String MID = Cart.getValueAt(row, column: 0).toString(); // Assuming column index 1 for MID
            String Mname = Cart.getValueAt(row, column: 1).toString(); // Assuming column index 2 for Mname
            String qty = Cart.getValueAt(row, column: 2).toString(); // Assuming column index 3 for qty
            String unitPrice = Cart.getValueAt(row, column: 3).toString(); // Assuming column index 4 for unitprice
            String total = Cart.getValueAt(row, column: 4).toString(); // Assuming column index 5 for total
            //  pstmt.setString(1, OID);
            pstmt.setString(parameterIndex: 1, x: MID);
            pstmt.setString(parameterIndex: 2, x: Mname);
            pstmt.setString(parameterIndex: 3, x: qty);
            pstmt.setString(parameterIndex: 4, x: unitPrice);
            pstmt.setString(parameterIndex: 5, x: total);

            pstmt.executeUpdate();
        }

        pstmt.close();
        conn.close();

        JOptionPane.showMessageDialog(parentComponent: null, message: "Data inserted successfully!");
    } catch (SQLException e) {
        e.printStackTrace();
        JOptionPane.showMessageDialog(parentComponent: null, "Error inserting data into database: " + e.getMessage());
    }
    getOrderDetails();
}
```

Figure 2.115: Add order details

It retrieves necessary information from the GUI components.cid: Retrieves the selected index from the combo box cust_id. netamount: Parses the text from the text field net_Amt to get the net amount. orderDate: Formats the date selected in the date component into a string in the format "yyyy-MM-dd". It establishes a database connection and creates a statement (st) to execute SQL queries. It inserts the order details (CID, date, netamount) into the ordersnew table using an SQL INSERT query. If the insertion is successful (count > 0), it displays a success message. It inserts the details of items in the order (OID, MID, Mname, qty, unitprice, total) into the orderdetails table using a prepared statement (pstmt). It iterates through the rows of the Cart table, retrieves the details of each item, and executes the prepared statement for each item. After inserting all items into the database, it closes the prepared statement and the database connection. If any SQL exception occurs during the database operations, it prints the exception stack trace and displays an error message. Finally, it calls the getOrderDetails () method, presumably to update the displayed order details after adding the new order.

- Medicine ID state changed-Retrieve medicine details based on the selected medi ID

```
private void medicine_IDItemStateChanged(java.awt.event.ItemEvent evt) {
    try {
        Statement st = DBconnection.createDBconnection().createStatement();

        int id = medicine_ID.getSelectedIndex();
        ResultSet rst = st.executeQuery("SELECT med_name,unit_price FROM medicine where med_ID = '" + id + "'");

        while (rst.next()) {
            String medName = rst.getString(columnLabel: "med_name");
            String medUnitPrice = rst.getString(columnLabel: "unit_price");
            med_Name.setText(t: medName);
            unit_Price.setText(t: medUnitPrice);
        }
    } catch (SQLException ex) {
        Logger.getLogger(name:OrderDetails.class.getName()).log(level:Level.SEVERE, msg: null, thrown: ex);
    }
}
```

Figure 2.116: Retrieve medicine details

- Statement st = DBconnection.createDBconnection(). createStatement (); This line establishes a database connection using a method DBconnection.createDBconnection() and creates a Statement object st for executing SQL queries.

- int id = medicine_ID. getSelectedIndex (); This line retrieves the index of the selected item in the combo box medicine_ID.
- ResultSet rst = st. executeQuery ("SELECT med_name, unit_price FROM medicine where med_ID = '" + id + "'"); This line executes an SQL SELECT query to retrieve the medicine name (med_name) and unit price (unit_price) from the medicine table based on the selected medicine ID (id). The retrieved data is stored in a ResultSet object rst. Inside a while loop: For each row in the result set rst, it retrieves the medicine name and unit price. It assigns the retrieved medicine name to a text field named med_Name and the unit price to another text field named unit_Price. This presumably updates the GUI components to display the selected medicine's name and unit price. Any SQLException that occurs during the database operation is caught and logged for error handling using a logger associated with the OrderDetails class.

➢ quantityActionPerformed - Calculates the total amount and updates the net total

```
private void quantityActionPerformed(java.awt.event.ActionEvent evt) {
    //  int total = 0;
    if (!isupdate) {
        int qty = Integer.parseInt(s: quantity.getText());
        int unitPrice = Integer.parseInt(s: unit_Price.getText());

        total = unitPrice * qty;
        nettotal = nettotal + total;

        tot.setText(t: String.valueOf(i: total));
        net_Amt.setText(t: String.valueOf(i: nettotal));
    } else {
        int qty = Integer.parseInt(s: quantity.getText());
        int unitPrice = Integer.parseInt(s: unit_Price.getText());

        total = unitPrice * qty;

        tot.setText(t: String.valueOf(i: total));

        total = unitPrice * qty;

        if (totalAmount > total) {

            int x = totalAmount - total;

            net_Amt.setText(t: String.valueOf(nettotal - x));
        }
    }
}
```

Figure 2.117: Calculates the total amount and updates the net total

- It checks whether the update mode (isupdate) is set to false: If isupdate is false, it means a new item is being added to the quantity. It parses the quantity and unit price from their respective text fields (quantity and unit_Price), calculates the total price (total) by multiplying the quantity and unit price, updates the total price text field (tot), and updates the net total amount (nettotal) by adding the new total to the existing net total. Finally, it updates the net total text field (net_Amt) with the new net total. If isupdate is true, it means an existing item's quantity is being updated. It recalculates the total price (total) based on the updated quantity and unit price, updates the total price text field (tot) with the new total, and adjusts the net total amount (net_Amt) if the total amount of the item changes It deducts the difference from the net total if the total amount drops. Runtime errors are likely to arise from any problems with non-integer inputs or other exceptions because this code does not explicitly address mistakes that arise during the parsing of integer values or calculations.

- Cart MouseClicked – Retrieves details of the selected row in the Cart table

```
private void CartMouseClicked(java.awt.event.MouseEvent evt) {
    int i = Cart.getSelectedRow();
    isupdate = true;
    TableModel model = Cart.getModel();

    // cmb_medId.setSelectedIndex((int) model.getValueAt(i, 0));
    med_Name.setText((String) model.getValueAt(rowIndex:i, columnIndex: 1));
    quantity.setText((String) model.getValueAt(rowIndex:i, columnIndex: 2));
    unit_Price.setText((String) model.getValueAt(rowIndex:i, columnIndex: 3));

    String totalAmountStr = (String) model.getValueAt(rowIndex:i, columnIndex: 4);
    totalAmount = Integer.parseInt(s: totalAmountStr);
}
```

Figure 2.118: Retrieves details of the selected row in the Cart table

Order Status Page - Order Management Page (IT22884138 – RATHNAYAKA R.M.T.D)

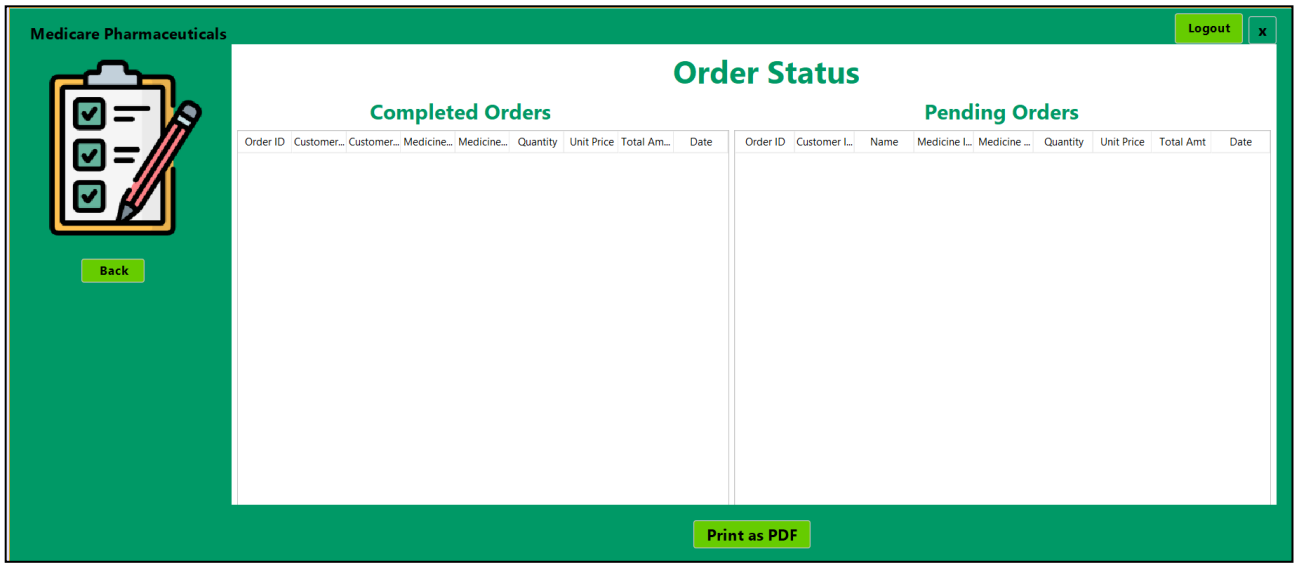

Figure 2.119: Order Status Page UI

The Order Status page's primary purpose is to show order data and classify them according to their current status (complete or pending). Order details such as order ID, customer ID, medicine ID, quantity, unit price, total amount, and order date are normally displayed once the order data is retrieved from the database and presented in tabular form. It also offers choices like returning to the Order Management page and printing the order details. This page functions as a dashboard for managing and keeping track of orders, giving a summary of their specifics and current status.

Major Module Structures in OrderDetails Class

Initial Components:

```
public class OrderStatus extends javax.swing.JFrame {

    /**
     * Creates new form OrderStatus
     */
    public OrderStatus() {
        initComponents();
        getCompleteOrderDetails();
        getPendingOrderDetails();
    }
```

Figure 2.120: Initialize Order Status Page Components

The constructor retrieves the order data and initialises the JFrame's component parts. In order to set up the GUI, it calls initComponents (). It also calls methods to retrieve and display order data. OrderStatus () is the OrderStatus class's constructor. When an instance of OrderStatus is created, it is invoked. InitialComponents method, which is produced by the IDE (such as NetBeans), is in charge of configuring the GUI elements, including buttons, tables, panels, and so on. Usually generated automatically, this method contains all the code required to create and arrange the form's visual elements. orderDetails.getComplete() function is used to retrieve the information from the database about orders that have been flagged as "Complete" and presents it in the relevant GUI.

PrintButtonClicked:

```
private void Print_BtnMouseClicked(java.awt.event.MouseEvent evt) {

    //Monthly Attendance Report Print Button
    PrinterJob job = PrinterJob.getPrinterJob();
    job.setJobName(jobName: "Print Data");

    job.setPrintable(new Printable()
    {
        public int print(Graphics pg , PageFormat pf , int pageNum)
        {
            pf.setOrientation(orientation: PageFormat.PORTRAIT);
            if(pageNum > 0)
            {
                return Printable.NO_SUCH_PAGE;
            }
            Graphics2D g2 = (Graphics2D) pg;
            g2.translate (tx: pf.getImageableX() , ty: pf.getImageableY());
            g2.scale(sx: 0.47, sy: 0.47);

            printable_panel.print(g: g2);

            return Printable.PAGE_EXISTS;
        }
    });
    boolean ok = job.printDialog();
    if(ok)
    {
        try
        {
            job.print();
        }
        catch(PrinterException ex)
            {
                ex.printStackTrace();
            }
     }

}
```

Figure 2.121: Print Order Status Report

getCompleteOrderDetails: this method retrieves complete order details from the database and populates them into the completeOrderTbl table in the GUI.

It establishes a database connection and creates a statement to execute a SQL query. The SQL query selects various fields (OID, CustomerID, MID, qty, unitprice, netamount, date) from two tables (ordersnew and orderdetails) where the order status is "Complete". It retrieves the results of the query as a ResultSet (rst) It accesses the completeOrderTbl table's model (DefaultTableModel) to clear its existing rows (tblmodel. setRowCount (0)) before populating it with new data It iterates through each row of the ResultSet (rst) and extracts values for each column (OID, CustomerID, MID, qty, unitprice, netamount, date). For each row, it creates an array of strings (tbData) containing the values of each column. It adds the array of data (tbData) to the table model, effectively populating the table with the retrieved order details. If any SQLException occurs during the database operation, it logs the exception using a Logger instance.

```
public void getCompleteOrderDetails(){

try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery("SELECT onew.OID, onew.CID AS CustomerID,od.MID,od.qty,od.unitprice,onew."
                + "netamount,onew.date FROM ordersnew onew INNER JOIN orderdetails od ON onew.OID = od.OID where onew.order_status = \"Complete\";");

        DefaultTableModel tblmodel = (DefaultTableModel) completeOrderTbl.getModel();

        tblmodel.setRowCount(rowCount:0);
        while (rst.next())
        {
            String OID = String.valueOf(obj: rst.getString(columnLabel: "OID"));
            String CID = String.valueOf(obj: rst.getString(columnLabel: "CustomerID"));
            String MID = String.valueOf(obj: rst.getString(columnLabel: "MID"));
            String Qty = String.valueOf(obj: rst.getString(columnLabel: "qty"));
            String unit = String.valueOf(obj: rst.getString(columnLabel: "unitprice"));
            String Net = String.valueOf(obj: rst.getString(columnLabel: "netamount"));
            String Date = String.valueOf(obj: rst.getString(columnLabel: "date"));

            String tbData[] = {OID, CID, MID,Qty, unit, Net, Date};

            tblmodel.addRow(rowData: tbData);
        }
    }
    catch (SQLException ex)
    {
        Logger.getLogger(name:OrderDetails.class.getName()).log(level:Level.SEVERE, msg: null, thrown: ex);
    }
```

Figure 2.122: Retrieve complete order details

getPendingOrderDetails: This method retrieves pending order details from the database and populates them into the pendingOrderTbl table in the GUI, it establishes a database connection and creates a statement (st) to execute a SQL query. The SQL query selects various fields (OID, CustomerID, MID, qty, unitprice, netamount, date) from two tables (ordersnew and orderdetails) where the order status is "pending". It retrieves the results of the query as a ResultSet (rst). It accesses the pendingOrderTbl table's model (DefaultTableModel) to clear its existing rows (tblmodel. setRowCount (0)) before populating it with new data. It iterates through each row of the ResultSet (rst) and extracts values for each column (OID, CustomerID, MID, qty, unitprice, netamount, date). For each row, it creates an array of strings (tbData) containing the values of each column. It adds the array of data (tbData) to the table model, effectively populating the table with the retrieved order details.

```
public void getPendingOrderDetails(){

try
    {
        Statement st = DBconnection.createDBoonnection().createStatement();
        ResultSet rst = st.executeQuery("SELECT onew.OID, onew.CID AS CustomerID,od.MID,od.qty,od.unitprice,onew.netamount,onew.date FROM "
                + "ordersnew onew INNER JOIN orderdetails od ON onew.OID = od.OID where onew.order_status = \"pending\";");

        DefaultTableModel tblmodel = (DefaultTableModel) pendingOrderTbl.getModel();

        tblmodel.setRowCount(rowCount:0);
        while (rst.next())
        {
            String OID = String.valueOf(obj: rst.getString(columnLabel: "OID"));
            String CID = String.valueOf(obj: rst.getString(columnLabel: "CustomerID"));
            String MID = String.valueOf(obj: rst.getString(columnLabel: "MID"));
            String Qty = String.valueOf(obj: rst.getString(columnLabel: "qty"));
            String unit = String.valueOf(obj: rst.getString(columnLabel: "unitprice"));
            String Net = String.valueOf(obj: rst.getString(columnLabel: "netamount"));
            String Date = String.valueOf(obj: rst.getString(columnLabel: "date"));

            String tbData[] = {OID, CID, MID,Qty, unit, Net, Date};

            tblmodel.addRow(rowData: tbData);
        }
    }
    catch (SQLException ex)
    {
        Logger.getLogger(name:OrderDetails.class.getName()).log(level:Level.SEVERE, msg: null, thrown: ex);
    }


}
```

Figure 2.123: Retrieve pending order details

Employee Management Page (IT22319142 – WIJESINGHE A.G.T)

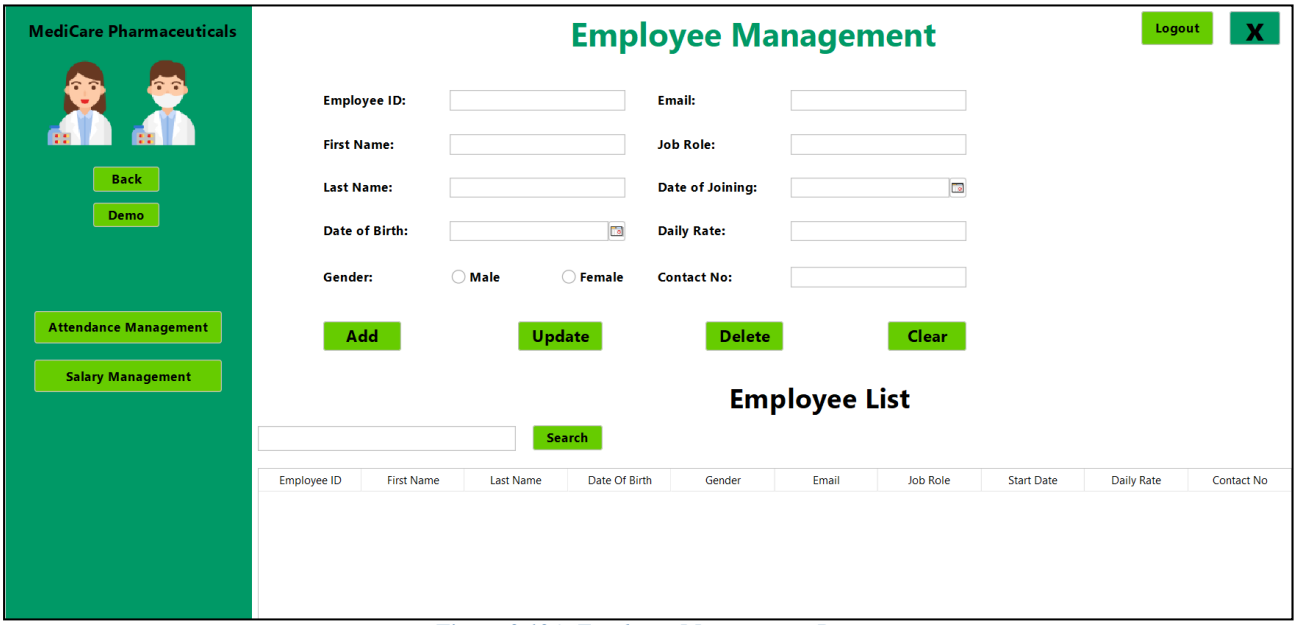

Figure 2.124: Employee Management Page

The Employee management function starts registering new employees, whose information is added to a centralized "Employee Details" table. By erasing entries from the database permanently, the pharmacist is still able to make changes to employee data and start the resignation procedure. Prominently, the feature allows the pharmacist to keep track of each employee's attendance with its special "Daily Attendance Tracking" feature. Based on daily attendance and the daily rate, monthly salary calculations are automatically generated.

It is possible to generate detailed salary reports at the end of each month. Medicare Pharmaceuticals' business operations are optimized overall because of this integrated approach to supplier and employee management, which guarantees a methodical and efficient workflow. As the result of this effective employee management function;

- Streamlines HR procedures
- Enhances the productivity

The functionality of the “Supplier Management Page” & the “Employee Management Page” are almost the same. Since I have explained the “Supplier Management Page Functionality” in detail, the Employee Management Page is not explained in depth.

❖ Employee Management Page Components Initialization Code:

```
package pharmacy;
//@author thath

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.time.Instant;
import java.util.Calendar;
import java.util.Date;
import javax.swing.ButtonGroup;
import javax.swing.DefaultComboBoxModel;
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableModel;


public class EmployeeDetails extends javax.swing.JFrame {
// Creates new form EmployeeDetails

    public EmployeeDetails()
    {
        initComponents();
        refreshEmployeeTable();

        id.setText(t: "Will Be Added Automatically");
        fname.setText(t: "");
        lname.setText(t: "");

        // Restricting the dob JDateChooser to only allow past dates
        dob.setDate(date: null); // Clear any default date set

        // Set maximum selectable date to yesterday's date
        Calendar cal = Calendar.getInstance();
        // Subtracting 1 day from today's date
        cal.add(field:Calendar.DAY_OF_MONTH, amount: -1);
        Date yesterday = cal.getTime();

        dob.setMaxSelectableDate(date: yesterday);
        // Create a ButtonGroup
        ButtonGroup attendanceGroup = new ButtonGroup();
        attendanceGroup.add(b: maleRadioButton);
        attendanceGroup.add(b: femaleRadioButton);

        // Set Male as the default selection
        maleRadioButton.setSelected(b: true);
        femaleRadioButton.setSelected(b: false);

        nic.setText(t: "");
        email.setText(t: "@gmail.com");

        //The method to populate the Job Role combo box
        getJobRole();

        // Set startDate JDateChooser to today's date only
        startDate.setDate(new Date()); // Set the date to today's date
        startDate.setMinSelectableDate(new Date()); // Disabling past dates
        startDate.setMaxSelectableDate(new Date()); // Disabling future dates

        dailyRate.setText(t: "0");
        phoneNo.setText(t: "");

        // Action listener to Demo button
        attachDemoButtonListener();
    }
```

Figure 2.125: Employee Management Page Components Initialization Code

The constructor sets default values for variables like ID and dates and initializes the GUI components. Additionally, it initializes the Job Role combo box and sets up action listeners.

❖ Demo Button Code of Employee Management Page:

```
private void attachDemoButtonListener()
{
    demo_Button.addActionListener(new ActionListener()
    {
        @Override
        public void actionPerformed(ActionEvent e)
        {
            fillDemoValues();
        }
    });
}
private void fillDemoValues()
{
    // Sample values
    String[] sampleValues =
    {
        "John", "Doe", "1980-01-01", "Male", "198056789012", "john@gmail.com",
        "Assistant Pharmacist", "3500", "0784567890"
    };

    // Filling fields with sample values
    fname.setText(sampleValues[0]);
    lname.setText(sampleValues[1]);

    try
    {
        dob.setDate(date: new SimpleDateFormat(pattern: "yyyy-MM-dd").parse(sampleValues[2]));
    }
    catch (ParseException ex)
    {
        ex.printStackTrace();
    }
    if (sampleValues[3].equals(anObject: "Male"))
    {
        maleRadioButton.setSelected(b: true);
        femaleRadioButton.setSelected(b: false);
    }
    else
    {
        maleRadioButton.setSelected(b: false);
        femaleRadioButton.setSelected(b: true);
    }
    nic.setText(sampleValues[4]);
    email.setText(sampleValues[5]);
    role.setSelectedItem(sampleValues[6]);
    dailyRate.setText(sampleValues[7]);
    phoneNo.setText(sampleValues[8]);
}
```

Figure 2.126: Employee Management Page Demo Button Code

When the Demo Button is clicked once, sample values are filled in into the corresponding fields for efficient testing or demonstration.

❖ **Method to Populate the Job Roles Combo Box with the Job Positions:**

```
// Populate the Job Roles combo box
private void getJobRole()
{
    DefaultComboBoxModel<String> model = new DefaultComboBoxModel<>();

    model.addElement(anObject: "Assistant Pharmacist");
    model.addElement(anObject: "Accountant");
    model.addElement(anObject: "Quality Control Analyst");
    model.addElement(anObject: "Sales and Marketing Representative");
    model.addElement(anObject: "Human Resources Manager");
    model.addElement(anObject: "Supply Chain Manager");

    role.setModel(aModel: model);
}
```

Figure 2.127: Populate The Job Role Combo Box

The following methods enable access to other pages and functions, such navigating to the dashboard, salary management page, or daily attendance tracking page.

❖ **Attendance Page Navigation Code:**

```
private void AttendancePageBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // Attendance Page Button
    EmployeeAttendance obj = new EmployeeAttendance();
   obj.show();
   // Close the Employee Management Page
   this.dispose();
}
```

Figure 2.128: Attendance Page Navigation Code

❖ **Salary Page Navigation Code:**

```
private void SalaryManagementPageBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // Salary Page Button
    EmployeeSalary obj = new EmployeeSalary();
    obj.show();

   // Close the Employee Management Page
   this.dispose();
}
```

Figure 2.129: Salary Page Navigation Code

❖ **Navigate Back to The Dashboard Code:**

```
private void BackBtn1ActionPerformed(java.awt.event.ActionEvent evt) {
    // Back Button -> navigates to the Dashboard
    this.dispose();
    Dashboard obj = new Dashboard();
    obj.show();
}
```

Figure 2.130: Dashboard Navigation Code

❖ Adding a New Employee's Details into the system database Code:

```
private void AddBtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Add Button
    String FirstName,LastName,DateOfBirth,Gender,Email, N_I_C, JobPosition,JoinedDate;
    int contactNo, daily_Rate;

    FirstName = fname.getText().trim();
    LastName = lname.getText().trim();

    N_I_C = nic.getText();
    Email = email.getText();

    // Check if any mandatory field is empty or has default values
    if (FirstName.isEmpty() || LastName.isEmpty() || N_I_C.isEmpty() || !Email.endsWith(suffix: "@gmail.com") || Email.equals(anObject: "@gmail.com"))
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Please fill all the required input fields!", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        return; // Stop processing if compulsory user input fields are empty
    }
    // Validating the date formatting
    Date selectedDate = dob.getDate();
    if (selectedDate == null)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Please select a valid Date of Birth!", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        return; // Stops processing if Date of Birth is not selected
    }

    //Validating the date formatting
    SimpleDateFormat sdf = new SimpleDateFormat(pattern: "yyyy-MM-dd");
    DateOfBirth = sdf.format(date: dob.getDate());

    //Take input for Male and Female radio buttons
    Gender = getSelectedGender();

    // Validating NIC length
    if (N_I_C.length() != 12)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Please enter 12 numerical values for the NIC!", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        return; // Stop processing,if NIC length is incorrect
    }
    // Check whether the NIC already exists in the database
    try
    {
        Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
        ResultSet existingNIC = st.executeQuery("SELECT * FROM employee WHERE NIC = '" + N_I_C + "'");

        if (existingNIC.next())
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Please check whether the NIC is correct! There is another registered employee with the same NIC you are trying to insert!",
                    title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
            return; // Stop processing further if the NIC already exists in employee table
        }
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while checking NIC: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        return; // Stop processing further if there is a database error
    }

    // Get Job Role from the combo box
    JobPosition = (String) role.getSelectedItem();

    //Validating the date formatting
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern: "yyyy-MM-dd");
    JoinedDate = simpleDateFormat.format(date: startDate.getDate());

    // User input validation for daily rate and contact no
    try
    {
        daily_Rate = Integer.parseInt(s: dailyRate.getText());
        contactNo = Integer.parseInt(s: phoneNo.getText());
    }
    catch (NumberFormatException e)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "You must enter a valid Input for Daily Rate & Contact Number...!", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        return; // Stop processing further if it is an invalid input
    }
    // Check if daily rate is 0
    if (daily_Rate == 0)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Please enter a valid Daily Rate!", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        return; // Stop processing if daily rate is still 0
    }
    //Validations of Database Interaction
    try
    {
        Statement st = pharmacy.DBconnection.createDBconnection().createStatement();

        //Fetch the next available Employee ID from the Database
        ResultSet rs = st.executeQuery(string: "SELECT MAX(emp_ID) FROM employee" );
        int nextEmployeeID = 1;

        if (rs.next())
        {
            nextEmployeeID =rs.getInt(i: 1) + 1;
        }

        int count = st.executeUpdate("INSERT INTO `employee`(`emp_ID`, `f_name`, `l_name`, `dob`, `gender`, `NIC` , `email`, `role`, `start_date`, `dailyRate`, `phone_no`) "
                + "VALUES ("" + nextEmployeeID + "','" + FirstName + "','" + LastName + "','" + DateOfBirth + "','" + Gender + "' , '" + N_I_C + "' , '" + Email + "','" + JobPosition + "','" + JoinedDate + "','"

        if (count > 0)
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "New Employee's Details Added Successfully", title:"Success", messageType: JOptionPane.INFORMATION_MESSAGE);

            //Refresh the table after adding the New Employee's Details
            refreshEmployeeTable();
        }
    }
     //Validations of Error Handling
    catch(NumberFormatException e)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "You must enter a valid Input for Daily Rate & Contact Number...!",title:"Error",messageType: JOptionPane.ERROR_MESSAGE);
    }
    catch(Exception e)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Something went wrong...!",title:"Error",messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.131: Code for Adding a New Employee

After the information given has been verified, this function handles adding new employee details to the database. It performs data validations and checks for required fields.

- **Method to get the selected gender from the Radio Buttons:**

```
private String getSelectedGender()
{
    if (maleRadioButton.isSelected())
        return "Male";

    else if (femaleRadioButton.isSelected())
        return "Female";

    else
        //When no option is selected
        return "Unknown";
}
```

Figure 2.132: Code for Retrieving the Gender Selection

- **Method to Refresh the Employee Table After Adding the New Employee:**

```
//Refresh the refreshEmployeeTable after updating/deleting the selected Employee's Details
private void refreshEmployeeTable()
{
    try
    {
        // Fetch updated data from the Database
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery(string: " SELECT * FROM employee ");

        // Create a new DefaultTableModel with updated data
        DefaultTableModel model = new DefaultTableModel();
        model.addColumn(columnName:"Employee ID");
        model.addColumn(columnName:"First Name");
        model.addColumn(columnName:"Last Name");
        model.addColumn(columnName:"Date Of Birth");
        model.addColumn(columnName:"Gender");
        model.addColumn(columnName:"NIC");
        model.addColumn(columnName:"Email");
        model.addColumn(columnName:"Job Role");
        model.addColumn(columnName:"Start Date");
        model.addColumn(columnName:"Daily Rate");
        model.addColumn(columnName:"Contact No");

        while (rs.next())
        {
            Object[] row =
            {
                rs.getInt(string: "emp_ID"),
                rs.getString(string: "f_name"),
                rs.getString(string: "l_name"),
                rs.getString(string: "dob"),
                rs.getString(string: "gender"),
                rs.getString(string: "NIC"),
                rs.getString(string: "email"),
                rs.getString(string: "role"),
                rs.getString(string: "start_date"),
                rs.getInt(string: "dailyRate"),
                rs.getInt(string: "phone_no"),
            };
            model.addRow(rowData: row);
        }

        // Set the updated model to the medicineTable
        EmployeeDetailsTable.setModel(dataModel: model);
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while refreshing Table: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.133: Code for Loading the Modified Table

By using this method, the latest information from the database is updated and displayed in the table.

❖ Loading the data in the Selected Row of the Employee Table:

```
private void EmployeeDetailsTableMouseClicked(java.awt.event.MouseEvent evt) {
    //Get the index of the selected row
    int selectedRow = EmployeeDetailsTable.getSelectedRow();

    //Establish the table model that represents the structure & data of the table
    TableModel model = EmployeeDetailsTable.getModel();

    //Validations to ensure that User interacts with the selected row data
    //Set the First Name,Last Name,Date Of Birth,Gender,Email,Job Position,Joined Date,Salary & Contact Number text fields to the value from the selected row
    fname.setText(s: model.getValueAt(rowIndex: selectedRow, columnIndex: 1).toString());
    lname.setText(s: model.getValueAt(rowIndex: selectedRow, columnIndex: 2).toString());

    //Set the Date Format for DateOfBirth
    SimpleDateFormat sdf = new SimpleDateFormat(pattern: "yyyy-MM-dd");
    String DateOfBirth = (String) model.getValueAt(rowIndex: selectedRow, columnIndex: 3);

    nic.setText(s: model.getValueAt(rowIndex: selectedRow, columnIndex: 5).toString());
    email.setText(s: model.getValueAt(rowIndex: selectedRow, columnIndex: 6).toString());

    // Set selected job role in the combo box
    String selectedRole = model.getValueAt(rowIndex: selectedRow, columnIndex: 7).toString();
    DefaultComboBoxModel<String> comboBoxModel = (DefaultComboBoxModel<String>) role.getModel();

    if (comboBoxModel != null)
    {
        role.setSelectedItem(anObject: selectedRole);
    }

    //Set the Date Format for JoinedDate
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern: "yyyy-MM-dd");
    String JoinedDate = (String) model.getValueAt(rowIndex: selectedRow, columnIndex: 8);

    dailyRate.setText(s: model.getValueAt(rowIndex: selectedRow, columnIndex: 9).toString());
    phoneNo.setText(s: model.getValueAt(rowIndex: selectedRow, columnIndex: 10).toString());

    //Retrieve the Employee ID from the First Column(index 0)of the selected row & store it
    id.setText(s: model.getValueAt(rowIndex: selectedRow, columnIndex: 0).toString());

    try
    {
      //Handle the parsing of the Birth Date
      Date DateofBirth = sdf.parse(source: DateOfBirth);
      dob.setDate(date: DateofBirth);
       //Handle the parsing of the Joined Date
       Date StartDate = simpleDateFormat.parse(source: JoinedDate);
       startDate.setDate(date: StartDate);

       // Retrieves and sets the gender
       String gender = (String) model.getValueAt(rowIndex: selectedRow, columnIndex: 4);
       if ("Male".equals(anObject: gender))
       {
         maleRadioButton.setSelected(b: true);
         femaleRadioButton.setSelected(b: false);
       }
       else if ("Female".equals(anObject: gender))
       {
         maleRadioButton.setSelected(b: false);
         femaleRadioButton.setSelected(b: true);
       }
       else
       {
         // If gender is neither Male nor Female
         maleRadioButton.setSelected(b: false);
         femaleRadioButton.setSelected(b: false);
       }
         // Disable radio buttons when user tries to update an existing data tuple
         maleRadioButton.setEnabled(b: false);
         femaleRadioButton.setEnabled(b: false);

         // Disable Date of Birth and Start Date fields when user tries to update an existing data tuple
         dob.setEnabled(bln: false);
         startDate.setEnabled(bln: false);
    }
    catch (ParseException e)
    {
        // Handle the parsing exception
        e.printStackTrace();
    }
}
```

Figure 2.134: Code of Table Mouse Click Event

The user can update or remove the selected record by clicking on a row in the table display, which fills in the fields with the employee's information.

❖ Updating the Details of a Registered Employee:

```
private void UpdateBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // Update Button
    try
    {
        String FirstName,LastName,DateOfBirth,Gender,Email,N_I_C,JobPosition,JoinedDate;
        int Daily_Rate,contactNo;

        FirstName = fname.getText();
        LastName  = lname.getText();

        //Validating the date formatting
        SimpleDateFormat sdf = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        DateOfBirth = sdf.format(date: dob.getDate());

        // Obtain input for Male and Female radio buttons
        Gender = getSelectedGender();

        N_I_C =nic.getText();
        Email = email.getText();

        // Get Job Role from the combo box
        JobPosition = (String) role.getSelectedItem();

        //Validating the date formatting
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        JoinedDate = simpleDateFormat.format(date: startDate.getDate());

        //User input validation
        Daily_Rate = Integer.parseInt(s: dailyRate.getText());
        contactNo = Integer.parseInt(s: phoneNo.getText());


        //Validations of Database Interaction
        try
        {
            Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
            int x = Integer.parseInt(s: id.getText());

            //Validations related to SQL
            String query = "UPDATE employee SET f_name = ?,l_name = ?, dob = ?, gender = ? , NIC = ? , email = ?, role = ? , start_date = ? , dailyRate = ? , phone_no = ?  WHERE emp_ID = ?";

            // Use PreparedStatement to avoid SQL injection
            java.sql.PreparedStatement prepSt = st.getConnection().prepareStatement(string: query);
             prepSt.setString(i: 1, string: FirstName);
             prepSt.setString(i: 2, string: LastName);
             prepSt.setDate(i: 3, date: java.sql.Date.valueOf(s: DateOfBirth));
             prepSt.setString(i: 4, string: Gender);
             prepSt.setString(i: 5, string: N_I_C);
             prepSt.setString(i: 6, string: Email);
             prepSt.setString(i: 7, string: JobPosition);
             // Assuming JoinedDate is a LocalDate
             prepSt.setDate(i: 8, date: java.sql.Date.valueOf(s: JoinedDate));
             prepSt.setInt(i: 9, i1:Daily_Rate);
             prepSt.setInt(i: 10, i1:contactNo);

             prepSt.setInt(i: 11, i1:x);

             int count = prepSt.executeUpdate();

             if (count > 0)
             {
                 JOptionPane.showMessageDialog(parentComponent: null, message: "Selected Employee's Details Updated Successfully", title: "Success",messageType: JOptionPane.INFORMATION_MESSAGE);

                 // Refresh the medicineTable after updating
                 refreshEmployeeTable();
             }
        }
        catch(Exception e)
        {
            e.printStackTrace();
            JOptionPane.showMessageDialog(parentComponent: null, message: "Error occurred while Updating the selected Employee's Details", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
        }
    }
    catch(NumberFormatException e)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Invalid Input for Daily Rate...!", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
    catch(Exception e)
    {
        //Generic Exception for any kind of exception
    }
}
```

Figure 2.135: Code for Updating a Registered Employee

The database's current employee information is updated using this method. It retrieves data from the user input fields, verifies it, and then updates the database.

❖ Deleting the Details of a Registered Employee:

```
private void DeleteBtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Delete Button
    int confirm = JOptionPane.showConfirmDialog(parentComponent: null, message: "Are you sure you want to 'Delete' this Employee?", title:"Confirm Deletion", optionType:JOptionPane.YES_NO_OPTION);
    if (confirm == JOptionPane.YES_OPTION)
    {
        try
        {
            Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
            String query = "DELETE FROM `employee` WHERE `emp_ID` = " + Integer.valueOf(s: EmployeeDetailsTable.getValueAt(row: EmployeeDetailsTable.getSelectedRow(), column: 0).toString());

            //executeUpdate Method call to execute the DELETE query & returns the No of rows deleted
            int count = st.executeUpdate(string: query);
            //System.out.println(query);

            //Data Validations
            if (count > 0)
            {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Selected Employee's Details Deleted Successfully.", title:"Success", messageType: JOptionPane.INFORMATION_MESSAGE);

                // Refresh the EmployeeDetailsTable after deleting
                refreshEmployeeTable();
            }

            //Deselect previous selected rows
            EmployeeDetailsTable.clearSelection();
            refreshEmployeeTable();
        }
        catch (SQLException ex)
        {
            JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while Deleting Tuple: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        }
        catch (Exception e)
        {

        }
    }
}
```

Figure 2.136: Code for Deleting a Registered Employee

This method is used to eliminate an employee's information from the database.

❖ Searching the Details of a Registered Employee:

```
private void SearchBtnActionPerformed(java.awt.event.ActionEvent evt) {
    String searchText = searchTxt.getText().trim();

    // Check if the search text is not empty
    if (!searchText.isEmpty())
    {
        try
        {
            Statement st = DBconnection.createDBconnection().createStatement();
            ResultSet rs = st.executeQuery("SELECT * FROM employee WHERE (emp_ID LIKE '%" + searchText + "%') || (f_name LIKE '%" + searchText + "%') || (gender LIKE '%" + searchText + "%')|| "
                    + "(NIC LIKE '%" + searchText + "%')");

            // Create a table model to hold the search results
            DefaultTableModel model = new DefaultTableModel();
            model.addColumn(columnName:"Employee ID");
            model.addColumn(columnName:"First Name");
            model.addColumn(columnName:"Last Name");
            model.addColumn(columnName:"Date Of Birth");
            model.addColumn(columnName:"Gender");
            model.addColumn(columnName:"NIC");
            model.addColumn(columnName:"Email");
            model.addColumn(columnName:"Job Role");
            model.addColumn(columnName:"Start Date");
            model.addColumn(columnName:"Daily Rate");
            model.addColumn(columnName:"Contact No");

             // Add the search results to the table model
            while (rs.next())
            {
                Object[] row =
                {
                    rs.getInt(string: "emp_ID"),
                    rs.getString(string: "f_name"),
                    rs.getString(string: "l_name"),
                    rs.getString(string: "dob"),
                    rs.getString(string: "email"),
                    rs.getString(string: "gender"),
                    rs.getString(string: "NIC"),
                    rs.getString(string: "email"),
                    rs.getString(string: "role"),
                    rs.getString(string: "start_date"),
                    rs.getInt(string: "dailyRate"),
                    rs.getInt(string: "phone_no")
                };
                 model.addRow(rowData: row);
            }

            // Set the table model to the JTable
            EmployeeDetailsTable.setModel(dataModel: model);

        }
        catch (SQLException e)
        {
            JOptionPane.showMessageDialog(parentComponent: null, "Error Occurred: " + e.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
            e.printStackTrace();
        }
    }
    else
    {
        // If search text is empty, refresh the table with all suppliers
        refreshEmployeeTable();
    }
}
```

Figure 2.137: Code for Searching a Registered Employee

This method utilizing the data entered in the search text area, searches for employees. The employee details table is updated when it receives records from the database that corresponds to it.

Major module structures, reusable codes & Development tools used in Employee Management Page's Functionality

1) Major Module Structures

- The code creates a user interface for managing employee details by using Swing components (JFrame, JButton, JTextField, JDateChooser, etc.).
- The EmployeeDetails () constructor initializes the default values, event listeners, and GUI components.
- Adding, updating, deleting, and searching employee records is handled by a variety of button action methods, including attachDemoButtonListener (), SalaryManagementPageBtnActionPerformed (), AddBtnActionPerformed (), UpdateBtnActionPerformed (), DeleteBtnActionPerformed (), and SearchBtnActionPerformed ().

- The code uses Java Database Connectivity (JDBC), to communicate with a database. To add, update, delete, and search employee data from the database, SQL queries are executed. Utility methods like "fillDemoValues ()", "getJobRole ()", "refreshEmployeeTable ()", "getSelectedGender ()" performs important tasks such as populate form fields with sample values, load job roles into the combo box, load the modified employee table with the latest values, retrieve selected gender from the radio buttons etc.

2) Reusable Codes

- For testing or demonstration purposes, any form can be filled with predetermined sample data by using the "fillDemoValues ()" method repeatedly.
- The Job Role combo box can be filled with predefined options by reusing the "getJobRole ()" method.
- Employee can be refreshed by using the reusable "refreshEmployeeTable ()" method whenever the table is modified.
- Other database processes can make use of the same method for running SQL queries and managing the outcomes.

3) Development Tools

- Java Swing
- JDBC: Java Database Connectivity
- MySQL Database
- NetBeans or Eclipse IDE 19
- Simple Date Format
- Button Group

3) - I. Employee Daily Attendance Tracking Page (IT22319142 – WIJESINGHE A.G.T)

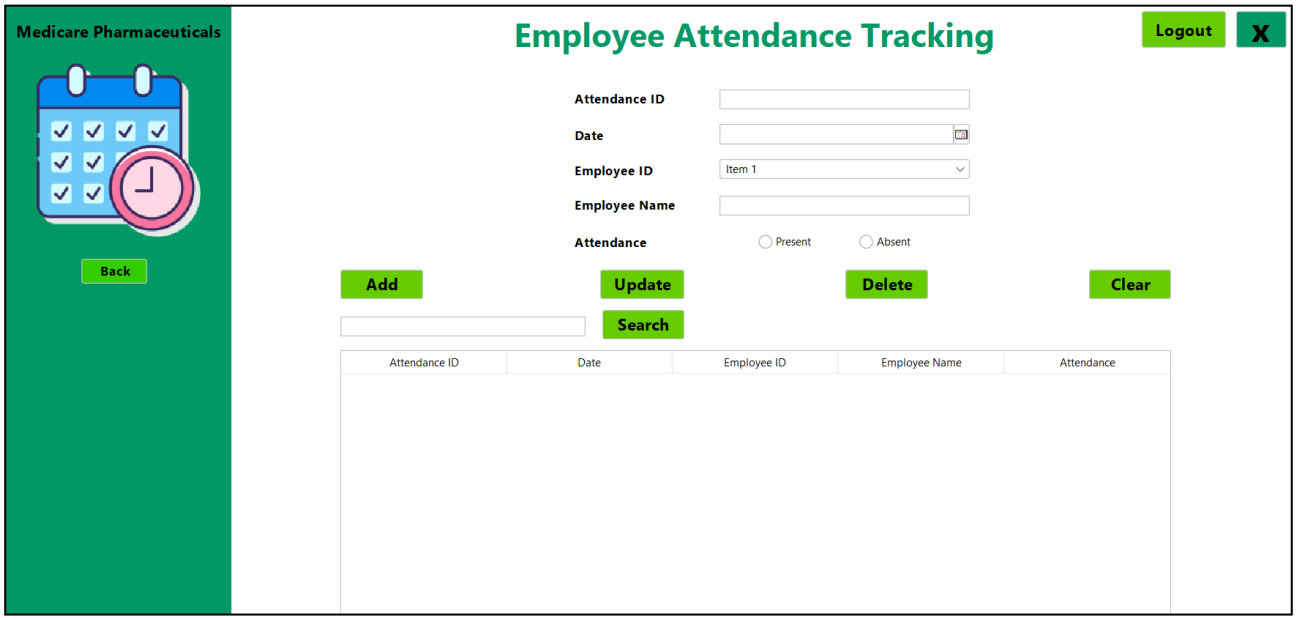

Figure 2.138: Employee Attendance Tracking Page UI

```
public class EmployeeAttendance extends javax.swing.JFrame
{
//Creates new form EmployeeAttendance

    public EmployeeAttendance()
    {
        initComponents();
        refreshAttendanceTable();

        //The method to populate the employeeID combo box
        getEmployeeIDs();

        employeeID.addActionListener(new ActionListener()
        {
            @Override
            public void actionPerformed(ActionEvent e)
            {
                int selectedMedID = Integer.parseInt(s: employeeID.getSelectedItem().toString());
                String employeeName = getEmployeeNameByEmpID(EmployeeID:selectedMedID);
                emp_name.setText(t: employeeName);
            }
        });

        id.setText(t: "Will Be Added Automatically");

        // Set startDate JDateChooser to today's date only
        day.setDate(new Date()); // Set the date to today's date
        day.setMinSelectableDate(new Date()); // Disabling past dates
        day.setMaxSelectableDate(new Date()); // Disabling future dates

        emp_name.setText(t: "");

        // Create a ButtonGroup
        ButtonGroup attendanceGroup = new ButtonGroup();
        attendanceGroup.add(b: present);
        attendanceGroup.add(b: absent);

        // Set Present as the default selection
        present.setSelected(b: true);
        absent.setSelected(b: false);
    }
```

Figure 2.139: Attendance Tracking Page Initial Code

First a class called "EmployeeAttendance" that extends "javax. swing. JFrame" is created. The constructor initializes the GUI elements, fills up the employee ID combo box, and configures an action listener to trigger the combo box's display of the matching employee name with selection of an ID. The attendance form is initialized, the ID field is set to a default message, a date chooser is configured to limit selection to the current day, and a button group for attendance status is established, with "Present" as the default option. This configuration ensures a user-friendly interface for tracking employees' daily attendance.

❖ Load Employee IDs into the Combo Box

```
private void getEmployeeIDs()
{
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery(string: "SELECT emp_ID FROM employee");

        // Clear existing items in the combo box
        employeeID.removeAllItems();

        // Populate combo box with employee IDs
        while (rst.next())
        {
            // Store emp_IDs in the combo box
            String item = String.valueOf(i: rst.getInt(string: "emp_ID"));
            employeeID.addItem(item);
        }

        // Close resources
        rst.close();
        st.close();
        DBconnection.createDBconnection().close();
    }
    catch (SQLException ex)
    {
        // Handle exceptions
        JOptionPane.showMessageDialog(parentComponent: null, "Error fetching Employee IDs: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.140: Employee ID Combo Box Code

The `getEmployeeIDs () ` function establishes a connection with a database to retrieve employee IDs, which are then used to fill the `employeeID} combo box. The function retrieves all employee IDs by running the SQL query `SELECT emp_ID FROM employee} after creating a `Statement` object from a database connection. The method iterates through the result set, transforming each {emp_ID} to a string and adds it to the combo box after first clearing the box of any items that may have been there before. The result set, statement, and database connection are closed at the end of processing. A `JOptionPane} is used to display an error message in the event that a `SQLException` arises during this process.

❖ Retrieve the Name of the Specific Employee ID

```
private String getEmployeeNameByEmpID(int EmployeeID)
{
    String emp_Name = null;
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery("SELECT f_name FROM employee WHERE emp_ID = " + EmployeeID);

        if (rst.next())
        {
            emp_Name = rst.getString(string: "f_name");
        }
        rst.close();
        st.close();
        DBconnection.createDBconnection().close();
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while fetching the employee name: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
    return emp_Name;
}

private String getAttendance()
{
    if (present.isSelected())
    {
        return "1";
    }
    else if (absent.isSelected())
    {
        return "0";
    }
    else //When no option is selected
    {
        return "Did not marked yet";
    }
}
```

Figure 2.141: Employee ID Combo Box Code

The "getEmployeeNameByEmpID ()" method handles any SQL exceptions that could occur and, if necessary, displays an error message. It performs this by connecting to the SQL pharmacy database and retrieving an employee's first name based on the employee ID that is provided. It executes a SQL query using a Statement object, retrieves the result, and then closes the connections. A string representing the attendance status is returned by the "getAttendance ()" method: "1" for present, "0" for absent, and "Did not marked yet" if no choice is chosen.

❖ Get the Attendance Status (Absent/Present) from the Radio Buttons

```
private String getAttendance()
{
    if (present.isSelected())
    {
        return "1";
    }
    else if (absent.isSelected())
    {
        return "0";
    }
    else //When no option is selected
    {
        return "Did not marked yet";
    }
}
```

Figure 2.142: Get the Marked Attendance

❖ Add a New Attendance Entry for the Daily Attendance Table

```
private void addBtnActionPerformed(java.awt.event.ActionEvent evt) {
    try
    {
        int selectedEmployeeID = Integer.parseInt(s: employeeID.getSelectedItem().toString());
        String attendanceStatus = getAttendance();

        if (isAttendanceMarkedToday(employeeID:selectedEmployeeID))
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "This Employee's today's attendance has already been marked!", title:"Duplicate Attendance", me
            return; // Stop execution if attendance is already marked
        }

        SimpleDateFormat sdf = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        String todayDate = sdf.format(date: day.getDate());

        String empName = getEmployeeNameByEmpID(EmployeeID:selectedEmployeeID);

        Statement st = DBconnection.createDBconnection().createStatement();

        // Fetch the next available attendanceID from the database
        ResultSet rs = st.executeQuery(string: "SELECT MAX(attendanceID) FROM attendance");
        int nextAttendanceID = 1;

        if (rs.next())
        {
            nextAttendanceID = rs.getInt(i: 1) + 1;
        }

        rs.close();

        // Insert attendance record into the database
        String insertQuery = "INSERT INTO attendance (attendanceID, date, empID, empName, attendance) VALUES (" + nextAttendanceID + ", '" + todayDate + "', "
                + selectedEmployeeID + ", '" + empName + "', '" + attendanceStatus + "')";
        int count = st.executeUpdate(string: insertQuery);

        if (count > 0)
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Attendance Added Successfully", title:"Success", messageType: JOptionPane.INFORMATION_MESSAGE);
            refreshAttendanceTable();
        }

        st.close();
    }
    catch (SQLException e)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error: " + e.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.143: Code for Adding a New Attendance Entry

This "addBtnActionPerformed ()" method is used to add an employee's attendance record and is enabled by clicking the Add button. The chosen employee ID and attendance status are first retrieved. To avoid duplicates, it verifies whether the employee's attendance has already been recorded for the day. If not, it uses the employee's ID to retrieve their name and format the current date. Next, it connects to the database, retrieves the next attendance ID that is available, and adds a new attendance entry to the database. If the insertion is successful, a success message appears and the attendance table is refreshed; if not, an error message is displayed. Any SQL exceptions are detected and shown in a dialog box with error messages.

❖ **Select one of the attendance data entries to update or delete**

```
private int EmployeeIDofTheSelectedRow;

private void attendanceTableMouseClicked(java.awt.event.MouseEvent evt) {
    // Get the index of the selected row
    int selectedRow = attendanceTable.getSelectedRow();

    // Check if a row is selected
    if (selectedRow == -1)
    {
        return;
    }

    // Get the table model
    TableModel model = attendanceTable.getModel();

    // Store the original Employee ID
    EmployeeIDofTheSelectedRow = Integer.parseInt(s: model.getValueAt(rowIndex: selectedRow, columnIndex: 2).toString());

    // Populate the input fields with data from the selected row
    id.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 0).toString()); // attendanceID

    // Parse date from string to Date object
    try
    {
        SimpleDateFormat dateFormat = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        Date selectedDate = dateFormat.parse(source: model.getValueAt(rowIndex: selectedRow, columnIndex: 1).toString());
        day.setDate(date: selectedDate); // date
    }
    catch (ParseException ex)
    {
        // Handle parsing exception
        ex.printStackTrace();
    }

    // Set selected Employee ID in the combo box
    String selectedEmpID = model.getValueAt(rowIndex: selectedRow, columnIndex: 2).toString();
    employeeID.setSelectedItem(anObject: selectedEmpID);

    // Populate employee name based on selected employee ID
    String employeeName = getEmployeeNameByEmpID(EmployeeID: Integer.parseInt(s: selectedEmpID));
    emp_name.setText(t: employeeName);
    // Set the attendance status radio buttons
    String attendance = model.getValueAt(rowIndex: selectedRow, columnIndex: 4).toString();
    if (attendance.equalsIgnoreCase(anotherString: "Present"))
    {
        present.setSelected(b: true);
        absent.setSelected(b: false);
    }
    else if (attendance.equalsIgnoreCase(anotherString: "Absent"))
    {
        present.setSelected(b: false);
        absent.setSelected(b: true);
    }
    else
    {
        present.setSelected(b: false);
        absent.setSelected(b: false);
    }
}
```

Figure 2.144: Code for Selecting a row to update or delete an attendance record

When a user clicks on a row in an attendance table, the "attendanceTableMouseClicked ()" method is called. The method finds the selected row's index and determines whether a row is actually selected. Using the table model, it then retrieves information from the chosen row, including the employee ID, which it saves and uses to fill in a number of input fields. Using a date picker to determine the correct day, the approach adds text fields for the employee's name and attendance ID. It then modifies radio buttons to indicate the employee's attendance status (present or absent). It also manages possible exceptions in parsing for date values.

❖ Updating an Existing Entry in the Attendance Table

```
private void Update_BtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Update Button
    try
    {
        // Get the selected row index
        int selectedRow = attendanceTable.getSelectedRow();

        if (selectedRow == -1)
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Please select a row to update.", title:"No Row Selected", messageType: JOptionPane.WARNING_MESSAGE);
            return;
        }

        // Get the values from the selected row in the table
        String attendanceID = attendanceTable.getValueAt(row: selectedRow, column: 0).toString(); // Assuming the first column is the attendanceID
        String attendanceStatus = getAttendance();

        // Get the date from the JDateChooser
        SimpleDateFormat sdf = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        String selectedDate = sdf.format(date: day.getDate());

        // Update the attendance in the database
        Statement st = DBconnection.createDBconnection().createStatement();
        String updateQuery = "UPDATE attendance SET date = '" + selectedDate + "', empID = " + EmployeeIDofTheSelectedRow + ", attendance = '" + attendanceStatus + "' WHERE attendanceID = " + attendanceID;

        int rowsAffected = st.executeUpdate(string: updateQuery);

        if (rowsAffected > 0)
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Attendance record updated successfully.", title:"Success", messageType: JOptionPane.INFORMATION_MESSAGE);
            refreshAttendanceTable(); // Refresh the attendance table to reflect the changes
        }
        else
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Failed to update attendance record.", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        }

        st.close();
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while updating attendance: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.145: Code for Modifying an Attendance record

This code illustrates the method that updates an attendance record in the database when the "Update" button in the "Daily Attendance Tracking Page" is clicked. Initially, it determines whether a row in the attendance table is selected; if not, it displays an alert. When a row is chosen, a date is retrieved from a "JDateChooser" component, and the attendance ID and status are retrieved from the table. After that, it creates and runs a "SQL UPDATE query" to update the relevant database record with the updated date, employee ID, and attendance status. The attendance table is updated, and a success message appears if the modification is successful; if not, an error message displays. When a SQL exception occurs, the method generates an error message.

❖ Deleting an Existing Entry in the Attendance Table

```
private void Delete_BtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Delete Button
    int confirm = JOptionPane.showConfirmDialog(parentComponent: null, message: "Are you sure you want to 'Delete' this Record?", title:"Confirm Deletion", optionType:JOptionPane.YES_NO_OPTION);
    if (confirm == JOptionPane.YES_OPTION)
    {
        try
        {
            Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
            String query = "DELETE FROM `attendance` WHERE `attendanceID` = " + Integer.valueOf(s: attendanceTable.getValueAt(row: attendanceTable.getSelectedRow(), column: 0).toString());

            //executeUpdate Method call to execute the DELETE query & returns the No of rows deleted
            int count = st.executeUpdate(string: query);
            //System.out.println(query);

            //Data Validations
            if (count > 0)
            {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Selected Attendance Details Deleted Successfully.", title:"Success", messageType: JOptionPane.INFORMATION_MESSAGE);

                // Refresh the EmployeeDetailsTable after deleting
                refreshAttendanceTable();
            }

            //Deselect previous selected rows
            attendanceTable.clearSelection();
            refreshAttendanceTable();
        }
        catch (SQLException ex)
        {
            JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while Deleting Tuple: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        }
        catch (Exception e)
        {

        }
    }
}
```

Figure 2.146: Code for Deleting an Attendance record from the Database

The "Delete" button action is handled by the method defined by the above code. A confirmation message asking the user to confirm whether they wish to delete the selected record appears when the button is clicked. The method tries to remove the chosen record from the attendance table in the database if the user approves. Using the ID of the chosen row from the "JTable" (attendanceTable), it creates a "DELETE SQL query". The user is informed that the deletion was successful when the query is run. After that, any selected rows are deselected, and the table is updated to reflect the modifications. An error message is shown in the event that a SQL exception arises while the deletion is being performed.

- **Search for an Existing Attendance Record from the Employee's Name/Date/Employee's ID**

```
private void search_ButtonActionPerformed(java.awt.event.ActionEvent evt) {
    String searchText = searchTxt.getText().trim();

    // Check if the search text is not empty
    if (!searchText.isEmpty())
    {
        try
        {
            Statement st = DBconnection.createDBconnection().createStatement();
            ResultSet rs = st.executeQuery("SELECT * FROM attendance WHERE (empName LIKE '%" + searchText + "%') || (empID LIKE '%" + searchText + "%') || (date LIKE '%" + searchText + "%')");

            // Create a table model to hold the search results
            DefaultTableModel model = new DefaultTableModel();
            model.addColumn(columnName:"Attendance ID");
            model.addColumn(columnName:"Date");
            model.addColumn(columnName:"Employee ID");
            model.addColumn(columnName:"Employee Name");
            model.addColumn(columnName:"Attendance");

             // Add the search results to the table model
            while (rs.next())
            {
                Object[] row =
                {
                    rs.getInt(string: "attendanceID"),
                    rs.getString(string: "date"),
                    rs.getString(string: "empID"),
                    rs.getString(string: "empName"),
                    rs.getString(string: "attendance")
                };
                model.addRow(rowData: row);
            }

            // Set the table model to the JTable
            attendanceTable.setModel(dataModel: model);

        }
    else
    {
        // If search text is empty, refresh the table with all suppliers
        refreshAttendanceTable();
    }
}
```

Figure 2.147: Code for Searching an Attendance record from the Database

This code specifies what happens when the search button is clicked. After removing any additional spaces while making sure the content is not empty, it retrieves the text from a search field. If the search text appears, it uses a "Statement object" made from a database connection to query a database for attendance records where the employee's name, employee ID, or date matches the search text. The search results are then displayed by populating a table model (DefaultTableModel) with the results and setting it to a "JTable" (attendanceTable). An error message is displayed to the user in a situation where a SQL exception occurs during this operation. When the search text is empty, the table is refreshed to display every attendance. It invokes "refreshAttendanceTable ()" to update the table and display all attendance records if the search bar is empty.

Major module structures, reusable codes & Development tools used in Employee Attendance Tracking Page's Functionality

1)Major Module Structures

- The EmployeeAttendance Constructor initializes event listeners and GUI elements. Employee IDs are presented in a combo box, and default values are set for other UI elements.
- Database Interactions like filling a combo box with employee IDs that are retrieved from the employee table, Checking if an employee's attendance has been recorded for the day, displaying the most recent attendance data from the attendance table in the GUI table, altering a database entry for an already-existing attendance record, establishing a fresh attendance entry in the database, eliminating a record of attendance from the database & filtering attendance records using search criteria.
- Event handling functions like Button Actions (manages different button clicks) and Combo Box Actions (updates employee name based on the specified employee ID)

2)Reusable Codes

- To ensure consistency and reusability, Database Connection code repeatedly creates database connections using the DBconnection.createDBconnection() method.
- Common Error Handling (JOptionPane is used to display error messages for database-related issues.)
- Practicality The code is modular and reusable since methods like "isAttendanceMarkedToday ()" and "getEmployeeNameByEmpID ()" encapsulate particular functionalities.

3) Development Tools Used

- Java Swing
- JDBC (Java Database Connectivity)
- SimpleDateFormat

3) - II. Employee Monthly Salary Calculation Page (IT22319142 – WIJESINGHE A.G.T)

Figure 2.148: Employee Salary Calculation Page UI

First a graphical user interface is created for managing employee monthly salaries by defining a Java Swing application class called "EmployeeSalary", which extends "JFrame". It initializes the form's elements and configures the required GUI components, such as "text fields" and "a combo box" for employee IDs. The introduction of the flag "ignoreMonthChangeEvent ()" stops unbounded recursion. The "EmployeeSalary () constructor" initializes the form, uses the "getEmployeeIDs () function" to fill the employee ID combo box, and configures an action listener for the employee ID combo box (employID). The listener retrieves and displays the name and daily rate of the related employee in the designated fields when an employee ID is selected. To ensure that the GUI is properly initialized, several fields' initial values are set to their default settings.

```
public class EmployeeSalary extends javax.swing.JFrame
{
// Creates new form EmployeeSalary

    //Flag to prevent infinite recursion
    private boolean ignoreMonthChangeEvent = false;

    public EmployeeSalary()
    {
        initComponents();
        refreshSalaryTable();

        //The method to populate the employeeID combo box
        getEmployeeIDs();

        employID.addActionListener(new ActionListener()
        {
            @Override
            public void actionPerformed(ActionEvent e)
            {
                try
                {
                    // Get the selected employee ID from the combo box
                    int selectedEmpID = Integer.parseInt(s: employID.getSelectedItem().toString());

                    // Retrieve the employee's name and daily rate based on the selected employee ID
                    String employee_Name = getEmployeeNameByEmployeeID(employeeID:selectedEmpID);
                    int employeeDailyRate = getEmployeeDailyRateByEmployeeID(employeeID:selectedEmpID);

                    // Display the employee's name and daily rate in respective text fields or labels
                    employeeName.setText(s: employee_Name);
                    daily_rate.setText(s: String.valueOf(i: employeeDailyRate));
                }
                catch (NumberFormatException ex)
                {
                    // Handle number format exception if selectedEmpID is invalid
                    ex.printStackTrace();
                }
            }
        });

    id.setText(s: "Will Be Added Automatically");
    employeeName.setText(s: "");
    daily_rate.setText(s: "0");
    Year.setText(s: "");
    net_salary.setText(s: "");
    Attendance.setText(s: "0");
}
```

Figure 2.149: Employee Salary Calculation Page Class Initialization Code

❖ Retrieve the Employee IDs into the combo box from the Employee Table

```
private void getEmployeeIDs()
{
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery(string: "SELECT emp_ID FROM employee");

        // Clear existing items in the combo box
        employID.removeAllItems();

        // Populate combo box with employee IDs
        while (rst.next())
        {
            // Store emp_IDs in the combo box
            String item = String.valueOf(i: rst.getInt(string: "emp_ID"));
            employID.addItem(item);
        }

        // Close resources
        rst.close();
        st.close();
        DBconnection.createDBconnection().close();
    }
    catch (SQLException ex)
    {
        // Handle exceptions
        JOptionPane.showMessageDialog(parentComponent: null, "Error fetching Employee IDs: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.150: Code to Load Employee IDs into the Combo Box

❖ Auto Generate the Name of the Selected Employee ID

```
private String getEmployeeNameByEmployeeID(int employeeID)
{
    String employeeName = null;
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rst = st.executeQuery("SELECT f_name FROM employee WHERE emp_ID = " + employeeID);
        if (rst.next())
        {
            employeeName = rst.getString(string: "f_name");
        }
        rst.close();
        st.close();
        DBconnection.createDBconnection().close();
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while fetching the employee name: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
    return employeeName;
}
```

Figure 2.151: Get the Employee Name by the selected Employee ID

Based on the selected employee's ID, the "getEmployeeNameByEmployeeID ()" method obtains the employee's first name from the employee table. In order to retrieve the first name (f_name) from the employee table where the "emp_ID" matches the selected employeeID, it first establishes a database connection, prepares a statement, and runs a SQL query. The first name is extracted out and assigned to the "employeeName variable" if a matched entry is discovered. Before returning the retrieved employee name, the method ensures that database resources are correctly terminated. It handles SQL exceptions by using "JOptionPane" to display an error message.

❖ Retrieve the Employee's Daily rate

```
private int getEmployeeDailyRateByEmployeeID(int employeeID)
{
    int dailyRate = 0;

    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery("SELECT dailyRate FROM employee WHERE emp_ID = " + employeeID);

        if (rs.next())
        {
            dailyRate = rs.getInt(string: "dailyRate");
        }

        rs.close();
        st.close();
        DBconnection.createDBconnection().close();
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while fetching the employee's daily rate: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
    return dailyRate;
}
```

Figure 2.152: Get the Daily Rate by the selected Employee ID

Using the employee ID, the `getEmployeeDailyRateByEmployeeID` method obtains an employee's daily rate from a database. After setting the daily rate to zero, it makes a database query to find the employee's daily rate. The daily rate is extracted from the result set if a matching record is found. Ultimately, the obtained daily rate is returned.

❖ Retrieve the Employee's Daily rate

```
private void AddBtnActionPerformed(java.awt.event.ActionEvent evt) {
    try
    {
        String employee_Name;
        int employee_ID, AttendanceCount, SelectedYear, selectedMonth, Daily_Rate, Net_Salary;

        // Validate year input
        if (Year.getText().isEmpty())
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Year cannot be empty", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Fetch other necessary inputs
        employee_ID = Integer.parseInt(s: employID.getSelectedItem().toString());
        Daily_Rate = Integer.parseInt(s: daily_rate.getText());
        SelectedYear = Integer.parseInt(s: Year.getText());

        // Get the selected month from JMonthChooser
        selectedMonth = month.getMonth() + 1; // Month is 0-based

        // Checking if salary details already exist for the selected month and year
        if (checkWhetherSalaryDetailsAlreadyExist(employeeID:employee_ID, month:selectedMonth, year: SelectedYear))
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "For this Employee, Salary Details of the selected Month and Year are already added to the database", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
            return;
        }

        //Get Attendance count for the selected month
        AttendanceCount = getAttendanceCountForMonth(employeeID:employee_ID, month:selectedMonth, year: SelectedYear);

        // Calculate the Monthly salary
        Net_Salary = AttendanceCount * Daily_Rate;

        // Get the employee name (if not retrieved already)
        employee_Name = employeeName.getText();

        //Validations of Database Interaction
        try
        {
            // Fetch the next available Salary ID from the Database
            int nextSalaryID = 1;
            Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
            ResultSet rs = st.executeQuery(string: "SELECT MAX(salaryID) FROM salary");

            if (rs.next())
            {
                nextSalaryID = rs.getInt(i: 1) + 1;
            }

            // Prepare and execute SQL insert statement
            String query = "INSERT INTO `salary`(`salaryID`, `empID`, `empName`, `dailyRate`, `year`, `Month`, `attendance`, `netSalary`) VALUES (?, ?, ?, ?, ?, ?, ?, ?)";
            java.sql.PreparedStatement prepSt = st.getConnection().prepareStatement(string: query);
            prepSt.setInt(i: 1, i1:nextSalaryID);
            prepSt.setInt(i: 2, i1:employee_ID);
            prepSt.setString(i: 3, string: employee_Name);
            prepSt.setInt(i: 4, i1:Daily_Rate);
            prepSt.setInt(i: 5, i1:SelectedYear);
            prepSt.setInt(i: 6, i1:selectedMonth);
            prepSt.setInt(i: 7, i1:AttendanceCount);
            prepSt.setInt(i: 8, i1:Net_Salary);

            // Execute update
            int count = prepSt.executeUpdate();
            if (count > 0)
            {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Salary Details Added to the Database Successfully", title:"Success", messageType: JOptionPane.INFORMATION_MESSAGE);
                refreshSalaryTable();
            }

            // Close resources
            prepSt.close();
            rs.close();
            st.close();
        }
        catch (SQLException ex)
        {
            JOptionPane.showMessageDialog(parentComponent: null, "SQL Error...!" + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        }
        catch (NumberFormatException ex)
        {
            JOptionPane.showMessageDialog(parentComponent: null, "Invalid Number Format...!" + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        }
    }
    catch (Exception ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.153: Enter a New Salary Record to the System Database

When the Add button is clicked, the system verifies the year entered, retrieves and parses employee ID, daily rate, and year, and uses a method to see if salary details are already available for the chosen month and year. In the event that no record is obtained, the daily rate and attendance are used to determine the net salary. After interacting with the database to retrieve the next salary ID that becomes available, the code creates and runs a SQL insert statement to add the new salary record. The salary table is refreshed, and a success message is shown if the insertion is successful.

❖ Retrieve the Monthly Attendance from the Attendance Table

```
// Method to get attendance count for a specific month and year
private int getAttendanceCountForMonth(int employeeID, int month, int year)
{
    int attendanceCount = 0;
    try
    {
        // Constructing the SQL query
        String query = "SELECT COUNT(*) FROM attendance WHERE empID = ? AND MONTH(date) = ? AND YEAR(date) = ? AND attendance = '1'";

        // Debug
        System.out.println("Generated SQL query: " + query);

        java.sql.PreparedStatement prepSt = DBconnection.createDBconnection().prepareStatement(string: query);

        prepSt.setInt(i: 1, i1:employeeID);
        prepSt.setInt(i: 2, i1:month);
        prepSt.setInt(i: 3, i1:year);

        //Print the parameters being set
        System.out.println("Employee ID: " + employeeID + ", Month: " + month + ", Year: " + year);

        // Execute the query
        ResultSet rs = prepSt.executeQuery();

        if (rs.next())
        {
            attendanceCount = rs.getInt(i: 1);
        }
        rs.close();
        prepSt.close();
    }
    catch (SQLException ex)
    {
        // Error Handling: Display error message and print stack trace
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while fetching attendance count: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        ex.printStackTrace();
    }
    return attendanceCount;
}
```

Figure 2.154: Code for retrieving the Monthly Attendance

The method "getAttendanceCountForMonth ()" month was developed to obtain an employee's attendance count for a certain month and year from a database. Based on the parameters provided, it builds a SQL query, runs it, and returns the number of attendance records where the employee ID, the month, and the year match and the attendance status is set to '1'. This count is then returned by the method.

❖ Check if salary details already exist for the selected employee, month, and year

```
//Check if salary details already exist for a specific employee, month, and year
private boolean checkWhetherSalaryDetailsAlreadyExist(int employeeID, int month, int year)
{
    boolean exist = false;

    try
    {
        String query = "SELECT COUNT(*) FROM salary WHERE empID = ? AND `Month` = ? AND `year` = ?";

        java.sql.PreparedStatement prepSt = DBconnection.createDBconnection().prepareStatement(string: query);

        prepSt.setInt(i: 1, i1:employeeID);
        prepSt.setInt(i: 2, i1:month);
        prepSt.setInt(i: 3, i1:year);

        ResultSet rs = prepSt.executeQuery();
        if (rs.next())
        {
            int count = rs.getInt(i: 1);
            exist = count > 0;
        }
        rs.close();
        prepSt.close();
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while checking salary details: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
    return exist;
}
```

Figure 2.155: Check for Duplicate Salary Entries

❖ Select a single row from the Salary Table

```
private int EmployeeIDofTheSelectedRow;

private void employee_Salary_TableMouseClicked(java.awt.event.MouseEvent evt) {
    //Get the index of the selected row
    int selectedRow = employee_Salary_Table.getSelectedRow();

    // Check if a row is selected
    if (selectedRow == -1)
    {
        return;
    }

    // Get the table model
    TableModel model = employee_Salary_Table.getModel();

    // Store the original Employee ID
    EmployeeIDofTheSelectedRow = Integer.parseInt(s: model.getValueAt(rowIndex: selectedRow, columnIndex: 1).toString());

    // Populate the input fields with data from the selected row
    id.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 0).toString()); // salaryID
    employeeName.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 2).toString());
    daily_rate.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 3).toString());
    Year.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 4).toString());
    month.setMonth(Integer.parseInt(s: model.getValueAt(rowIndex: selectedRow, columnIndex: 5).toString()) - 1);
    Attendance.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 6).toString());
    net_salary.setText(t: model.getValueAt(rowIndex: selectedRow, columnIndex: 7).toString());

    // Set selected Employee ID in the combo box
    String selectedEmpID = model.getValueAt(rowIndex: selectedRow, columnIndex: 1).toString();
    employID.setSelectedItem(anObject: selectedEmpID);

    // Populate employee name based on selected employee ID
    String employee_Name = getEmployeeNameByEmployeeID(employeeID:Integer.parseInt(s: selectedEmpID));
    employeeName.setText(t: employee_Name);
}
```

Figure 2.156: Selecting a row from the Salary Table

❖ Update data of a row in the Salary Table

```
private void UpdateBtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Update Button
    try
    {
        String Employee_Name;
        int Employee_ID, AttendanceCount, selectedYear, Daily_Rate, Net_Salary;

        Employee_Name = employeeName.getText();
        Employee_ID = Integer.parseInt((String) employID.getSelectedItem());
        Daily_Rate = Integer.parseInt(s: daily_rate.getText());
        selectedYear = Integer.parseInt(s: Year.getText());
        AttendanceCount = getAttendanceCountForMonth(employeeID:Employee_ID, month.getMonth() + 1, year: selectedYear); // Assuming 'month' is a JMonthChooser
        Net_Salary = AttendanceCount * Daily_Rate;

        // Create the SQL query for updating the salary record
        String query = "UPDATE salary SET empID = ?, empName = ?, dailyRate = ?, year = ?, Month = ?, attendance = ?, netSalary = ? WHERE salaryID = ?";

        // Establish database connection and prepare the statement
        java.sql.Connection connection = pharmacy.DBconnection.createDBconnection();
        java.sql.PreparedStatement prepSt = connection.prepareStatement(string: query);

        prepSt.setInt(i: 1, i1:Employee_ID);
        prepSt.setString(i: 2, string: Employee_Name);
        prepSt.setInt(i: 3, i1:Daily_Rate);
        prepSt.setInt(i: 4, i1:selectedYear);
        prepSt.setInt(i: 5, month.getMonth() + 1); // Assuming 'month' is a JMonthChooser
        prepSt.setInt(i: 6, i1:AttendanceCount);
        prepSt.setInt(i: 7, i1:Net_Salary);
        prepSt.setInt(i: 8, i1:Integer.parseInt(s: id.getText()));

        // Execute the update statement
        int rowCount = prepSt.executeUpdate();

        // Check if update was successful
        if (rowCount > 0)
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Salary Details Updated Successfully", title:"Success", messageType: JOptionPane.INFORMATION_MESSAGE);
            refreshSalaryTable();
        }
        else
        {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Failed to update Salary Details", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        }

        // Close the connection and prepared statement
        prepSt.close();
        connection.close();
    }
    catch (NumberFormatException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Invalid Input for Employee ID, Year, or Daily Rate", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while updating Salary Details: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.157: Update details of a row in the Salary Table

When the Update button is clicked, text fields and combo box containing the employee's name, ID, daily rate, and chosen year are retrieved. After that, it changes the relevant database record with the updated salary information and calculates the net salary using the attendance count that was obtained from the "getAttendanceCountForMonth ()" method call. If the change is successful, it refreshes the salary table and displays a success message; if not, it displays an error message.

❖ Delete a Data Tuple from the Salary Table

```
private void DeleteBtnActionPerformed(java.awt.event.ActionEvent evt) {
    //Delete Button
    int confirm = JOptionPane.showConfirmDialog(parentComponent: null, message: "Are you sure you want to 'Delete' this Record?", title: "Confirm Deletion", optionType: JOptionPane.YES_NO_OPTION);
    if (confirm == JOptionPane.YES_OPTION)
    {
        try
        {
            Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
            String query = "DELETE FROM `salary` WHERE `salaryID` = " + Integer.valueOf(s: employee_Salary_Table.getValueAt(row: employee_Salary_Table.getSelectedRow(), column: 0).toString());

            //executeUpdate Method call to execute the DELETE query & returns the No of rows deleted
            int count = st.executeUpdate(string: query);
            //System.out.println(query);

            //Data Validations
            if (count > 0)
            {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Selected Employee's Salary Details Deleted Successfully.", title: "Success", messageType: JOptionPane.INFORMATION_MESSAGE);

                // Refresh the EmployeeDetailsTable after deleting
                refreshSalaryTable();
            }

            //Deselect previous selected rows
            employee_Salary_Table.clearSelection();
            refreshSalaryTable();
        }
        catch (SQLException ex)
        {
            JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while Deleting Tuple: " + ex.getMessage(), title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
        }
        catch (Exception e)
        {

        }
    }
}
```

Figure 2.158: Delete a row from the Salary Table

This code first asks the user to confirm whether they really want to delete a record by displaying a confirmation box. If the user provides confirmation, it gets the ID of the chosen record from a table, creates a “SQL DELETE query” that specifically targets that record, and runs it. If the deletion is successful, a success message appears, the table with the employee's salary details is refreshed, and the selection on the table is cleared. An error message is displayed if a problem arises during the process.

❖ Search for the Salary Details of a Specific Employee

```
private void searchButtonActionPerformed(java.awt.event.ActionEvent evt) {
    String searchText = searchTxt.getText().trim();

    // Check if the search text is not empty
    if (!searchText.isEmpty())
    {
        try
        {
            Statement st = DBconnection.createDBconnection().createStatement();
            ResultSet rs = st.executeQuery("SELECT * FROM salary WHERE empName LIKE '%" + searchText + "%'");

            // Create a table model to hold the search results
            DefaultTableModel model = new DefaultTableModel();
            model.addColumn(columnName:"Salary ID");
            model.addColumn(columnName:"Employee ID");
            model.addColumn(columnName:"Employee Name");
            model.addColumn(columnName:"Daily Rate");
            model.addColumn(columnName:"Year");
            model.addColumn(columnName:"Month");
            model.addColumn(columnName:"Attendance");
            model.addColumn(columnName:"Net Salary");

             // Add the search results to the table model
            while (rs.next())
            {
                Object[] row =
                {
                    rs.getInt(string: "salaryID"),
                    rs.getInt(string: "empID"),
                    rs.getString(string: "empName"),
                    rs.getInt(string: "dailyRate"),
                    rs.getInt(string: "year"),
                    rs.getInt(string: "Month"),
                    rs.getInt(string: "attendance"),
                    rs.getInt(string: "netSalary")
                };
                model.addRow(rowData: row);
            }
                                                                                                    _MESSAGE);

            // Set the table model to the JTable
            employee_Salary_Table.setModel(dataModel: model);

        }
        catch (SQLException e)
    }
    else
    {
        refreshSalaryTable();
    }
}
```

Figure 2.159: Search salary details in the Salary Table

7) Billing Details Management - (IT22310996 – THENNAKOON T.A.C.S)

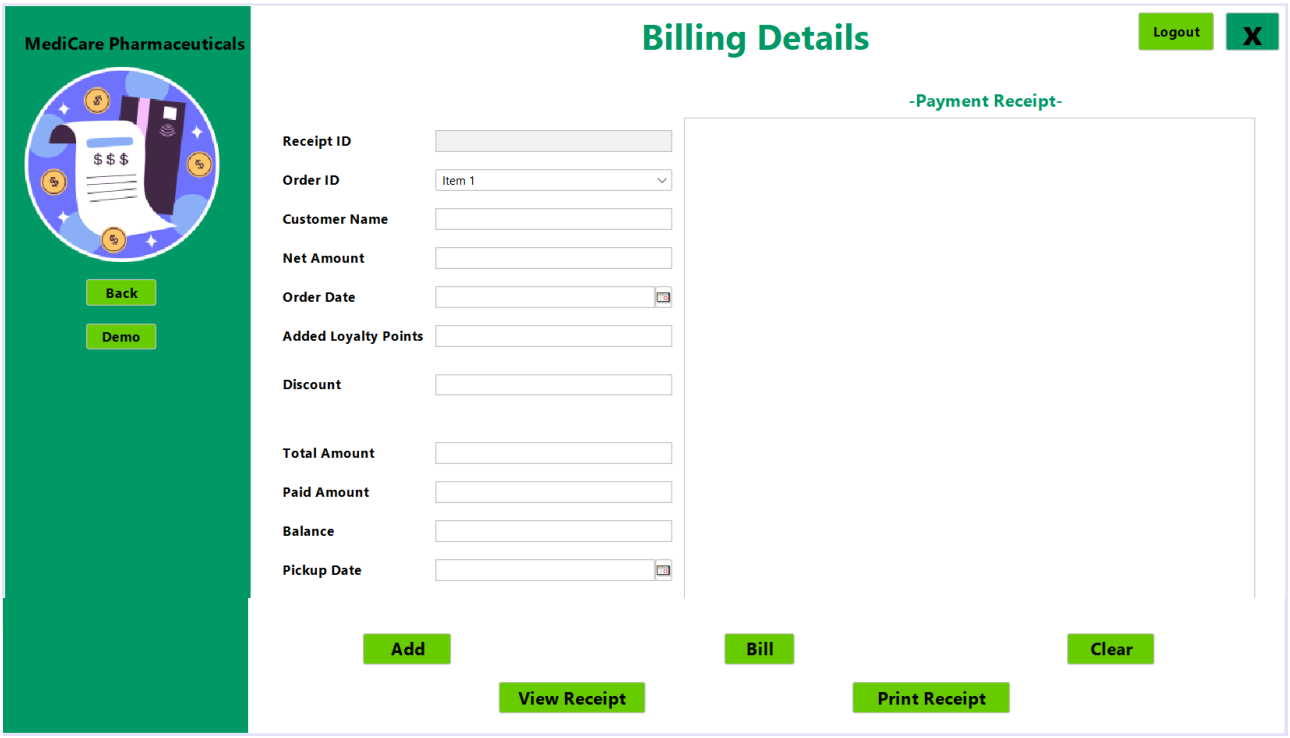

Figure 2.160: Billing Page UI

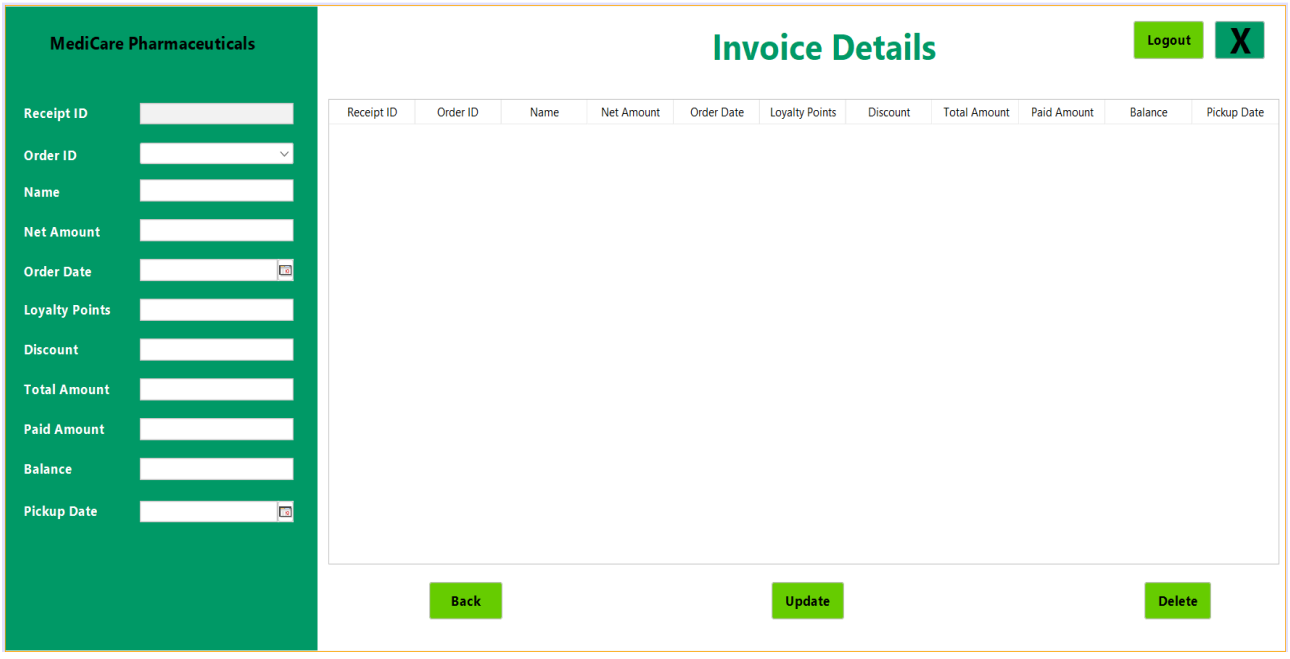

Figure 2.161: Invoice UI

❖ Class Definition and Initialization

```
public class BillingDetails extends javax.swing.JFrame {

    private int totalAmount = 0;
    private int discount = 0;
    private int balanceValue = 0;

     // HashMap to store the mapping between Order IDs and Customer IDs
     private final HashMap<String, Integer> orderCustomerMap = new HashMap<>();
     private int paidAmount;


     public BillingDetails() {
          initComponents();
          getOrderIDs();

          custName.setText(t: "");
          orderdate.setDate(date:Date.from(instant:Instant.now()));
          netamount.setText(t: "0");
          AddLoyal.setText(t: "0");
         // totalLoyal.setText("0");

          totamount.setText(t: "0");
          pamount.setText(t: "0");
          pickupdate.setDate(date:Date.from(instant:Instant.now()));
          balance.setText(t: "0");
     }
```

Figure 2.162:class definitions of billing management

Billing Details is a graphical user interface (GUI) window that is defined as a subclass of "JFrame." The class has several properties and methods for managing database interactions and billing logic.

❖ Fetching net amount by order ID

```
    private String getCustomerNameByID(int custID) {
    try {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery("SELECT f_name, l_name FROM doctor WHERE cust_ID = " + custID);

        if (rs.next()) {
            String customerName = rs.getString(string: "f_name") + " " + rs.getString(string: "l_name");
            return customerName;
        }

    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(parentComponent:null, "Error fetching customer name: " + ex.getMessage(),
                title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
    }
    return null; // Return null if no customer name found
}


    private int getNetAmountByID(int OrderID) {
    try {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery("SELECT netamount FROM ordersnew WHERE OID = " + OrderID);

        if (rs.next()) {
            int netAmnt = rs.getInt(string: "netamount");
            return netAmnt;
        }

    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(parentComponent:null, "Error fetching net amount: " + ex.getMessage(),
                title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
    }
    return -99; // Return null if no customer name found
}
```

Figure 2.163: fetching net amount of billing management

These two methods retrieve the customer's name for a selected order ID and the net amount for a selected order ID from the "ordersnew table."

❖ Filling Order IDs

```
    // Method to populate Order IDs combo box
private void getOrderIDs() {
    try {
        try (Statement st = DBconnection.createDBconnection().createStatement();
             ResultSet rst = st.executeQuery(string: "SELECT OID, CID FROM ordersnew")) {
            // Clear existing items in the combo box
            OrdID.removeAllItems();

            // Populate combo box with Order IDs
            while (rst.next()) {
                // Store order IDs in the combo box
                int orderID = rst.getInt(string: "OID");
                OrdID.addItem(item: String.valueOf(i: orderID));

                // Retrieve and display customer name for each order ID
                int custID = rst.getInt(string: "CID");
                String customerName = getCustomerNameByID(custID);
                String totAmount = String.valueOf(i: getNetAmountByID(OrderID: orderID)); // Getting net amount for the order
                // Set customer name in the text field
                custName.setText(t: customerName);
                netamount.setText(s: totAmount);
                // Set net amount in the corresponding field
                totamount.setText(s: totAmount);
            }

        }

        // Close resources
        DBconnection.createDBconnection().close();
    } catch (SQLException ex) {
        // Handle exceptions
        JOptionPane.showMessageDialog(parentComponent: null, "Error fetching Order IDs: " + ex.getMessage(),
                title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }

    // Add action listener to the combo box to dynamically fetch and display customer name
    OrdID.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            // Get the selected order ID from the combo box
            int selectedOrderID = Integer.parseInt(s: OrdID.getSelectedItem().toString());

          //  totamount.setText(netAmount);

            fetchOrderDetails(selectedOrderID);
        }
    });

}
```

Figure 2.164: filling orderID of billing management

The getOrderIDs function creates an action listener to dynamically retrieve order data and fills the OrdID combo box with order IDs from the ordersnew table.

❖ **Apply the discount and Total amount calculation.**

```
private void applyDiscount(int addedLoyalty) {
    int discount = 0;
    if (addedLoyalty > 30) {
        // Apply 7% discount
        discount = (int) (Integer.parseInt(s: netamount.getText()) * 0.07);
    }
    // Set discount in the corresponding field
    disc.setText(t: String.valueOf(i: discount));

}
// Method to calculate total amount
private int calculateTotalAmount() {
      String inputString = netamount.getText().trim();
    if (inputString.isEmpty()) {
        // Handle empty input
        return 0; // or throw an exception, depending on your requirements
    }
    try {
        int totalAmount = Integer.parseInt(s: inputString);
        // Perform any additional calculations if needed
        return totalAmount;
    } catch (NumberFormatException ex) {
        // Handle parsing errors
        JOptionPane.showMessageDialog(parentComponent:null, "Error parsing total amount: " + ex.getMessage(),
                title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
        return 0; // or throw an exception, depending on your requirements
    }
}
```

Figure 2.165: Apply discount & total amount of billing management

This method calculates the total amount and adds a discount based on added loyalty points which is calculated by net amount.Rs.1000.00 = 1 loyalty point, 15 loyalty points > = 7% discount for each bill (the bill total amount must be more than Rs. 15000.00)

❖ **Enter key pressed listen for calculate net amount.**

```
    private void enterKeyPressed(java.awt.event.KeyEvent evt) {
    // If the Enter key is pressed while in the netamount text field,
    // trigger the calculation and update process
    if (evt.getKeyCode() == KeyEvent.VK_ENTER) {
        netamountActionPerformed(evt:null);
    }
}
```

Figure 2.166: Enter key pressed of billing management

When in the netamount text box, another method called enterKeyPressed listens for the pressing of the Enter key. When the Enter key is hit, the netamountActionPerformed method is called.

❖ Add button and Validations.

```
private void addBActionPerformed(java.awt.event.ActionEvent evt) {

    // TODO add your handling code here:
    //Add Button
    String custname, orderDate, pickupDate;
    int orderID, addedLoyalty = 0, netAmount = 0, Discount = 0, totalAmount = 0, paidAmount = 0, Balance = 0;

    orderID = Integer.parseInt(s: OrdID.getSelectedItem().toString());

    // Get the associated customer ID


    try {
        addedLoyalty = Integer.parseInt(s: AddLoyal.getText());
        netAmount = Integer.parseInt(s: netamount.getText());
        totalAmount = Integer.parseInt(s: totamount.getText());
        paidAmount = Integer.parseInt(s: pamount.getText());
        //Balance = Integer.parseInt(balance.getText());
    } catch (NumberFormatException e) {
        JOptionPane.showMessageDialog(parentComponent:null, message:"Invalid Amount input. Please enter valid numbers.",
                title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
        return;
    }
```

```
        custname = custName.getText();

    if (addedLoyalty > 15) {
        // Apply 7% discount
        Discount = (int) (netAmount * 0.07);
    }

        totalAmount = netAmount - Discount;
        addedLoyalty = netAmount / 1000;
        Balance = totalAmount - paidAmount;



        //Validating the date formatting
        SimpleDateFormat sdf = new SimpleDateFormat(pattern:"yyyy-MM-dd");
        orderDate = sdf.format(date:orderdate.getDate());
        pickupDate = sdf.format(date:pickupdate.getDate());
        Date currentDate = new Date(); // Current date
        Date selectedDate = pickupdate.getDate(); // Selected date

        if (!pickupDate.equals(anObject:sdf.format(new Date()))) {
            JOptionPane.showMessageDialog(parentComponent:null, message:"Pickup date should be today's date.",
                    title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
            return; // Exit the method if pickup date is not today's date
        }
```

```
    //Validations of Database Interaction
    try {
        Statement st = pharmacy.DBconnection.createDBconnection().createStatement();

        //Fetch the next avaialable Order ID from the Database
        ResultSet rs = st.executeQuery(string: "SELECT MAX(receiptID) FROM billing");
        int nextreceiptID = 1;

        if (rs.next()) {
            nextreceiptID = rs.getInt(i: 1) + 1;
        }

        int count = st.executeUpdate("INSERT INTO billing"
                + " (receiptID, OrderID, f_name, Netamount, orderdate, addLoyal, discount, totAmount, paidAmount, balance, pickupdate)"
                + " VALUES ('" + nextreceiptID + "','" + orderID + "','" + custname + "','" + netAmount + "','" + orderDate + "','" +
                addedLoyalty + "','" + Discount + "','" + totalAmount + "','" + paidAmount + "','" + Balance + "','" + pickupDate + "')");

        int option = JOptionPane.showConfirmDialog(parentComponent:null, message:"Are you sure you want to add billing data?",
                title: "Confirmation", optionType: JOptionPane.YES_NO_OPTION);
if (option == JOptionPane.YES_OPTION) {
```

```
                JOptionPane.showMessageDialog(parentComponent:null, message:"Billing Details Added Successfully",
                        title: "Success", messageType:JOptionPane.INFORMATION_MESSAGE);
                // Refresh the table after adding the billing details
                //refreshBillingTable();
            } else {
            JOptionPane.showMessageDialog(parentComponent:null, message:"Failed to add billing details",
                    title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
      //   return;
    }
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(parentComponent:null, message:"Invalid Input for Billing Details...!",
                    title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(parentComponent:null, "Failed to add billing details. Exception: " + e.getMessage(),
                    title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
}
```

Figure 2.167: Add button of billing management

When the "Add" button is pressed, an event handler method called addBActionPerformed is invoked. The logic for inserting a new billing record into the database is managed by this function.

The procedure sets up several variables needed for the billing information. The AddLoyal, netamount, totamount, and pamount input fields' values are attempted to be parsed by the method. It stops the execution and displays an error message if any value is not a valid number. The procedure formats the date of order and the date of pickup, and it verifies that the pickup date is the present day. If it is not, then pop an error message.

The new billing record is inserted using a SQL connection made by the technique. After retrieving the subsequent receiptID that becomes available, it adds the updated billing information to the billing table. Before incorporating the billing information, the procedure displays a confirmation window. It displays a success message if the user confirms; if not, an error message is displayed.

- **Print Bill**

```
private void printRActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
try {
    bill.print();
}

catch (Exception e) {

        }
}
```

Figure 2.168: print bill of billing management

When a Java Swing application's "Print" button is pushed, an event handler known as printRActionPerformed is triggered. This function tries to use the print () method to print a bill.

❖ View Button for the bill

```
    private void viewRActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        // Query to fetch the most recent billing details from the database
        String query = "SELECT * FROM billing ORDER BY receiptID DESC LIMIT 1";
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery(string: query);

        if (rs.next()) {
            // Extract data from the result set
            int receiptID = rs.getInt(string: "receiptID");
            String customerName = rs.getString(string: "f_name");
            int netAmount = rs.getInt(string: "Netamount");
            //int totalLoyaltyPoints = rs.getInt("totLoyal");
            int addedLoyaltyPoints = rs.getInt(string: "addLoyal");
            int totalAmount = rs.getInt(string: "totAmount");
            int discount = rs.getInt(string: "discount");
            int paidAmount = rs.getInt(string: "paidAmount");
            int Balance = rs.getInt(string: "balance");
            Date orderDate = rs.getDate(string: "orderdate");
            Date pickupDate = rs.getDate(string: "pickupdate");


            // Display the billing details in the text area
            bill.setText("                                .......MediCare Pharmaceuticals.......\n\n"
                    + "\n\n ReceiptID \t\t " + receiptID + "\n\n Customer Name\t: " + customerName
                    + "\n\n NetAmount\t\t: Rs. " + netAmount +"\n\n AddedLoyaltyPoints\t: " + addedLoyaltyPoints
                    + "\n\n OrderDate\t\t: " + orderDate + "\n\n Discount\t\t: " + discount +
                    "\n\n TotalAmount\t\t: Rs. " + totalAmount +"\n\n PaidAmount\t\t: Rs. " + paidAmount
                    + "\n\n Balance\t\t: Rs. " + Balance + "\n\n PickupDate\t\t: " + pickupDate + "\n");
        } else {
            JOptionPane.showMessageDialog(parentComponent:null, message:"No billing details found.",
                    title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
        }
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(parentComponent:null, "Error fetching billing details: " + ex.getMessage(),
                title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
    }
```

Figure 2.169: view button of billing management

When a "View" button is hit, an event handler called `viewRActionPerformed` is triggered, displaying the most recent billing information. The first thing it does is open a database connection and run a SQL query to retrieve the most recent billing record from the {billing} table. To guarantee that the most recent entry is retrieved, the query restricts the result to a single record and arranges the entries by {receiptID} in descending order. When the method is successfully retrieved, it pulls several billing details, including the order date, the balance, the customer's name, the net amount, the total amount, the discount, the paid amount, the balance, and the receipt ID. After formatting, these facts are shown in a text field ({bill}). A dialog box notifies the user of the relevant error message if no records are found or if there is a database operation error.

❖ Bill button Action Performed

```
private void billBtnActionPerformed(java.awt.event.ActionEvent evt) {

    billReceipt obj= new billReceipt();
    obj.show();
    this.dispose();

    // Get the values from the fields
int receiptID = Integer.parseInt(s: rID.getText());
int orderID = Integer.parseInt(s: OrdID.getSelectedItem().toString());
String custname = custName.getText();
int addedLoyalty = Integer.parseInt(s: AddLoyal.getText());
// int totalLoyalty = Integer.parseInt(totalLoyal.getText());
int netAmount = Integer.parseInt(s: netamount.getText());
int discount = Integer.parseInt(s: disc.getText());
int totalAmount = Integer.parseInt(s: totamount.getText());
int paidAmount = Integer.parseInt(s: pamount.getText());
int Balance = Integer.parseInt(s: balance.getText());
String orderDate = new SimpleDateFormat(pattern:"yyyy-MM-dd").format(date: orderdate.getDate());
String pickupDate = new SimpleDateFormat(pattern:"yyyy-MM-dd").format(date: pickupdate.getDate());

// Validation for pickup date
if (!pickupDate.equals(anObject: new SimpleDateFormat(pattern:"yyyy-MM-dd").format(new Date()))) {
    JOptionPane.showMessageDialog(parentComponent:null, message:"Pickup date should be today's date.",
            title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
    return;
}
```

Figure 2.170: bill button of billing management

When a "Bill" button is hit, an event handler known as billBtnActionPerformed is triggered. This function ends the current window after creating and displaying a new billReceipt object. The process involves the extraction of values from many input fields, including order ID, balance, order date, order ID, net amount, discount, customer name, additional loyalty points, total amount, paid amount, and balance.These values are retrieved and transformed to the correct data types to ensure they are ready for processing in the future. Additionally, it includes a validation step that verifies whether the pickup date and the current date match. If they do not, an error notice is shown. This method successfully prepares and verifies the data required for billing procedures before transitioning to the display of the billing receipt.

❖ Calculation of net amount

```
private void netamountActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
try {
    int netAmount = Integer.parseInt(s: netamount.getText());

    // Calculate added loyalty points
    int addedLoyalty = netAmount / 1000; // 1 loyalty point for every 1000 rupees

    // Update added loyalty points field
    AddLoyal.setText(t: String.valueOf(i: addedLoyalty));

     calculateFieldsAndUpdate();

    // Apply discount if applicable
    //applyDiscount(totalLoyalty);

    // Calculate total amount
    calculateTotalAmount();

    // Calculate balance
    calculateBalance();
} catch (NumberFormatException ex) {
    // Handle parsing errors
    // Display error message or perform appropriate action
}
}
```

Figure 2.171: net amount calculation of billing management

When an action is taken on the net amount input field, the netamountActionPerformed method is called as an event handler. The first step in the procedure is to try to parse the net amount that the user entered. If the parsing is successful, the extra loyalty points are computed using the net amount; one loyalty point is given for each thousand rupees spent.

The associated loyalty points field is then updated with this computed value. The procedure then invokes calculateFieldsAndUpdate (), which modifies other relevant fields. Calls to calculateTotalAmount () and calculateBalance (), which determine the total amount and the balance owed, respectively, are also included in the method. The provided code snippet does not describe the precise error handling steps, but it catches the exception in case of a NumberFormatException (caused by invalid input) and manages any parsing failures accordingly.

❖ Paid amount calculation

```
  private void pamountActionPerformed(java.awt.event.ActionEvent evt) {
 // Validate if the entered value is a valid integer
  try {
  String enteredPaidAmount = pamount.getText().trim();
  int paidAmount = Integer.parseInt(s: enteredPaidAmount);

  // Check if the entered value is negative
  if (paidAmount < 0) {
      JOptionPane.showMessageDialog(parentComponent:null, message:"Paid amount cannot be negative.",
              title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
      pamount.setText(t: ""); // Clear the field
  } else {
      // Update paidAmount variable with the entered value
      this.paidAmount = paidAmount;

      // Calculate fields and update them
      calculateFieldsAndUpdate();
  }
catch (NumberFormatException ex) {
  JOptionPane.showMessageDialog(parentComponent:null, message:"Please enter a valid integer for the paid amount.",
          title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
  pamount.setText(t: ""); // Clear the field
  }
      // Calculate fields and update them
      calculateFieldsAndUpdate();
```

Figure 2.172: paid amount calculation of billing management

The `pamountActionPerformed` method is an event handler triggered when an action is done on the paid amount input field. To make sure the entered number is a valid integer, it first tries to parse it. It determines whether the entered amount is negative if parsing is successful. If so, the field is cleared and an error message stating that the paid amount cannot be negative is displayed to the user. If not, it uses the "calculateFieldsAndUpdate()" function to recalculate and update related fields and updates the "paidAmount" variable with the entered value. The program captures `NumberFormatException` exceptions (which indicate invalid input), shows an error message, and clears the field. To guarantee constant changes to the fields, it also calls "calculateFieldsAndUpdate()" at the end.

❖ Clear button for the billing data text fields

```
private void clearBActionPerformed(java.awt.event.ActionEvent evt) {
    // Reset all the text fields to their initial state
custName.setText(t: "");
netamount.setText(t: "0");
AddLoyal.setText(t: "0");
totamount.setText(t: "0");
disc.setText(t: "0");
pamount.setText(t: "0");
balance.setText(t: "0");

// Reset the date picker components to the current date
orderdate.setDate(date: Date.from(instant: Instant.now()));
pickupdate.setDate(date: Date.from(instant: Instant.now()));

// Reset the Order IDs combo box

}
```

Figure 2.173: clear button code of billing management

When a button is clicked by the user to clear all input fields and return them to their initial state, the clearBActionPerformed function is triggered. This is accomplished by resetting amounts to zero and other text field text to their default values. Furthermore, it uses the Date.from(Instant.now()) function to reset the date picker components to the current date. Lastly, it makes sure that the Order IDs combo box is also returned to its original state, so erasing all data that has been submitted by the user and making the form ready for fresh data.

❖ **Refresh the Billing Table after a modification.**

```
    private void refreshBilltable() {
try {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery(string: "SELECT * FROM billing");

        DefaultTableModel model = new DefaultTableModel();
        model.addColumn(columnName: "receipt id");
        model.addColumn(columnName: "Order id");
        model.addColumn(columnName: "Name");
        model.addColumn(columnName: "Amount");
        model.addColumn(columnName: "Order date");
        model.addColumn(columnName: "Loyalty Points");
        model.addColumn(columnName: "Discount");
        model.addColumn(columnName: "Total Amount");
        model.addColumn(columnName: "Paid Amount");
        model.addColumn(columnName: "Balance");
        model.addColumn(columnName: "Pickup date");
```

```
    while (rs.next()) {
        Object[] row = {
            rs.getInt(string: "receiptID"),
            rs.getString(string: "OrderID"),
            rs.getString(string: "f_name"),
            rs.getString(string: "Netamount"),
            rs.getString(string: "orderdate"),
            rs.getString(string: "addLoyal"),
            rs.getString(string: "discount"),
            rs.getString(string: "totAmount"),
            rs.getString(string: "paidAmount"),
            rs.getInt(string: "balance"),
            rs.getDate(string: "pickupdate") // Use getDate for pickup date
        };
        model.addRow(rowData:row);
    }

    bill.setModel(dataModel: model);
} catch (SQLException ex) {
    JOptionPane.showMessageDialog(parentComponent:null, "Error occurred while refreshing Table: " + ex.getMessage(),
            title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
}
}
```

Figure 2.174: refresh table code of billing management

The update the billing table in the user interface is updated with the latest information retrieved from the database using the “billtable()” method. To do this, a SQL query is run to obtain every billing record from the "billing" database table. Next, the function defines the columns for the table, such as "receipt id," "order id," "name," "Amount," "Order date," "Loyalty Points," "Discount," "Total Amount," "Paid Amount," "Balance," and "Pickup date," by creating a DefaultTableModel object. The method extracts the pertinent data for each row that is fetched from the result set (rs) and adds it to an Object array that represents a row in the database. The model is then updated with this array. Lastly, the procedure adds the revised data-containing model to the billing table (bill). An error message dialog is presented to the user if an exception arises during the database query or table update procedure.

All things considered, this approach makes sure that the billing table is updated with the most recent billing data from the database, giving users accurate and current information.

❖ Update Button for billing data.

```
private void updateBtnActionPerformed(java.awt.event.ActionEvent evt) {
try {
    String FirstName, orderDate, pickupDate, order_Id;
    int netamount, loyalty, disco, totalamount, amountpaid, balancee;

    FirstName = fname.getText();
    order_Id = (String) orderid.getSelectedItem();

    SimpleDateFormat sdf = new SimpleDateFormat(pattern:"yyyy-MM-dd");
    orderDate = sdf.format(date:odate.getDate());

    SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern:"yyyy-MM-dd");
    pickupDate = simpleDateFormat.format(date:pdate.getDate());

    // Parse amount fields with exception handling
    try {
        netamount = Integer.parseInt(s: amount.getText());
        loyalty = Integer.parseInt(s: points.getText());
        disco = Integer.parseInt(s: disc.getText());
        totalamount = Integer.parseInt(s: tamount.getText());
        amountpaid = Integer.parseInt(s: pamount.getText());
        //balancee = Integer.parseInt(blc1.getText());
        balancee = amountpaid - totalamount;
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(parentComponent:null, message:"Invalid Input for Amount...!",
                title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
        return; // Exit method if parsing fails
    }
```

```
    try {
        Statement st = DBconnection.createDBconnection().createStatement();
        int x = Integer.parseInt(s: reid.getText());

        String query = "UPDATE billing SET f_name = ?, OrderID = ?, Netamount = ?,"
                + " orderdate = ?, addLoyal = ?, discount = ?, totAmount = ?, paidAmount = ?, balance = ?, "
                + "pickupdate = ? WHERE receiptID = ?";

        java.sql.PreparedStatement prepSt = st.getConnection().prepareStatement(string: query);

        prepSt.setString(i: 1, string: FirstName);
        prepSt.setString(i: 2, string: order_Id);
        prepSt.setInt(i: 3, i1: netamount);
        prepSt.setString(i: 4, string: orderDate);
        prepSt.setInt(i: 5, i1: loyalty);
        prepSt.setInt(i: 6, i1: disco);
        prepSt.setInt(i: 7, i1: totalamount);
        prepSt.setInt(i: 8, i1: amountpaid);
        prepSt.setInt(i: 9, i1: balancee);
        prepSt.setString(i: 10, string: pickupDate);
        prepSt.setInt(i: 11, i1: x);

        int count = prepSt.executeUpdate();
int option = JOptionPane.showConfirmDialog(parentComponent:null, message:"Are you sure you want to add billing data?",
        title: "Confirmation", optionType: JOptionPane.YES_NO_OPTION);
    if (option == JOptionPane.YES_OPTION){
            JOptionPane.showMessageDialog(parentComponent:null, message:"Selected Bill's Details Updated Successfully",
                    title: "Success", messageType:JOptionPane.INFORMATION_MESSAGE);
            refreshBilltable();
        } else {
            JOptionPane.showMessageDialog(parentComponent:null, message:"Failed to update selected bill",
                    title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
        }
    } catch (SQLException e) {
        e.printStackTrace();
        JOptionPane.showMessageDialog(parentComponent:null, message:"Error occurred while Updating the selected Bill's Details",
                title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
```

Figure 2.175: update button code of billing management

When a user clicks on a button in the user interface, titled "Update" or something similar, the “updateBtnActionPerformed()” function is called. It oversees updating the database's billing information considering the user's amended input. This is an overview of the functions of the method: It obtains the updated data that the user has entered from a variety of text fields and date pickers within the user interface. It converts the supplied money and loyalty points into integers and uses a NumberFormatException catch block to manage any parsing issues. To update the appropriate billing record in the database with the updated data, it creates a SQL UPDATE query. Using the obtained and parsed data, it prepares a statement and establishes the query's parameters. To find out if the update was successful, it runs the update query and counts the rows that are impacted. The user receives a success message, and the billing table is refreshed to show the updated information if the update was successful.The user sees an error message if the update does not work. Given the circumstances, this technique guarantees that the user can accurately update billing information and notifies the user if the update process was successful or unsuccessful. Furthermore, it guards against SQL injection threats by employing prepared statements to guarantee data integrity.

❖ **Delete button for billing data.**

```
private void deleteBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    try {
        Statement st = DBconnection.createDBconnection().createStatement();
        String query = "DELETE FROM billing WHERE receiptID = " + Integer.valueOf(s: reid.getText());
        int count = st.executeUpdate(string: query);
int option = JOptionPane.showConfirmDialog(parentComponent:null, message:"Are you sure you want to add billing data?",
        title: "Confirmation", optionType: JOptionPane.YES_NO_OPTION);
if (option == JOptionPane.YES_OPTION) {
            JOptionPane.showMessageDialog(parentComponent:null, message:"Selected Bill Details Deleted Successfully.",
                    title: "Success", messageType:JOptionPane.INFORMATION_MESSAGE);
            refreshBilltable();
        }

        bill.clearSelection();
        refreshBilltable();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(parentComponent:null, "Error occurred while Deleting Tuple: " + ex.getMessage(),
                title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
    }

}
```

Figure 2.176: delete button code of billing management

The backend mechanism for deleting billing records from the database within a graphical user interface is provided by the `deleteBtnActionPerformed` function. First, it takes the billing entry's receipt ID that must be removed out of a specified text field. It then creates a SQL {DELETE} query based on the receipt ID found to successfully remove the associated billing record from the database. The deletion query is executed, and the method shows a confirmation popup to the user to make sure they want to proceed with the deletion. A success message confirming the effective removal of the billing details is shown after the user confirms. In addition, it ensures real-time data synchronization by updating the billing table in the user interface to reflect the most recent database state.The method gracefully manages any unforeseen issues that may arise during the deletion process, such as SQL exceptions, by informing the user of the error and preserving data integrity. By using this approach, users can profit from a strong and engaging interface while managing and maintaining billing records with confidence.

❖ Bill mouse clicked.

```
    private void billMouseClicked(java.awt.event.MouseEvent evt) {
int selectedRow = bill.getSelectedRow();
    TableModel model = bill.getModel();

    if (selectedRow != -1) {
        Object receiptIDObj = model.getValueAt(rowIndex: selectedRow, columnIndex:0);
        Object orderIDObj = model.getValueAt(rowIndex: selectedRow, columnIndex:1);
        Object nameObj = model.getValueAt(rowIndex: selectedRow, columnIndex:2);
        Object amountObj = model.getValueAt(rowIndex: selectedRow, columnIndex:3);
        Object orderDateObj = model.getValueAt(rowIndex: selectedRow, columnIndex:4);
        Object loyaltyObj = model.getValueAt(rowIndex: selectedRow, columnIndex:5);
        Object discountObj = model.getValueAt(rowIndex: selectedRow, columnIndex:6);
        Object totalAmountObj = model.getValueAt(rowIndex: selectedRow, columnIndex:7);
        Object paidAmountObj = model.getValueAt(rowIndex: selectedRow, columnIndex:8);
        Object balanceObj = model.getValueAt(rowIndex: selectedRow, columnIndex:9);
        Object pickupDateObj = model.getValueAt(rowIndex: selectedRow, columnIndex:10);

        // Check for null values before invoking toString()
        String receiptID = receiptIDObj != null ? receiptIDObj.toString() : "";
        String orderID = orderIDObj != null ? orderIDObj.toString() : "";
        String name = nameObj != null ? nameObj.toString() : "";
        String amou = amountObj != null ? amountObj.toString() : "";
        String orderDate = orderDateObj != null ? orderDateObj.toString() : "";
        String loyalty = loyaltyObj != null ? loyaltyObj.toString() : "";
        String discount = discountObj != null ? discountObj.toString() : "";
        String totalAmount = totalAmountObj != null ? totalAmountObj.toString() : "";
        String paidAmount = paidAmountObj != null ? paidAmountObj.toString() : "";
        String balance = balanceObj != null ? balanceObj.toString() : "";
```

```
        points.setText(t: loyalty);
        disc.setText(t: discount);
        tamount.setText(t: totalAmount);
        amount.setText(t: amou);

        reid.setText(t: receiptID);

        DefaultComboBoxModel<String> comboBoxModel = new DefaultComboBoxModel<String>();
        comboBoxModel.addElement(anObject: orderID);
        orderid.setModel(aModel: comboBoxModel);

        fname.setText(t: name);
        pamount.setText(t: paidAmount);

        SimpleDateFormat sfd = new SimpleDateFormat(pattern:"yyyy-MM-dd");
        try {
            if (!pickupDate.isEmpty()) {
                Date pickup = sfd.parse(source: pickupDate);
                pdate.setDate(date: pickup);
            } else {
                pdate.setDate(date: null);
            }
        } catch (ParseException e) {
            e.printStackTrace();
            pdate.setDate(date: null);
        }
```

Figure 2.177: Bill mouse clicked of billing management

In a graphical user interface, the “billMouseClicked()” method is intended to manage mouse clicks on the billing table. It retrieves the information from the table that corresponds to the chosen row, including the order ID, receipt ID, client name, quantities, dates, and other pertinent information.

It makes sure that the chosen billing entry is accurately represented by obtaining these values from the table model.

The collected data is then used to fill in several text fields and combo boxes, giving users the ability to examine and edit the specifics of the chosen billing record. It also uses the retrieved numbers to establish the chosen loyalty points, discount, total amount, and paid amount, giving users a thorough picture of the billing details.

It guarantees correct processing and presentation of date values through formatting and parsing procedures. In addition, the approach includes error handling procedures to manage possible parsing errors politely, preserving the UI's stability and usability. Given the circumstances, this approach improves user interaction by offering simple access to payment information and enabling fluid interface navigation.

- **Update the Paid amount and balance at the same time after the mouse clicked.**

```
    private void pamountActionPerformed(java.awt.event.ActionEvent evt) {
String paidAmountText = pamount.getText().trim();

    // Validate if the entered text is a valid integer
    try {
        int paidAmount = Integer.parseInt(s: paidAmountText);
        // If the entered text is a valid integer, calculate and update the balance
        int totalAmount = Integer.parseInt(s: tamount.getText()); // Get the total amount
        int balance = paidAmount - totalAmount; // Calculate the balance
        blc1.setText(t: String.valueOf(i: balance)); // Update the balance text field
    } catch (NumberFormatException ex) {
        // If the entered text is not a valid integer, display an error message
        JOptionPane.showMessageDialog(parentComponent:null, message:"Please enter a valid integer for paid amount.",
            title:"Error", messageType:JOptionPane.ERROR_MESSAGE);
    }


// Add key listener to the paid amount text field
pamount.addKeyListener(new java.awt.event.KeyAdapter() {
    public void keyPressed(java.awt.event.KeyEvent evt) {
        if (evt.getKeyCode() == KeyEvent.VK_ENTER) {
            pamountActionPerformed(evt:null); // Call the method when the enter key is pressed
        }
    }
});
    }
```

Figure 2.178: update paid amount code of billing management

This code snippet's pamountActionPerformed method is called when the user enters a value in the paid amount text field (pamount) or performs another action. To ensure correct parsing, it starts by extracting the text that was entered into the field and removing any leading or trailing whitespace. Next, to determine whether the text entered reflects a legitimate number value, it parses the text into an integer.

If successful, it deducts the paid amount from the total amount (retrieved from a separate text field called “tamount”) to determine the balance. Next, the computed balance is updated in field blc1, another text field.

On the other hand, it captures the NumberFormatException and raises a dialog box with an error message if the text entered cannot be parsed into an integer, indicating that it is not a valid numeric input.The method also adds a key listener, which is specifically designed to listen for the Enter key, to the mount text box. Users can conveniently initiate the balance computation by pressing Enter, as it executes the pamountActionPerformed method one more.
This feature guarantees that customers may quickly and easily compute the corresponding balance with little effort, precisely input and validate the paid amount, and receive fast feedback on problems.

Major module structures, reusable codes & Development tools used in Billing Management Page’s Functionality

1) Major Module Structures

- BillingDetails Class: Javax.swing is extended by this class. JFrame and functions as the billing application's main window.
    - Fields:

      Balance, discount, and total amountValue and paidAmount are integer fields used to hold amounts of money. orderCustomerMap: A hash map that associates customer IDs with order IDs.
    - Swing Components:

      A variety of elements, including text fields, buttons, and date pickers (custName, orderdate, netamount, AddLoyal, totamount, pamount, pickupdate, balance, and OrdID) are examples of swing components.

- GUI Elements Initialization initComponents (): This method sets up the GUI and initializes Swing components.
  Event listeners for several buttons (backBtn, logoutBtn, exitBtn, addBtn1, billBtn, clearBtn, printR, viewR).

2)Reusable Codes

1. database connection. createDBconnection (): The procedure for opening a database connection. (Utilized often in various ways to run SQL queries and get data.)

2. Practical Techniques
   Get the net amount for a specific order ID with the getNetAmountByID (int OrderID) function. getOrderIDs () method Retrieves related customer information and populates a combo box with the order IDs.

3)Development Tools

- Java Development Kit (JDK): Used to compile and execute Java programs.
- Integrated Development Environment (IDE): For development and debugging, use tools like Eclipse or IntelliJ IDEA.
- Swing Library: Used in the development of graphical user interfaces.
- JDBC: For running SQL queries and connecting to databases.

8)Sales Management Page - (IT22310996 – THENNAKOON T.A.C.S)

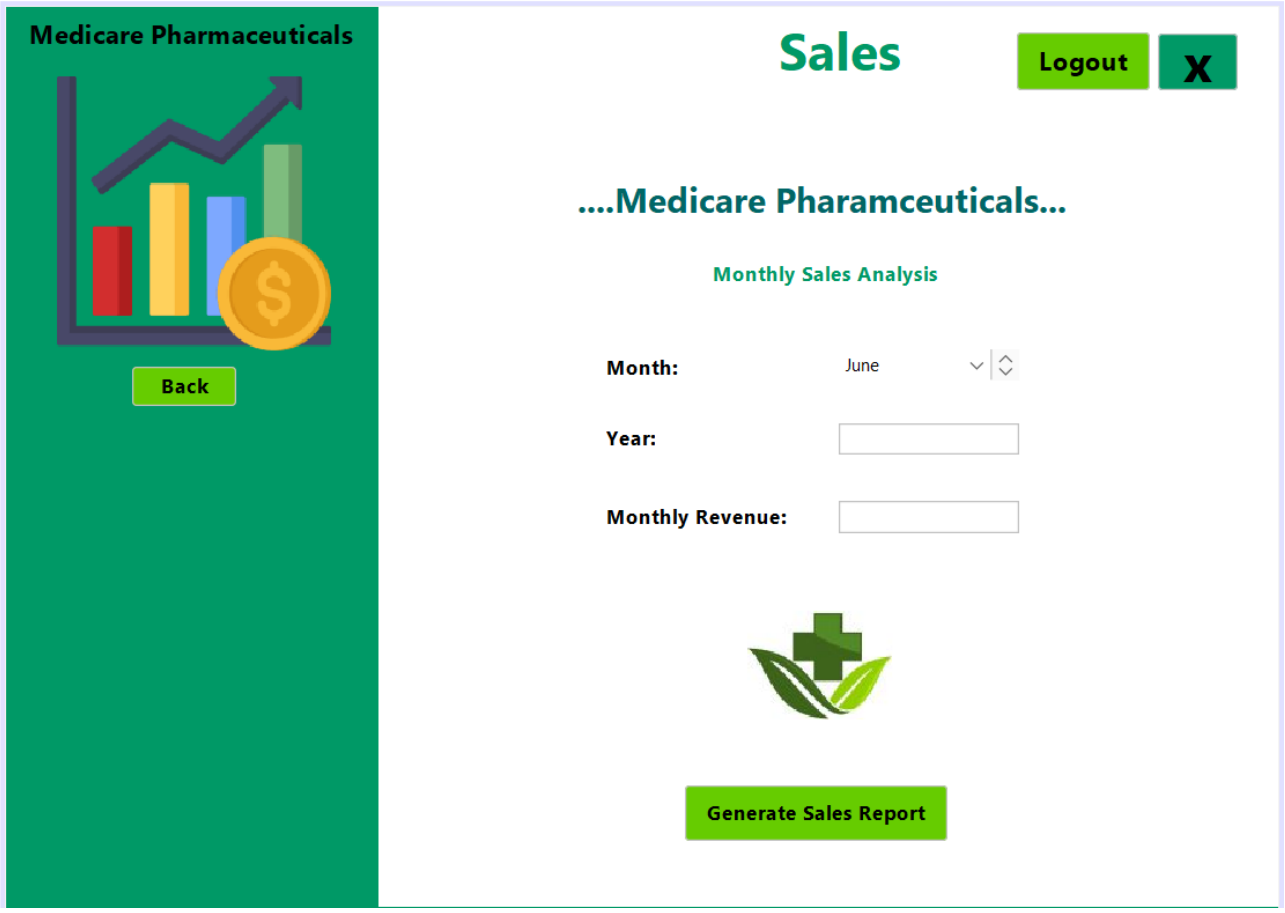

Figure 2.179: Sales Management Page UI

```
private void generateMonthlyReport() {

try {
        // Get month and year from text fields
        int selectedMonth = Integer.parseInt(s: month.getText());
        int selectedYear = Integer.parseInt(s: year.getText());

        // Construct the start and end dates of the selected month
        SimpleDateFormat sdf = new SimpleDateFormat(pattern:"yyyy-MM-dd");
        Date startDate = sdf.parse(selectedYear + "-" + selectedMonth + "-01");
        Date endDate = new Date(year:startDate.getYear(), startDate.getMonth() + 1, date:1); // End date is the first day of next month

        // Query to fetch billing details within the selected month
        String billingQuery = "SELECT SUM(Netamount) AS total_revenue "
                            + "FROM billing "
                            + "WHERE MONTH(pickupdate) = " + selectedMonth + " AND YEAR(pickupdate) = " + selectedYear;

        Statement st = DBconnection.createDBconnection().createStatement();

        // Fetch billing details
        ResultSet billingRS = st.executeQuery(string: billingQuery);

        // Process the results and generate the report
        int totalRevenue = 0;
        if (billingRS.next()) {
            totalRevenue = billingRS.getInt(string: "total_revenue");
        }
```

```
        // Fetch billing details
        ResultSet billingRS = st.executeQuery(string: billingQuery);

        // Process the results and generate the report
        int totalRevenue = 0;
        if (billingRS.next()) {
            totalRevenue = billingRS.getInt(string: "total_revenue");
        }

        // Set the value of the revenue text area
        revenue.setText(t: String.valueOf(i: totalRevenue));

        // Display the report in a JTextArea
        String report = "Total Monthly Revenue for " + selectedMonth + "/" + selectedYear + ": " + totalRevenue;
        JTextArea reportArea = new JTextArea(text: report);
        JScrollPane scrollPane = new JScrollPane(view: reportArea);
        JOptionPane.showMessageDialog(parentComponent:this, message:scrollPane, title: "Monthly Revenue Report",
                messageType:JOptionPane.INFORMATION_MESSAGE);
    } catch (NumberFormatException | ParseException | SQLException ex) {
        JOptionPane.showMessageDialog(parentComponent:null, "Error generating monthly report: " + ex.getMessage(),
                title: "Error", messageType:JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 2.180: report generation of sales management

A monthly income report based on the user-selected month and year is produced by the "generateMonthlyReport()" method. First retrieves Month and Year. Then the procedure begins by parsing the user-entered month and year from the corresponding text fields. It constructs the start date as the first day of the chosen month and the end date as the first day of the subsequent month using the parsed month and year. A SQL query is created to retrieve the total income from the billing information for the chosen month and year. The billing records are filtered by this query according to the date of pickup. The query is run by the method, which then extracts the total income from the result set. The entire revenue is kept in an integer variable if there are records. Display Report: It establishes the total revenue value in the revenue "JTextArea." After that, it creates a report string with the entire amount of money received each month and shows it in a "JOptionPane" message window that has a scrollable "JTextArea."

During the procedure, the function captures potential errors like NumberFormatException, Parse Exception, and SQL Exception. It notifies the user of the problem by displaying an error message box if an exception occurs. Given the circumstances, this technique improves the application's reporting capabilities by offering a user-friendly means of creating and presenting monthly income reports.

Major module structures, reusable codes & Development tools used in Supplier Management Page's Functionality

1) Major Module Structures

- Pharmaceutical package: the package that the class Sales is assigned to. This file appears to be a component of the pharmaceutical package.
- Class Defined: Javax.swing is extended by the public class Sales.JFrame: This defines the primary class Sales, which is an extension of the Swing library's JFrame class.
- Bringing in: In order to perform several operations including GUI components, database connectivity, printing, and date manipulation, Java AWT, SQL, Swing, and other utility libraries are imported.

2) Reusable Codes

- Link to Database: Conn = DBconnection is the connection. createDBconnection (); Uses a reusable technique to create a database connection. establish a DB connection.
- Error Resolution: Try-catch blocks are used to handle exceptions such as NumberFormatException and SQLException.
- Practical Techniques: JOptionPane.showMessageDialog(...): This function is used to notify or error-message users.
- Printing: utilizes PrinterJob to print the panel's contents and implements the Printable interface.

2.4 Testing

The Desktop Pharmaceutical Management Application testing strategy will include detailed methods to verify the functionality and performance of every functional element. To make that each module's functionality complies with requirements, unit testing will be required. Integration testing will also confirm that modules work together seamlessly. End users will participate in user acceptance testing (UAT) to evaluate usability and confirm system needs. To make sure upgrades or alterations don't negatively impact already-existing functionalities, regression testing will be carried out. For efficiency, automated testing technologies will be used, while scenarios needing human judgment will be covered by manual testing. Comprehensive test cases and results documentation will demonstrate that the system has been thoroughly tested and validated. [5][7]

1) Medicine Inventory Management (IT22364692 – KANDAGE K.T.S)

Table 2.1 : Medicine Inventory Management Test Cases

Test Case ID	Test Case Name	Input Data	Expected Output	Actual Result	Pass/Fail
M_001	Add a new medicine to the inventory	Provide medicine details (and click the 'Add' button) Medicine Name: 'Piriton' Type: Allergic Dosage: 1 tablet Unit Price (Rs.): 2.00 Storage Location: A601 Date Added: 11.06.24 Quantity: 100	Displays pop-up message, "New Medicine Added Successfully" and display newly added medicine in the 'Available Medicine List' table.	Displays newly added medicine in the 'Available Medicine List'.	Pass
M_002	Add a medicine that has already been added (duplicate)	Provide medicine details (and click the 'Add' button). Medicine Name: 'Piriton' Type: Allergic Dosage: 1 tablet Unit Price (Rs.): 2.00 Storage Location: A601 Date Added: 11.06.24 Quantity: 50	Displays pop-up message, "New Medicine Added Successfully" and only the available quantity of medicines should be updated, or the if it is not the same storage location, then it is added newly.	Displays pop-up message, "Selected Medicine Details Added Successfully" and the quantity of available medicines should be updated. If it is not the same storage location, then it is added newly.	Pass
M_003	Update already added medicine details.	Select the row to be updated and update medicine details (and click the 'Update' button). Medicine Name: 'Panadol' Type: Analgesics Dosage: 2 tablets Unit Price (Rs.): 3.00 Storage Location: B402 Date Added: 11.06.24 Quantity: 200	Displays pop-up message, "Selected Medicine Details Updated Successfully" and that selected medicine details updated in the table.	Displays pop-up message, "Selected Medicine Details Updated Successfully" and that medicine details updated in the table.	Pass

M_004	Delete added medicine details	Select the row to delete (the text fields will be filled in automatically) and click the 'Delete' button.	Displays pop-up message, "Medicine Deleted Successfully" and that medicine details deleted in the table.	Display pop-up message, "Medicine Deleted Successfully" and that selected medicine details deleted successfully.	Pass
M_005	Try to add a medicine detail that is not a 'Unit Price' or 'Quantity' number	Provide medicine details (and click the 'Add' button) Medicine Name: 'Amoxicillin' Type: Antibiotics Dosage: 1 tablet Unit Price (Rs.): 'abc' Storage Location: B401 Date Added: 11.06.24 Quantity: 'pqr'	Displays pop-up message, "Enter Numerical Values for the Unit Price or Quantity!" and that medicine is not added to the 'Available Medicine List' table.	Displays pop-up message, "Enter Numerical Values for the Unit Price or Quantity!" and that medicine is not added to the 'Available Medicine List' table.	Pass
M_006	Search for medicines using the search bar	Enter the Medicine Name or Storage Location in the search bar. Search Text: 'Panadol' Or Search Text: B402 Input: 1	'Available Medicine List' table display the relevant medicine details searched in the search bar. Medicine ID: 1 Medicine Name: 'Panadol' Type: Analgesics Dosage: 2 tablets Unit Price (Rs.): 3.00 Storage Location: B402 Date Added: 11.06.24 Quantity: 200	'Available Medicine List' table display the relevant medicine details searched in the search bar. Medicine ID: 1 Medicine Name: 'Panadol' Type: Analgesics Dosage: 2 tablets Unit Price (Rs.): 3.00 Storage Location: B402 Date Added: 11.06.24 Quantity: 200	Pass
M_007	Generate the Inventory Report and print it	Click the 'Inventory Report Button'.	Successfully submitted inventory report to printer.	The inventory report was successfully printed.	Pass

2) Stock Management (IT22364692 – KANDAGE K.T.S)]

Table 2.2: Medicine Stock Management Test Cases

Test Case ID	Test Case Name	Input Data	Expected Output	Actual Result	Pass/Fail
S_ 001	Check that when a Medicine ID is chosen from the combo box, the proper 'Medicine Name' is shown.	From the combo box, select a Medicine ID	Based on the selected ID, the 'Medicine Name' text field should display relevant medicine name.	Based on the selected ID, displayed the relevant medicine name in 'Medicine Name' text field.	Pass
S_ 002	Check that when a Medicine ID is chosen from the combo box, the proper 'Storage Location' is shown.	From the combo box, select a Medicine ID	Based on the selected ID, the 'Storage Location' text field should display relevant storage location.	Based on the selected ID, displayed the relevant storage location in the 'Storage Location' text field.	Pass
S_ 003	Add a new stock to the Available Stock List table	Provide stock details (and click the 'Add' button) Quantity: 200 Medicine ID: 1 Medicine Name: 'Panadol' Storage Location: B402 Manufactured Date: 11.05.24 Expired Date: 11.06.24 Days To Expire: 11.06.25	The pop-up message displays, "New stock successfully added" and the newly added stock should be displayed in the 'Available Stock List' table and the current stock quantity should be updated in the Available Stock List table.	The pop-up message displays, "New stock successfully added" and the newly added stock is displayed in the 'Available Stock List' table and the current stock quantity is updated in the Available Stock List table.	Pass
S_ 004	Update already added stock details	Select the row to be updated and update stock details (and click the 'Update' button);	Displays pop-up message, "Stock updated successfully, and Medicine quantity updated" and that stock details updated, and medicine quantity updated in the relevant tables.	Displays pop-up message, "Stock updated successfully, and Medicine quantity updated" and that stock details updated, and medicine quantity updated in the relevant tables.	Pass

S_ 005	Delete added stock details	Select the row to delete (the text fields will be filled in automatically) and click the 'Delete' button.	Displays pop-up message, "Stock deleted successfully, and Medicine quantity updated" and that stock details deleted, and medicine quantity updated in the relevant tables.	Displays pop-up message, "Stock deleted successfully, and Medicine quantity updated" and that stock details deleted, and medicine quantity updated in the relevant tables.	Pass
S_ 006	Search for available stocks using the search bar	Enter the Medicine Name or Storage Location in the search bar. Search Text: 'Panadol' Or Search Text: B402	'Available Stock List' table displays the stock details searched in the search bar. Stock ID: 1 Quantity: 200 Medicine ID: 1 Medicine Name: 'Panadol' Storage Location: B402 Manufactured Date: 11.05.24 Expired Date: 11.06.24 Days To Expire: 11.06.25	'Available Stock List' table displays the stock details searched in the search bar. Stock ID: 1 Quantity: 200 Medicine ID: 1 Medicine Name: 'Panadol' Storage Location: B402 Manufactured Date: 11.05.24 Expired Date: 11.06.24 Days To Expire: 11.06.25	Pass

3)Employee Management – Salary & Attendance (IT22319142 – WIJESINGHE A.G.T)

Table 2.3: Employee Management Test Cases

Test Case ID	Test Case Name	Input Data	Expected Output	Actual Result	Pass/Fail
EM_001	Register a New Employee – **Valid Input**	Employee ID: 7, First Name: Chirath, Last Name: Gomez, Date of Birth: 1995-03-10, Gender: Male, NIC: 199557401156, Email: chirath@gmail.com, Job Role: Accountant, Date of Joining:2020-01-29, Daily Rate:3500, Contact No:0771538567	New Employee 's details added successfully	New Employee 's details added successfully	Pass
EM_002	Register a New Employee – **Same NIC of a Registered Employee**	Employee ID: 8, First Name: Vihanga, Last Name: Senani, Date of Birth: 1998-07-20, Gender: Female, NIC: 199557401156, Email: vihasenani@gmail.com, Job Role: Assistant Pharmacist, Date of Joining:2022-05-25, Daily Rate:2500, Contact No:0786458567	Error message " Please check whether the NIC is correct! There is another registered employee with the same NIC you are trying to insert!"	Error message " Please check whether the NIC is correct! There is another registered employee with the same NIC you are trying to insert!"	Pass
EM_003	Register a New Employee – **Incorrect NIC**	Employee ID: 9, First Name: Sarika, Last Name: Samson, Date of Birth: 1990-09-30, Gender: Female, NIC: 19901156, Email: sarika1990@gmail.com, Job Role: Accountant, Date of Joining:2019-04-29, Daily Rate:3500, Contact No:0761538567	Error message " Please enter 12 numerical values for the NIC!"	Error message " Please enter 12 numerical values for the NIC!	Pass

EM_004	Record Attendance Status for an Employee - **Valid Input**	Attendance ID: 83 Date:2024-06-13 Employee ID:7 Employee Name: Chirath Attendance: Present	Attendance Recorded Successfully	Attendance Recorded Successfully	Pass
EM_005	Record Attendance Status for an Employee - **For the same Employee for the same day**	Attendance ID: 83 Date:2024-06-13 Employee ID:7 Employee Name: Chirath Attendance: Present	Error message " This Employee's today's attendance has already been marked!"	Error message " This Employee's today's attendance has already been marked!"	Pass
EM_006	Update an Attendance record - **Without selecting a Row**	Click update button without selecting a row	Error message " Please select a row to update!"	Error message " Please select a row to update!"	Pass
EM_007	Insert an Employee's Salary Record - **Valid Input**	Salary ID: 8 Employee ID: 7 Employee Name: Chirath Daily rate:3500 Year: 2024 Month:06 Attendance: Net Salary:	Auto Generate the Attendance = 1 & Net Salary = Rs:3500. Salary Record was Successfully added to the Salary Table	Auto Generate the Attendance = 1 & Net Salary = Rs:3500. Salary Record was Successfully added to the Salary Table	Pass
EM_008	Insert an Employee's Salary Record - **Without entering a year**	Salary ID: 8 Employee ID: 7 Employee Name: Chirath Daily rate:3500 Year: **NULL** Month:06 Attendance: Net Salary:	Error Message "Year cannot be empty!"	Error Message "Year cannot be empty!"	Pass

4)Supplier Management (IT22319142 – WIJESINGHE A.G.T)

Table 2.4: Supplier Management Test Cases

Test Case ID	Test Case Name	Input Data	Expected Output	Actual Result	Pass/Fail
SM_001	Register a New Supplier – **Valid Input Values**	Supplier ID: 5 Company Name: Health Guard Pvt LTD, Medicine ID: 2, Medicine Name: Norvasc, Email: healthguard234@gmail.com, Contact No: 0112769810, Contact Person: Ms. Sherin Date: 2024-06-08	New supplier's details added successfully	New supplier's details added successfully	Pass
SM_002	Register a New Supplier - **Invalid Contact No**	Supplier ID: 6 Company Name: Life First Pvt LTD, Medicine ID: 3, Medicine Name: Zoloft, Email: life123first@gmail.com, Contact No: 011270, Contact Person: Ms. Harini Date: 2024-06-08	Error message " Enter exactly 10 numerical values for the contact number!"	Error message " Enter exactly 10 numerical values for the contact number!"	Pass
SM_003	Register a New Supplier – **without filling some of the required input fields**	Supplier ID: 7 Company Name: Medicine ID: 1, Medicine Name: Metformin, Email: @gmail.com, Contact No: 0115867810, Contact Person: Ms. Anne Date: 2024-06-10	Error message "Please fill all the required fields!"	Error message "Please fill all the required fields!"	Pass
SM_004	Register a New Supplier – **Invalid Email Address**	Supplier ID: 8 Company Name: Suwasetha Distributors, Medicine ID: 2, Medicine Name: Norvasc, Email: suwasethacolombo@.com, Contact No: 0112064687, Contact Person: Mr. Thomas Date: 2024-06-08	Error message "Enter a valid Gmail address!"	Error message "Enter a valid Gmail address!"	Pass

SM_005	Register a New Supplier – **already registered company name & same official Email Address**	Supplier ID: 9 Company Name: Health Guard Pvt LTD, Medicine ID: 4, Medicine Name: Losartan, Email: healthguard234@gmail.com, Contact No: 0112758240, Contact Person: Ms. Gayani Date: 2024-06-08	Error message " The company you are trying to Add is already registered in the database!"	Error message " The company you are trying to Register is already registered in the database!"	Pass
SM_006	Search a registered Supplier record by **valid Supplier ID**	Supplier ID: 5	Details related to the Supplier with the Supplier ID 5 are sorted out and displayed successfully	Details related to the Supplier with the Supplier ID 5 are sorted out and displayed successfully	Pass
SM_007	Delete a Supplier record by **valid Supplier ID**	Supplier ID: 5	Selected Supplier 's details deleted successfully	Selected Supplier 's details deleted successfully	Pass

5)Customer Management (IT22884138 – RATHNAYAKA R.M.T.D)

Table 2.5: Customer Management Test Cases

Test Case ID	Test Case Name	Input Data	Expected Output	Actual Result	Pass/Fail
001	Add New Customer - Valid Input	First Name: John, Last Name: Doe, Registration ID:1000000, Email: johndoe@gmail.com, Contact No: 0776543210, Date: 2024-06-10, Loyalty Points: 10	New customer's details added successfully	New customer's details added successfully	Pass
002	Add New Customer - Invalid Email	Email: Jhonwen@gmail	Error message "Enter valid email"	Error message "Enter valid email"	Pass
003	Add New Customer - Invalid Contact No	Contact No: 07181135	Error message "Enter valid phone number"	Error message "Enter valid phone number"	Pass
004	Update Customer Email-	First Name: John, Last Name: Doe, Registration ID:1000000, Email: Jhonethen@gmail.com, Contact No: 0776543210, Date: 2024-06-10, Loyalty Points: 10	Selected customer's details updated successfully	Selected customer details updated successfully	Pass
005	Delete Customer by valid ID-	Customer ID: 2	Selected customer's details deleted successfully	Selected customer's details deleted successfully	Pass
006	Search Customer by valid ID -	Search Input: 1	Table displays customer with ID 1	Table displays customer with ID 1	Pass
007	Search Customer by ID - Invalid ID	Search Input: abc	Error message "Invalid input!"	No customer displayed	Fail
008	Navigate to Dashboard	Click Back Button	Current window closed; Dashboard window displayed	Current window closed; Dashboard window displayed	Pass

6)Order Management (IT22884138 – RATHNAYAKA R.M.T.D)

Table 2.6: Order Management Test Cases

Test Case ID	Test Case Name	Input Data	Expected Output	Actual Result	Pass/Fail
001	Adding Medicine to Cart	Medicine ID: 101, Medicine Name: Paracetamol, Quantity: 10, Unit Price: 5	Cart Table: [101, Paracetamol, 10, 5, 50], Total Amount: 50	Cart Table: [101, Paracetamol, 10, 5, 50], Total Amount: 50	Pass
002	Updating Medicine in Cart	Select Medicine in Cart: 101, Update Quantity: 15	Cart Table: [101, Paracetamol, 15, 5, 75], Total Amount: 75	Cart Table: [101, Paracetamol, 15, 5, 70], Total Amount: 70	Fail
003	Deleting Medicine from Cart	Select Medicine in Cart: 101, Delete Medicine	Cart Table: [], Total Amount: 0	Cart Table: [], Total Amount: 0	Pass
004	Fetching Medicine ID	Database (Medicine table data)	Medicine ID Combo Box: [101, 102, 103]	Medicine ID Combo Box: [101, 104, 105]	Fail
005	Fetching Order ID	Database (Order table data)	Order ID: 10	Order ID: 11	Fail
006	Calculating Total Amount	Quantity: 10, Unit Price: 5	Total Amount: 50	Total Amount: 50	Pass
007	Inserting New Order	Customer ID: 1, Order Date: 2023-06-10, Net Amount: 100, Cart: [[101, Paracetamol, 10, 5, 50]]	Success Message: "New Order's Details Added Successfully"	Success Message: "New Order's Details Added Successfully"	Pass
008	Searching Order by ID	Search Text: "10"	Orders Table: [[10, 1, John, 2023-06-10, 100, Completed]]	Orders Table: [[10, 1, John, 2023-06-10, 100, Completed]]	Pass

7) Sales and Billing Management (IT22310996 – THENNAKOON T.A.C.S)

Table 2.7: Sales and Billing Test Cases

Test Case ID	Test Case Name	Input Data	Expected Output	Actual Result	Pass/Fail
Bill_01	Select the Order to pay	Select Order ID: 29	Display the selected order's total amount to pay	Display the selected order's total amount to pay	Pass
Bill_02	Update the customer's paid amount	The paid Amount: LKR 12000.00	Display the updated amount and pop-up message "Updated Paid amount successfully"	Change the paid amount of the selected bill and update the balance at the same time	Pass
Bill_03	Delete the customer's Bill	Select the bill in the Billing table	Display pop-up message "Deleted the bill successfully"	Remove the whole billing details.	Pass
Bill_04	Clear the billing data	Click clear button	Clear text fields and select a new bill	Clear text fields and select a new bill	Pass
Bill_05	Insert the customer's bill without entering the paid amount	The paid amount is null.	Display pop-up message "Invalid Paid amount input. Please enter the valid numbers"	Display pop-up message "Invalid Paid amount input. Please enter the valid numbers"	Pass
Bill_06	Download report	Select month and year	Display selected monthly revenue with the year	Display selected monthly revenue with the selected year	Pass

3. Evaluation

3.1 Assessment of the Project results

❖ **Splash Form**

Figure 3.1: Splash Form UI

❖ **Dashboard**

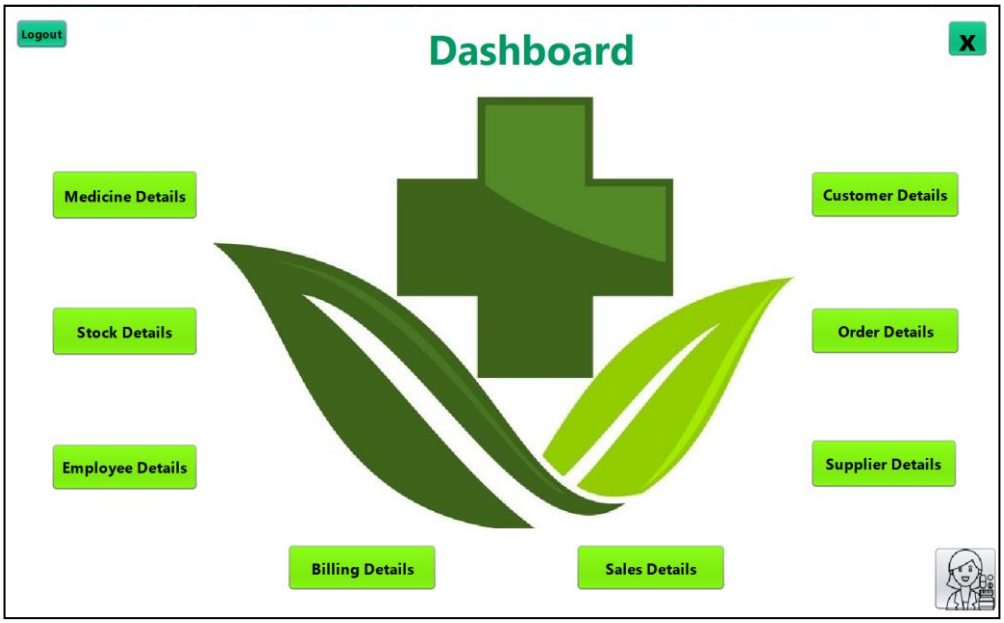

Figure 3.2: Dashboard UI

❖ Medicine Inventory Management – (IT22364692 - KANDAGE K.T.S.)

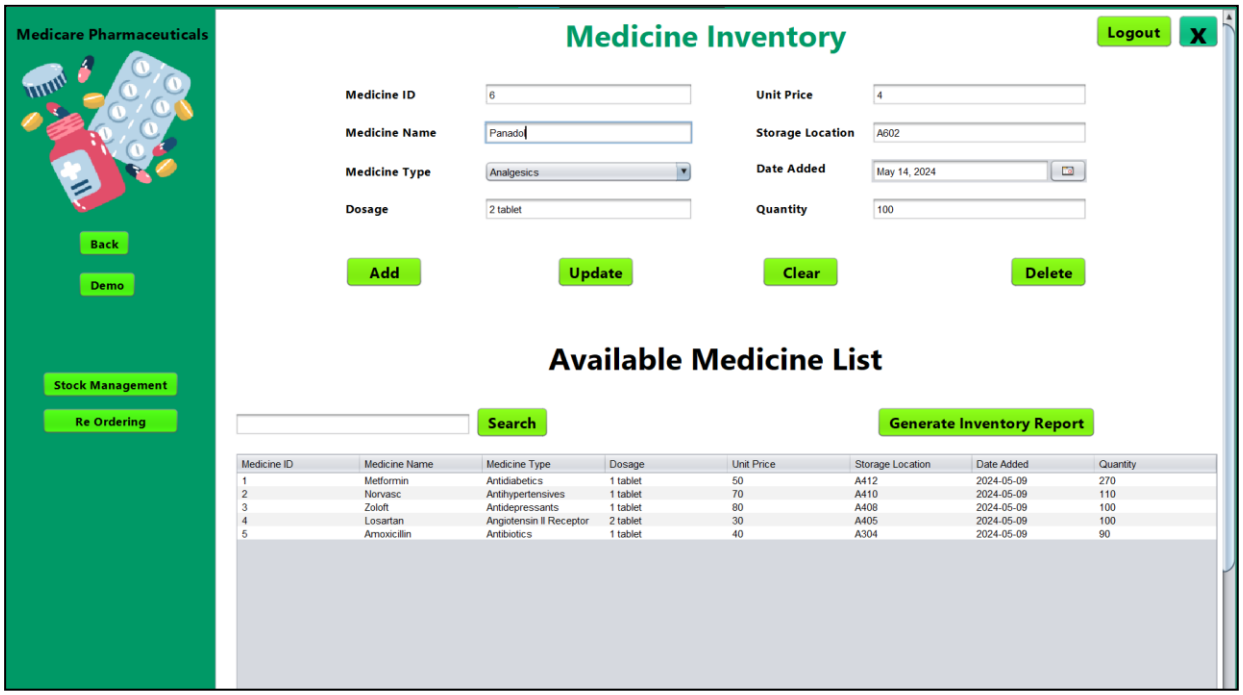

Figure 3.3: Medicine Inventory UI

- **Insert New Medicines**

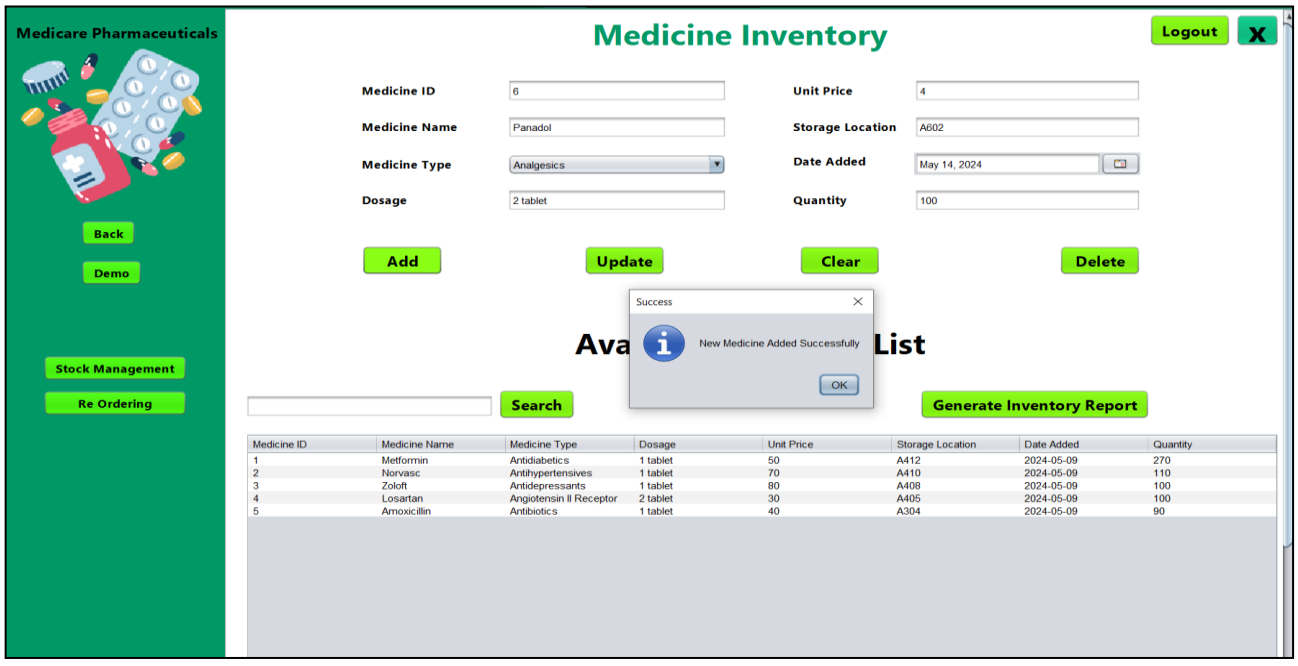

Figure 3.4: Medicine inventory Adding Details

- **Update Medicine Data**

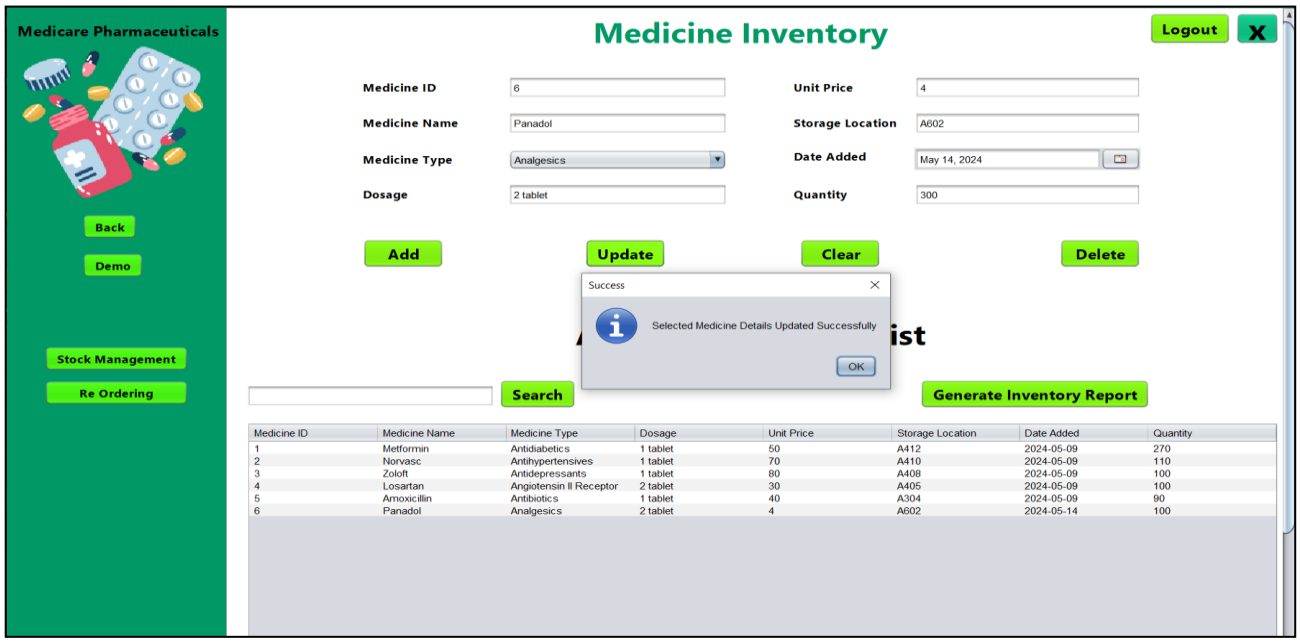

Figure 3.5: Medicine inventory Update Details

- **Delete Medicine Data**

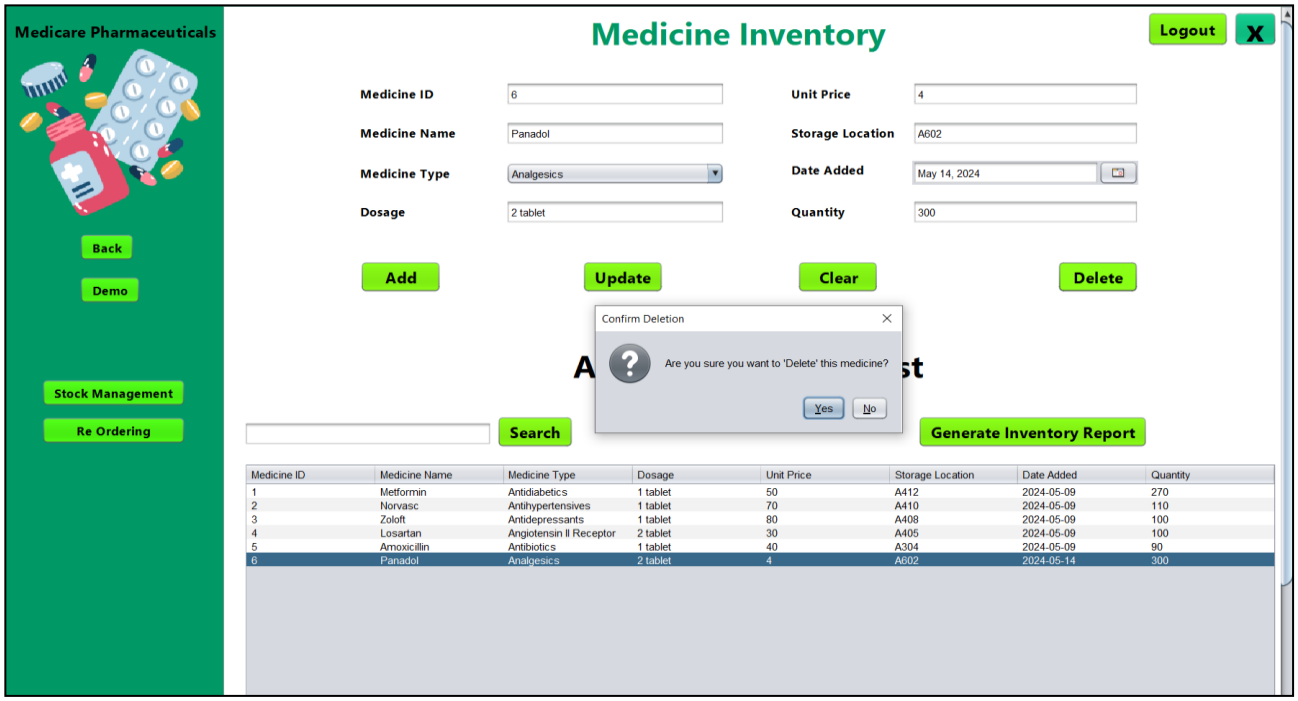

Figure 3.6: Medicine inventory Delete Details

- **Generate Inventory Report**

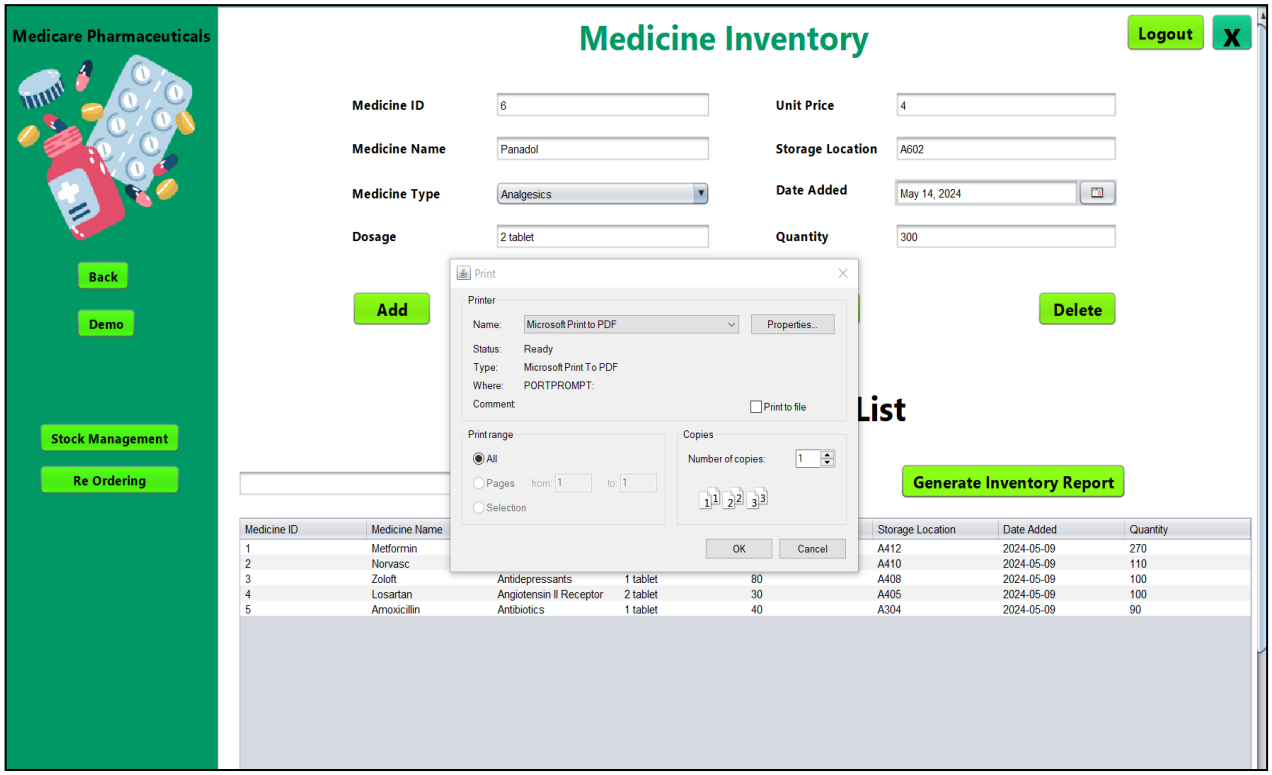

Figure 3.7: Medicine inventory Report Generation

- **Inventory Report**

1	Metformin	Antidiabetics	1 tablet	50
2	Norvasc	Antihypertensives	1 tablet	70
3	Zoloft	Antidepressants	1 tablet	80
4	Losartan	AngiotensinIIReceptor	2 tablet	30
5	Amoxicillin	Antibiotics	1 tablet	40

Figure 3.8: Medicine inventory report

❖ Medicine Stock Management – (IT22364692 - KANDAGE K.T.S.)

- **Add a New Stock**

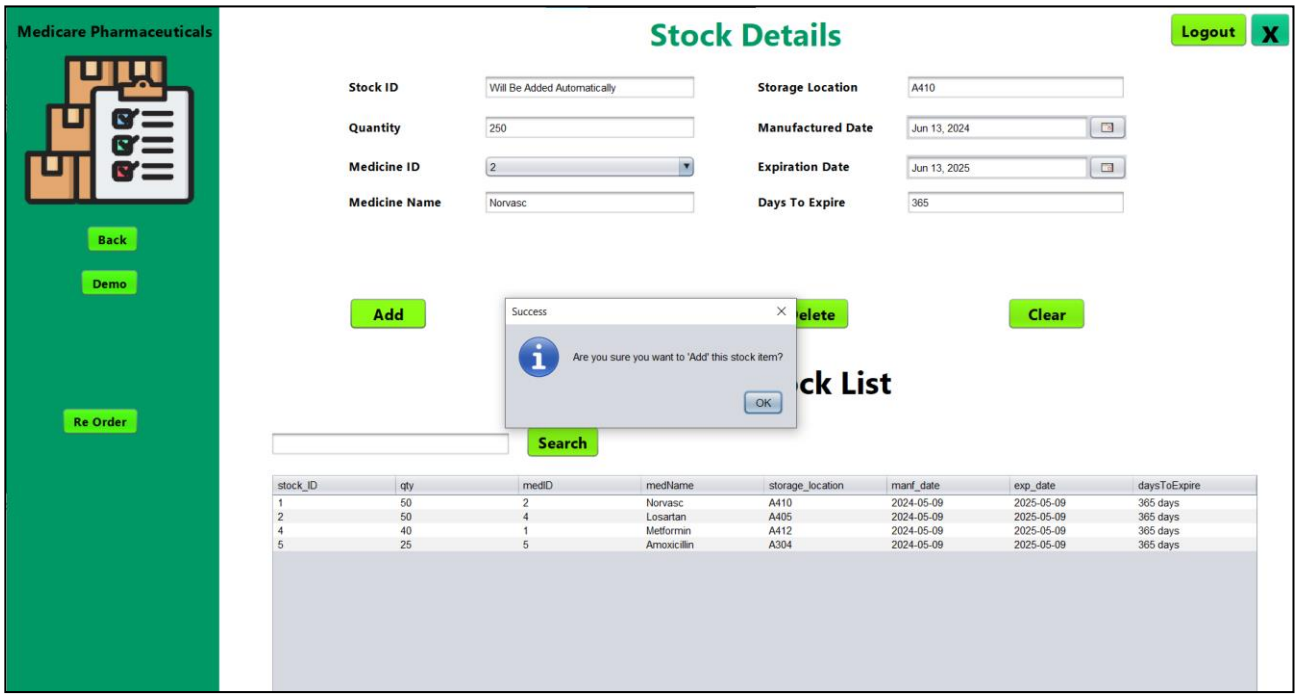

Figure 3.9: Stock details UI

- **Update Stock Details**

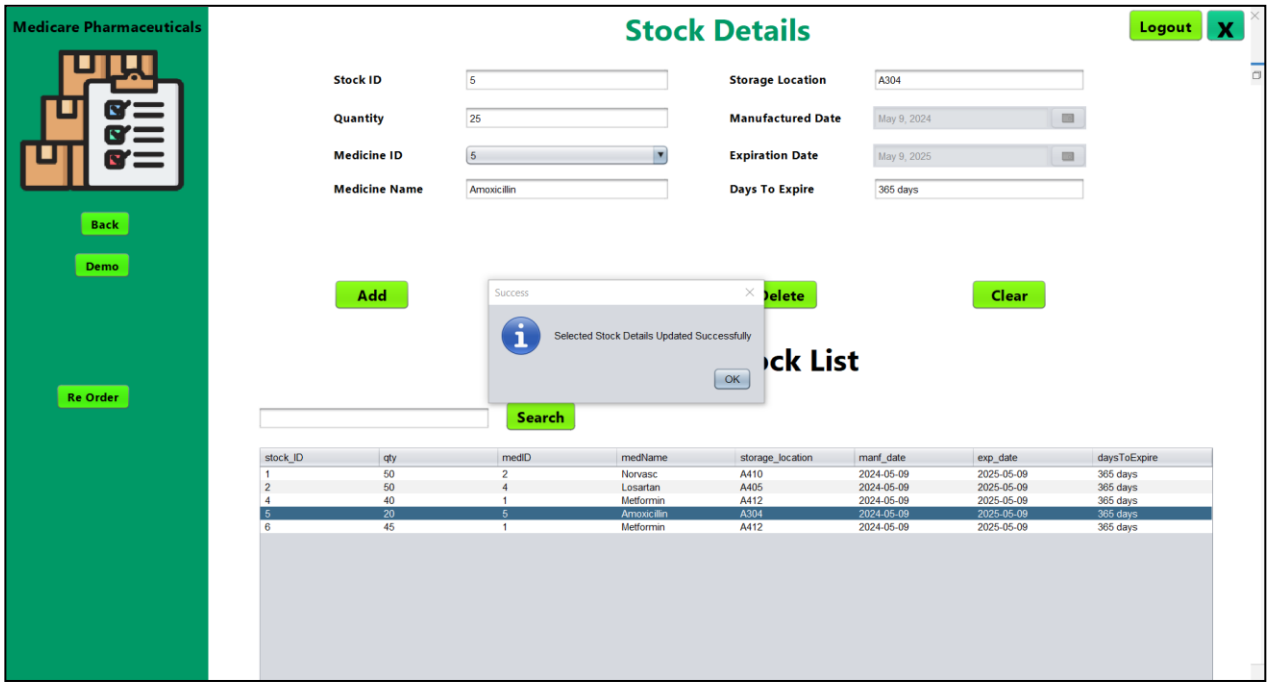

Figure 3.10: Stock details update

- **Delete Stocks**

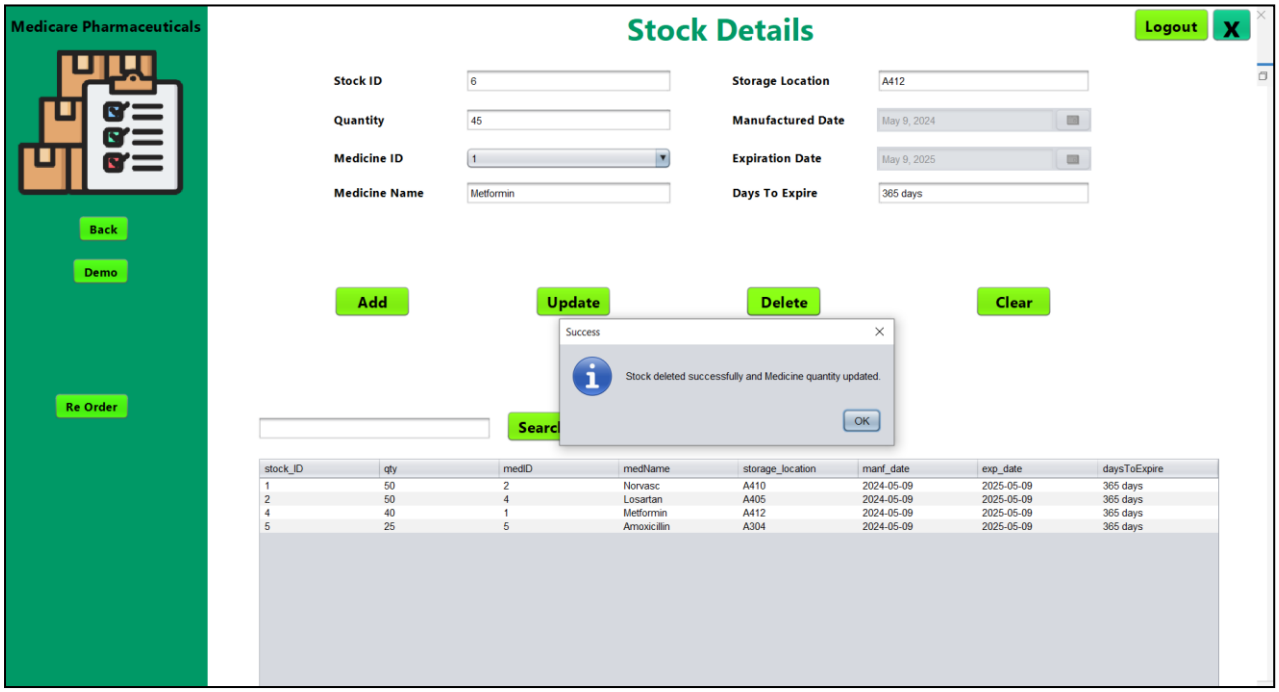

Figure 3.11: Stock details Delete

- **Re-Ordering Stocks –** When the Stock Level reach to the "Minimum Stock Level" reorder stocks by sending an Email to the specific Supplier

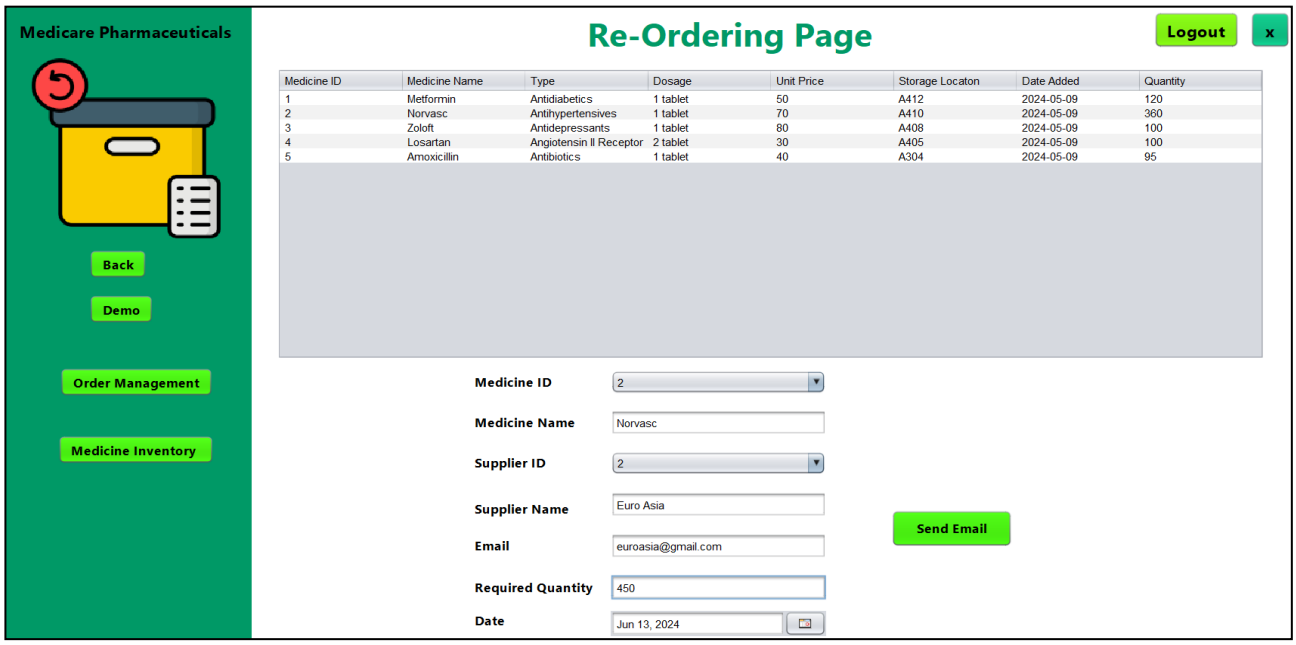

Figure 3.12: Stock Re-Ordering Page UI

❖ Supplier Management – (IT22319142 – WIJESINGHE A.G.T)

- **Register a New Supplier**

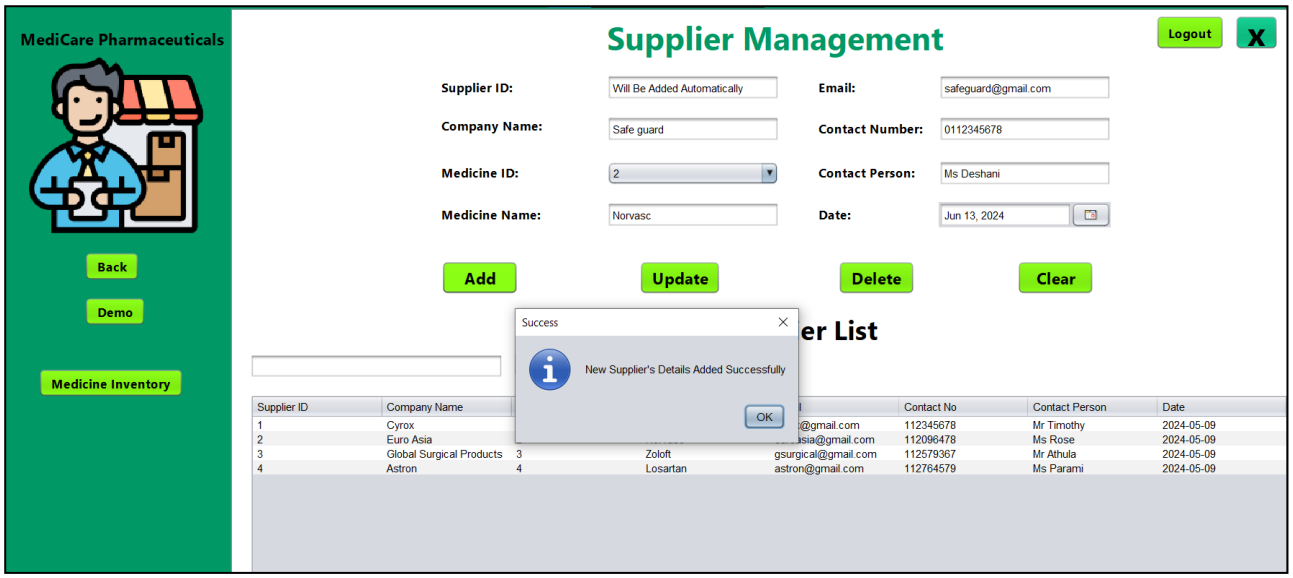

Figure 3.13: Add New Supplier

- **Update Supplier Details**

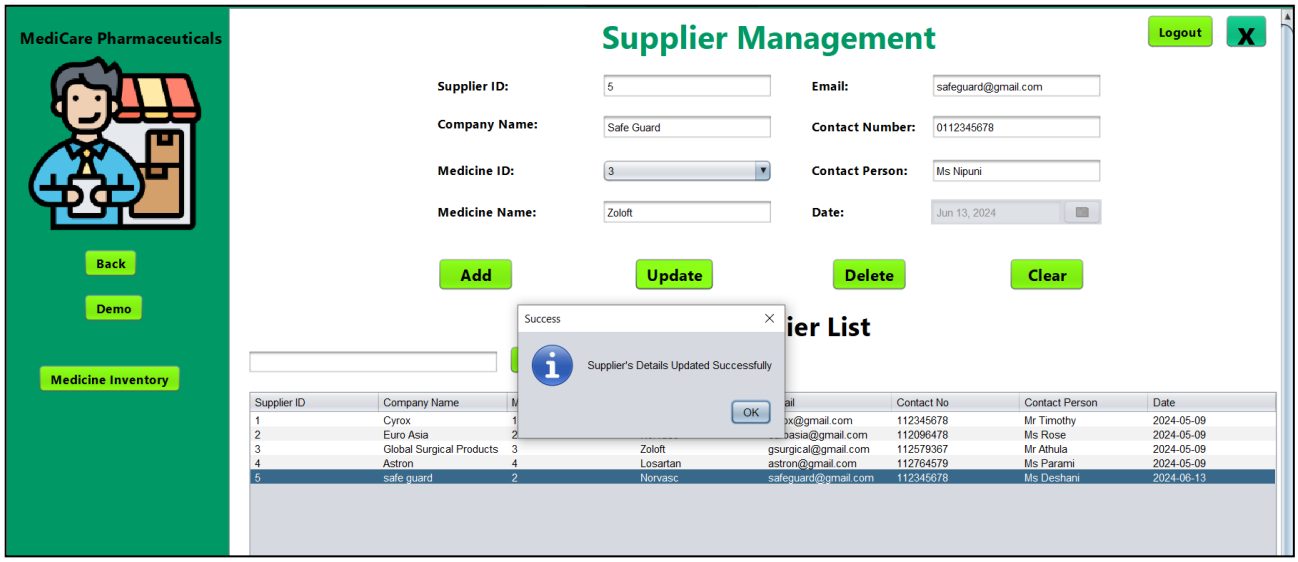

Figure 3.14: Modify Supplier Details

- **Remove a Supplier from the system database**

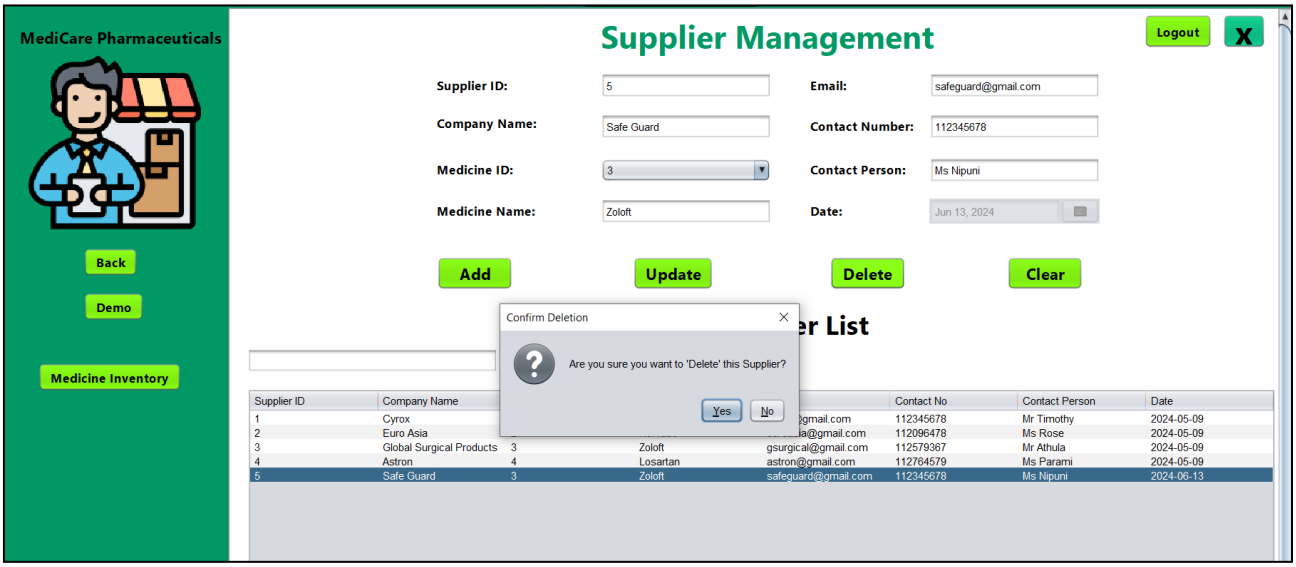

Figure 3.15: Delete a Supplier Record

- **Search for a Supplier**

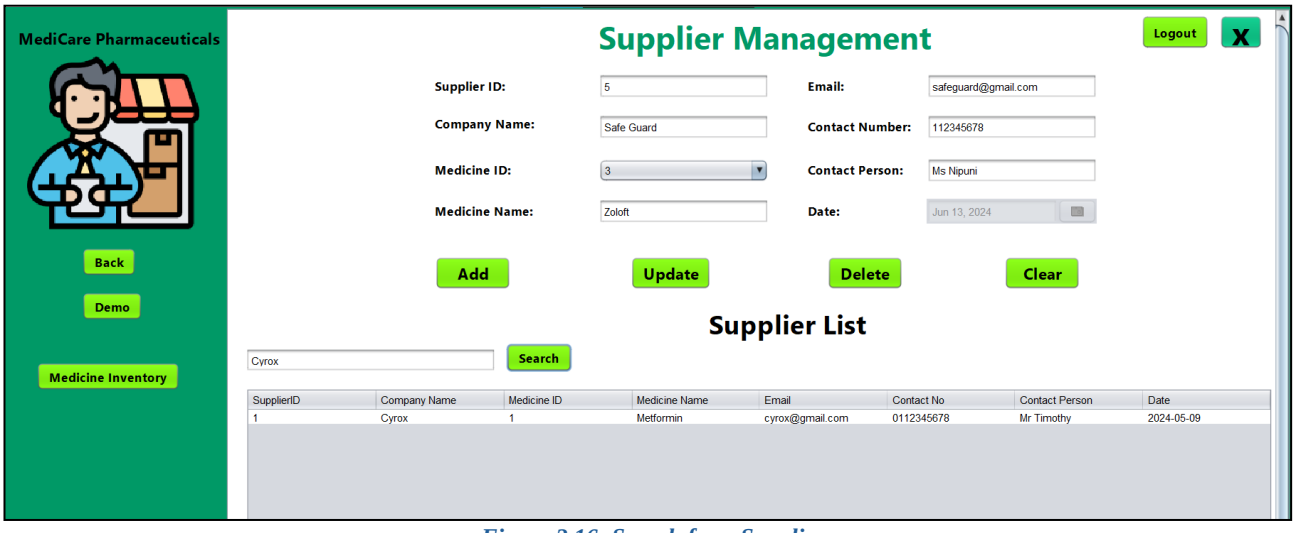

Figure 3.16: Search for a Supplier

❖ Employee Management – (IT22319142 – WIJESINGHE A.G.T)

- **Register a New Employee**

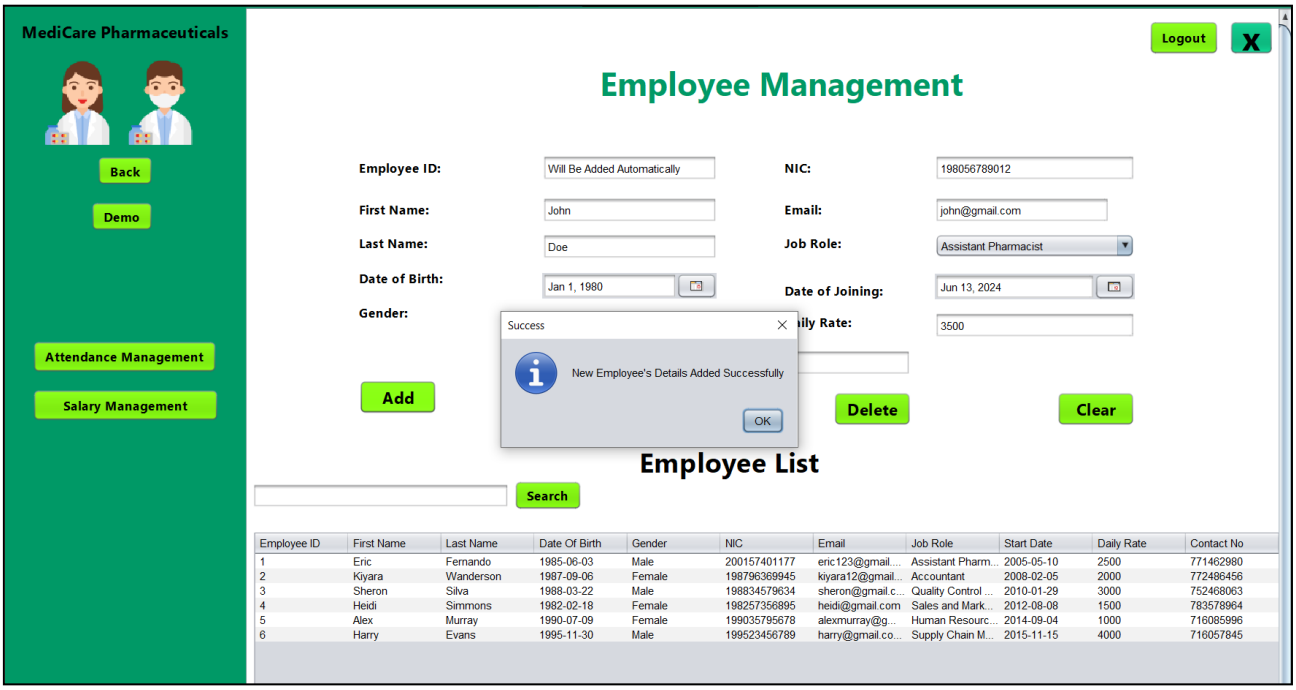

Figure 3.17: Add New Employee

- **Update the details of a Registered Employee**

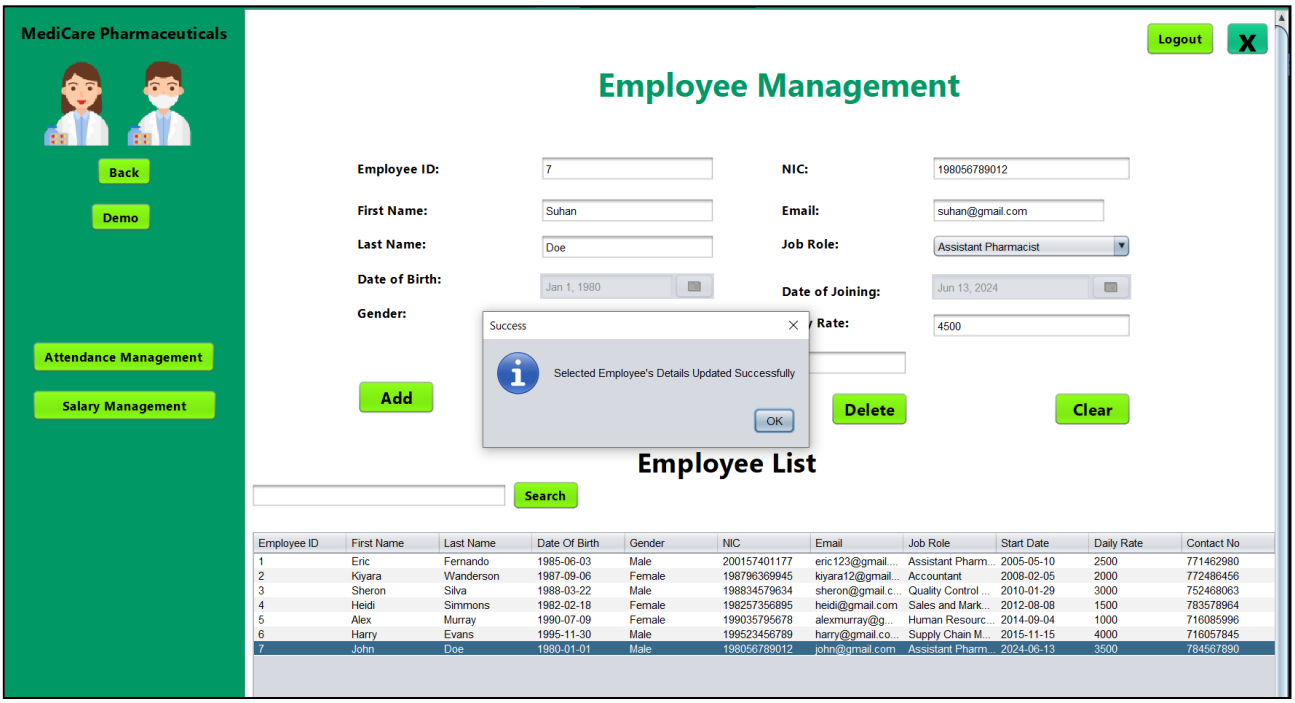

Figure 3.18: Update a Registered Employee

- **Delete a record of a Registered Employee**

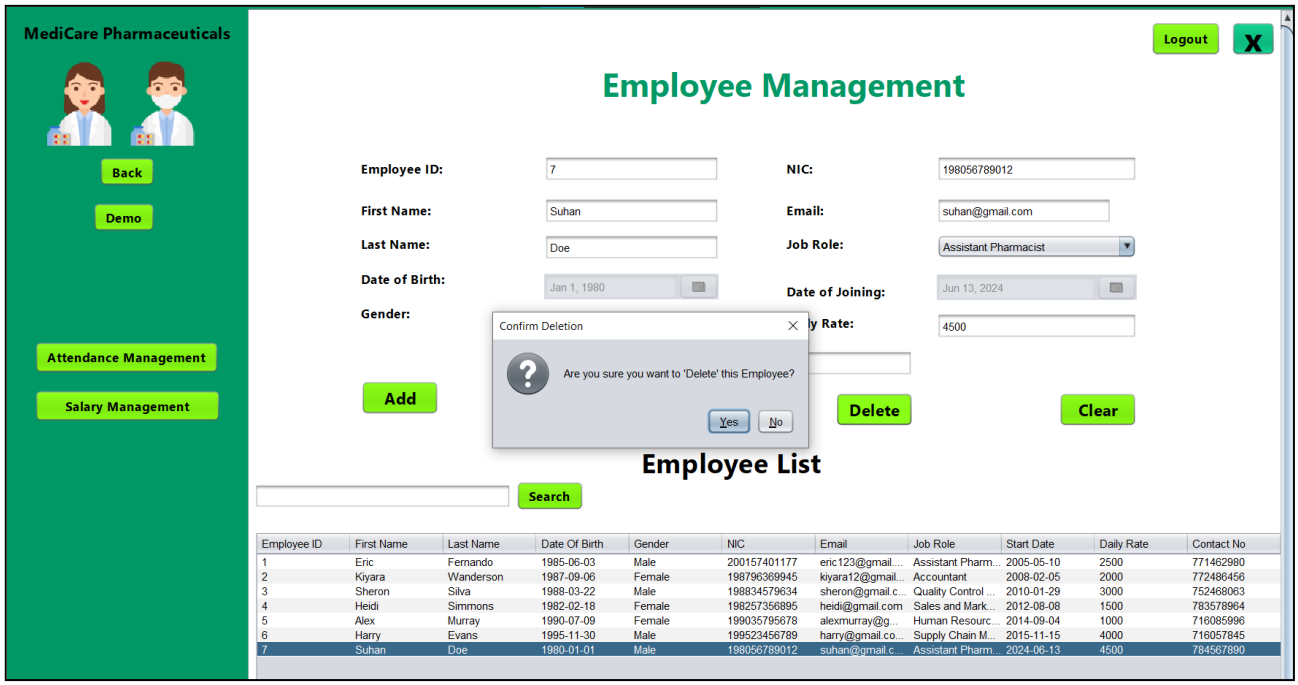

Figure 3.19: Delete a Registered Employee

- **Search a Registered Employee**

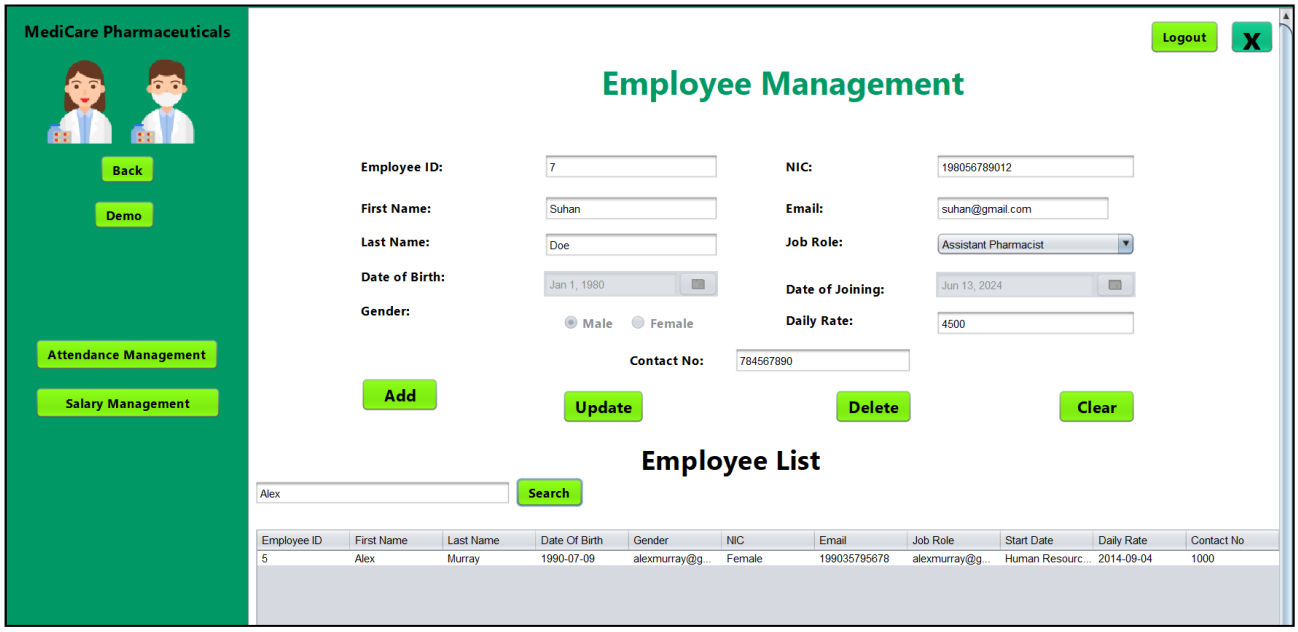

Figure 3.20: Search details of a Registered Employee

❖ Employee Daily Attendance Tracking – (IT22319142 – WIJESINGHE A.G.T)

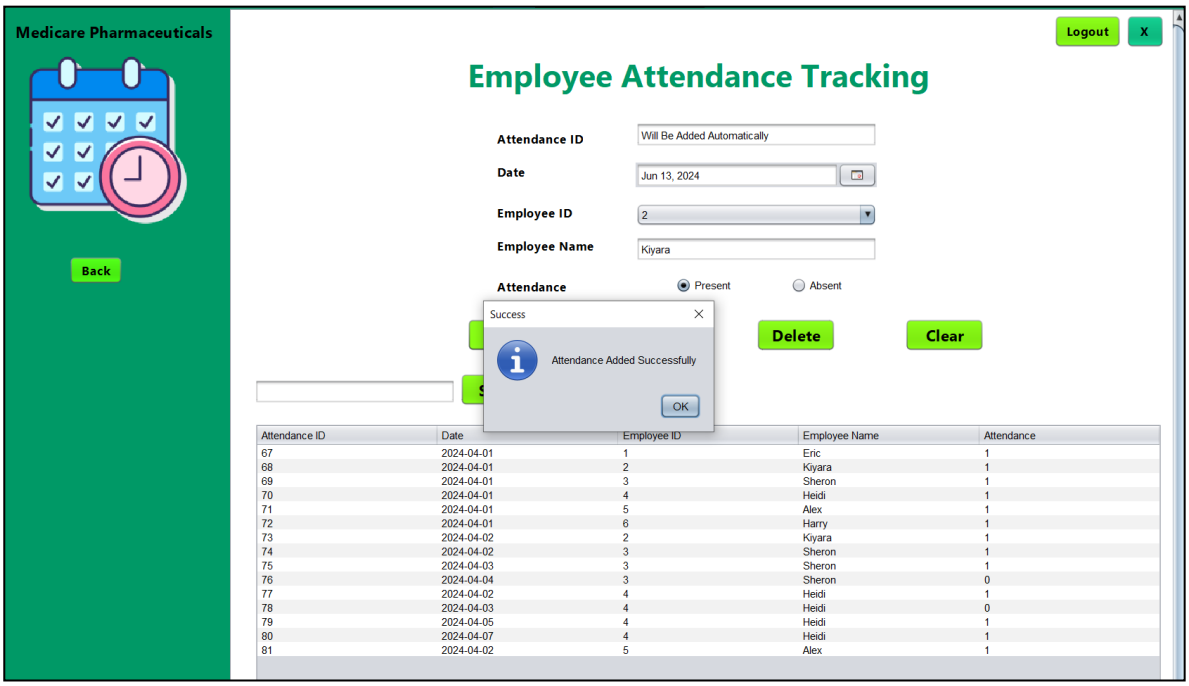

Figure 3.21: Insert a New Attendance Record

- **Error pop ups when try to mark attendance twice in a same day (for a single employee)**

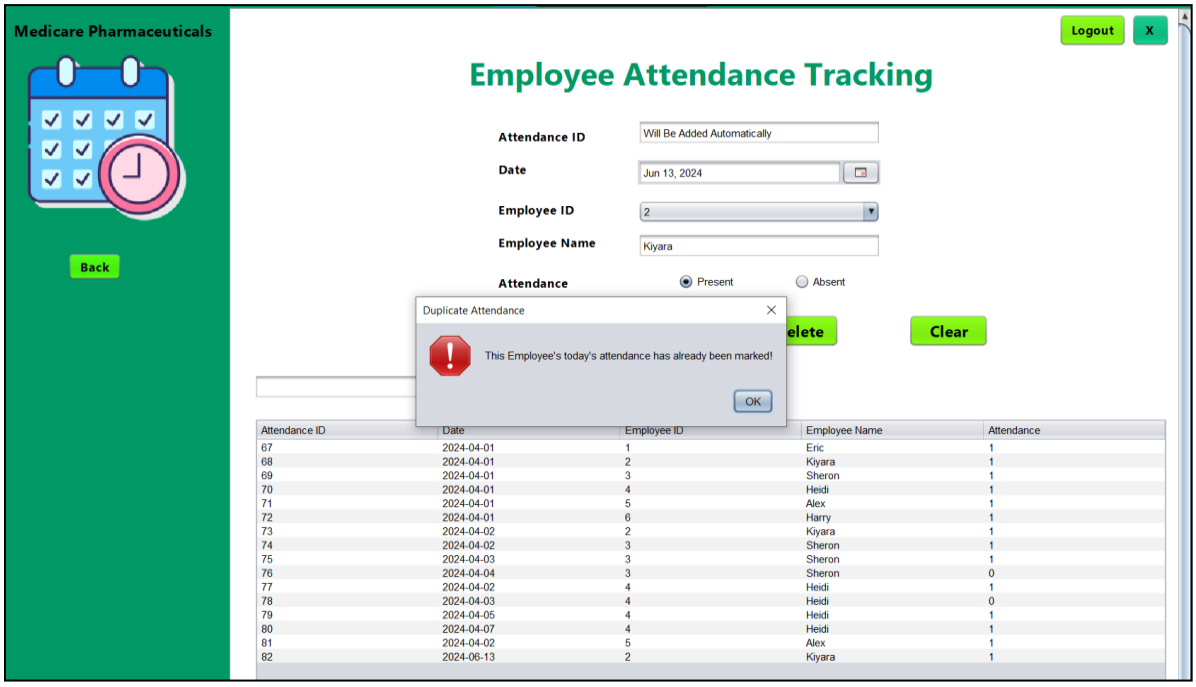

Figure 3.22: Error message for Attendance Duplication

- **Update the Attendance Status (Absent/Present)**

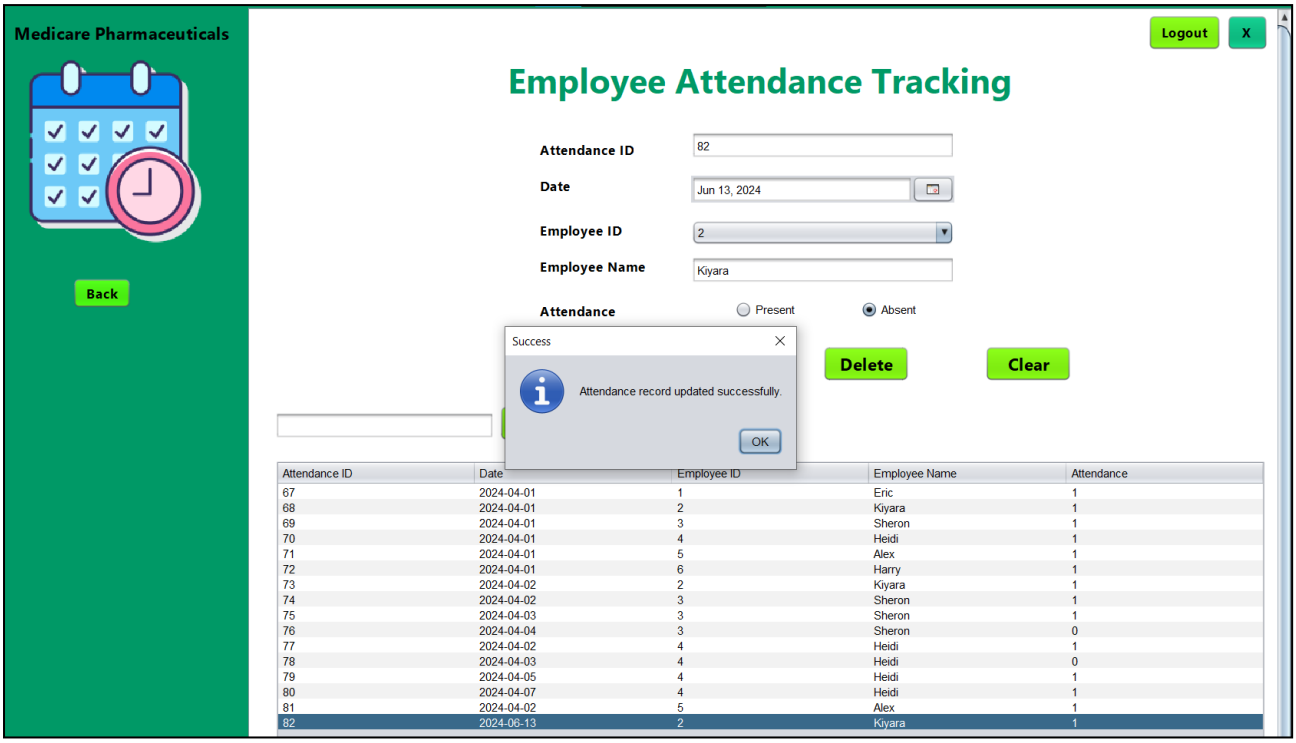

Figure 3.23: Updating the Attendance Status of an Employee

- **Deleting an Attendance Record**

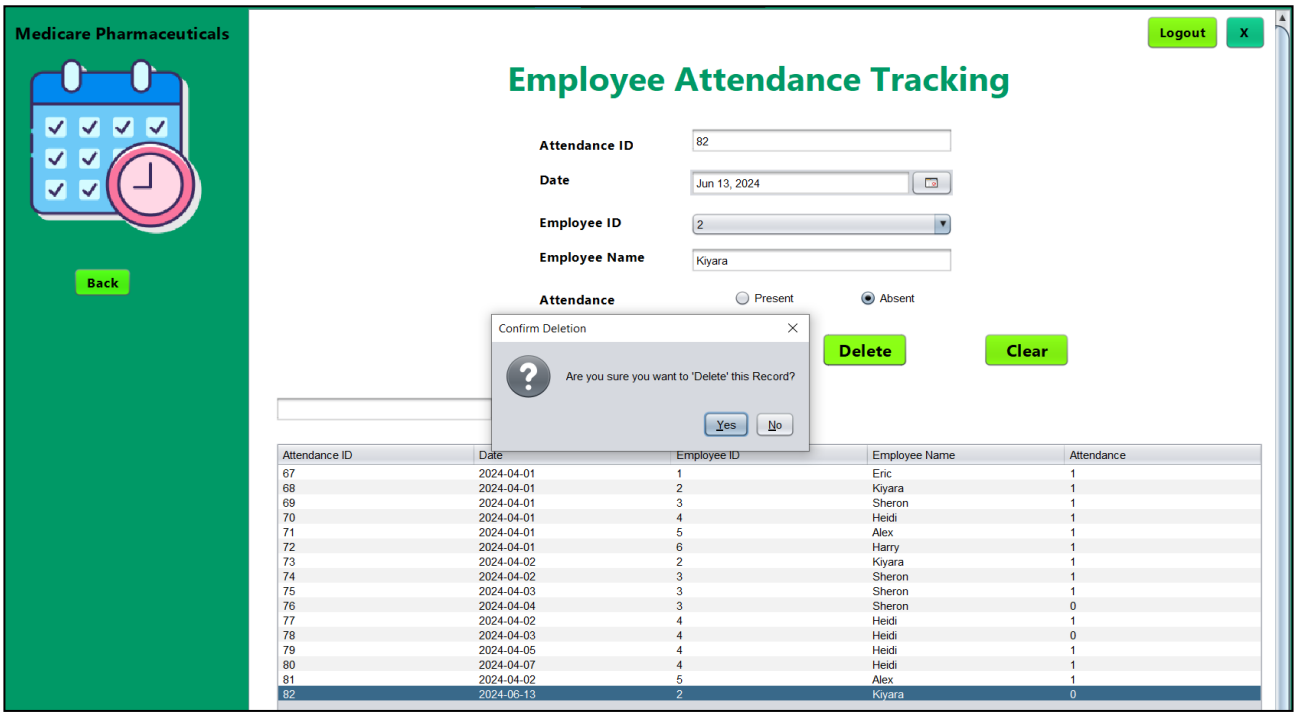

Figure 3.24: Deleting an Attendance Record

- **Search Attendance of an Employee**

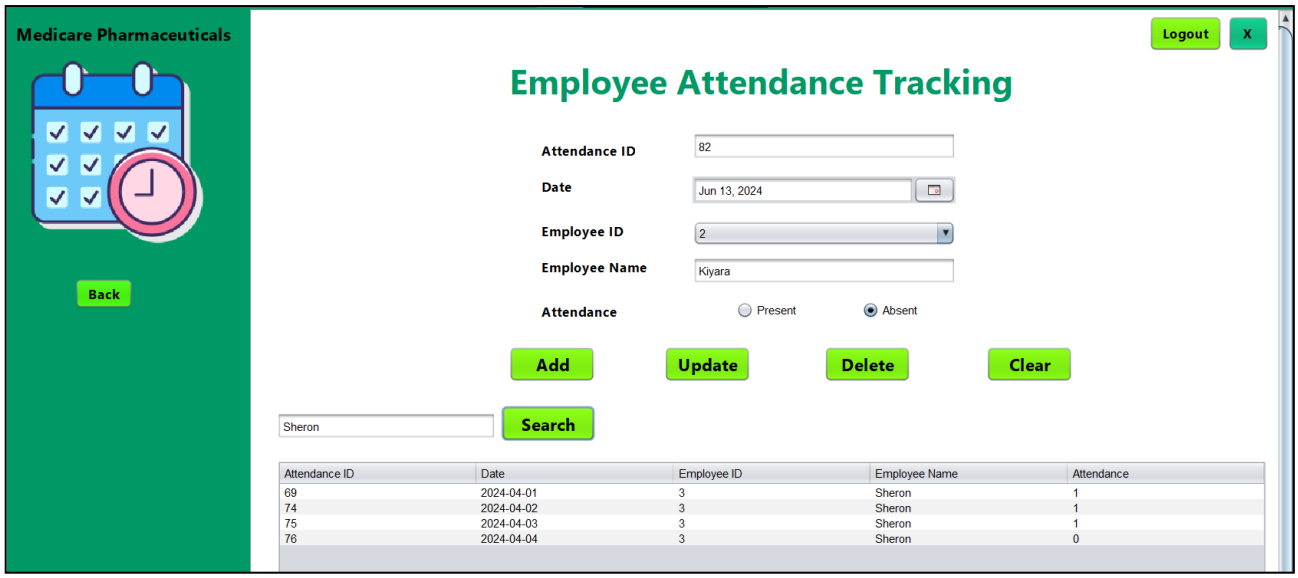

Figure 3.25: Searching Attendance of an Employee

❖ Employee Monthly Salary Calculation – (IT22319142 – WIJESINGHE A.G.T)

- **Add New Salary Record**

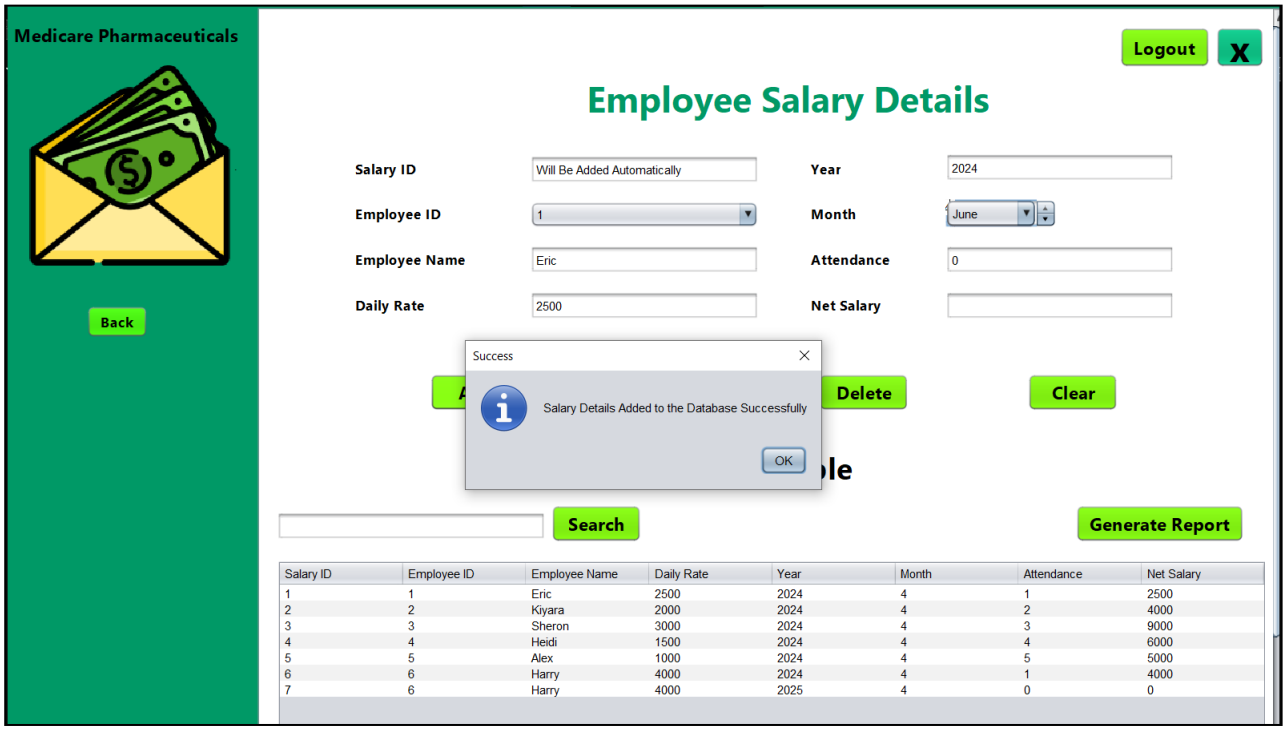

Figure 3.26: Add New Salary Record

- **Error pop ups when try to add a monthly salary record that is already added**

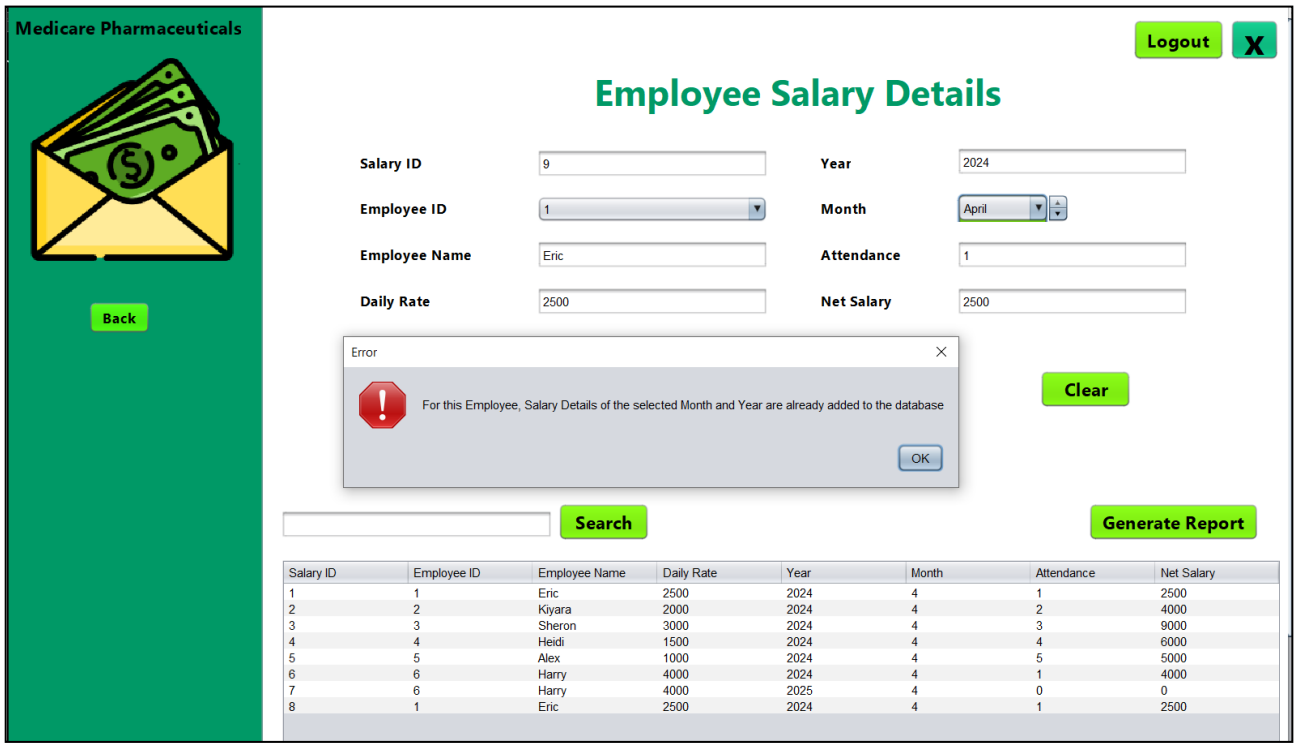

Figure 3.27: Add New Salary Record

- **Update a monthly salary record**

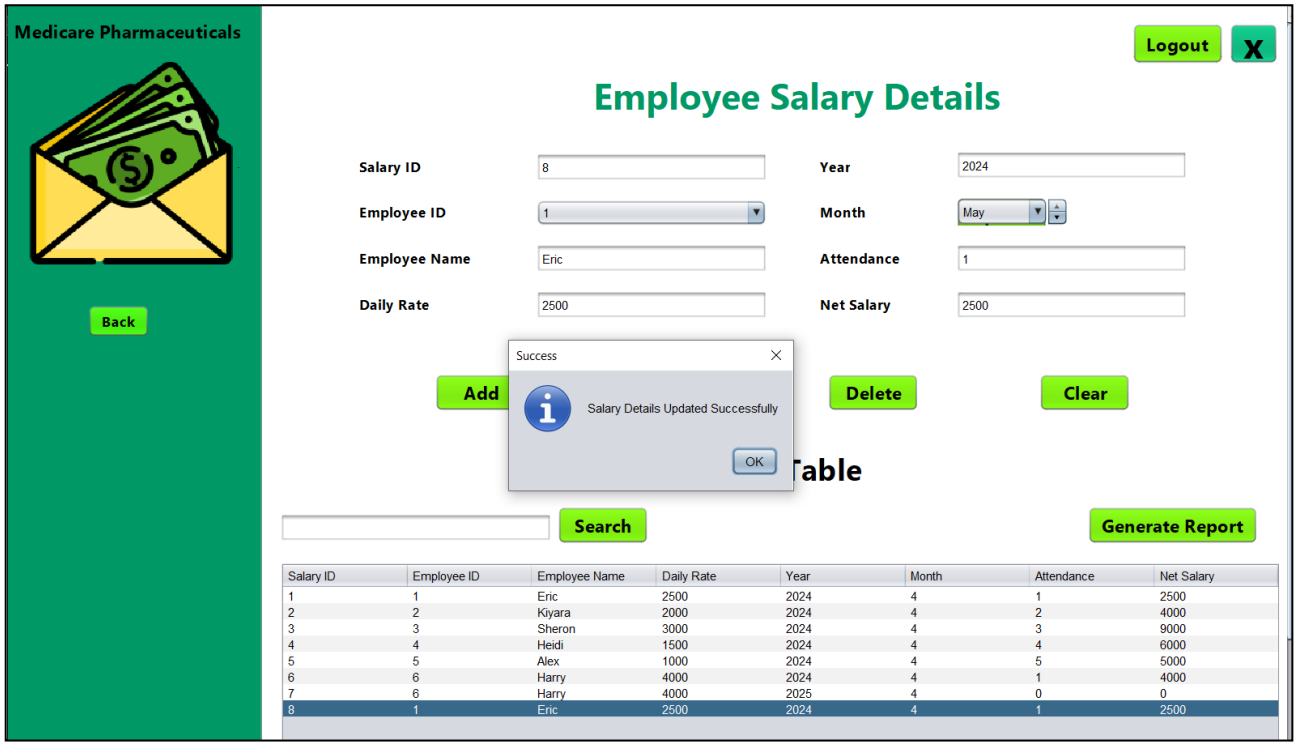

Figure 3.28: Update a Salary Record

- **Delete a monthly salary record**

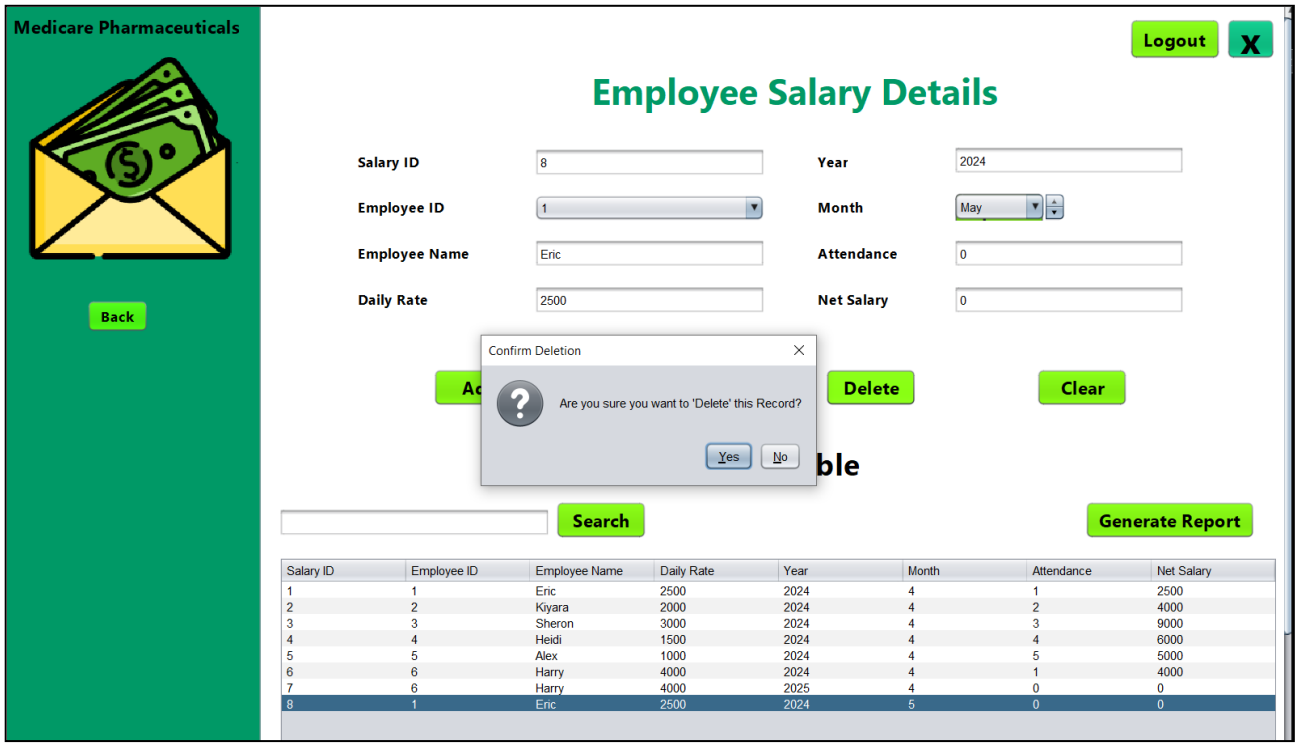

Figure 2.29: Delete a Salary Record

- **Search for a monthly salary records of a Specific Employee**

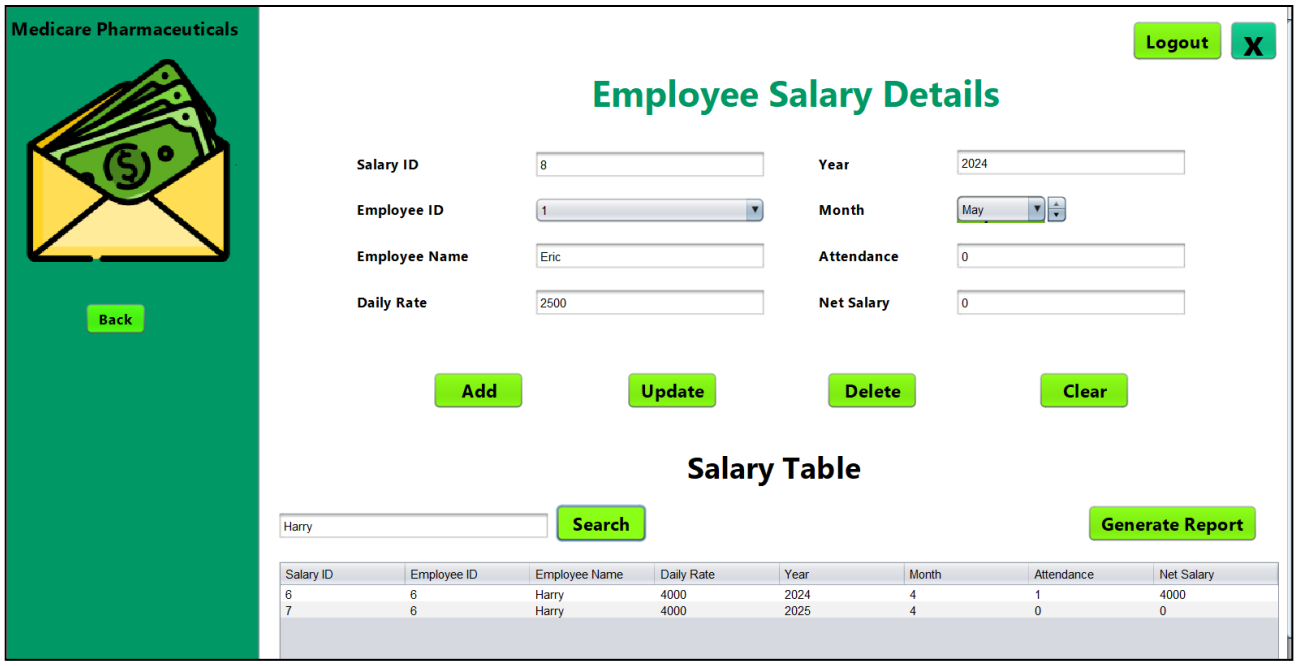

Figure 2.30: Search Salary Records of an Employee

- **Generate the Attendance & Monthly Salary Reports**

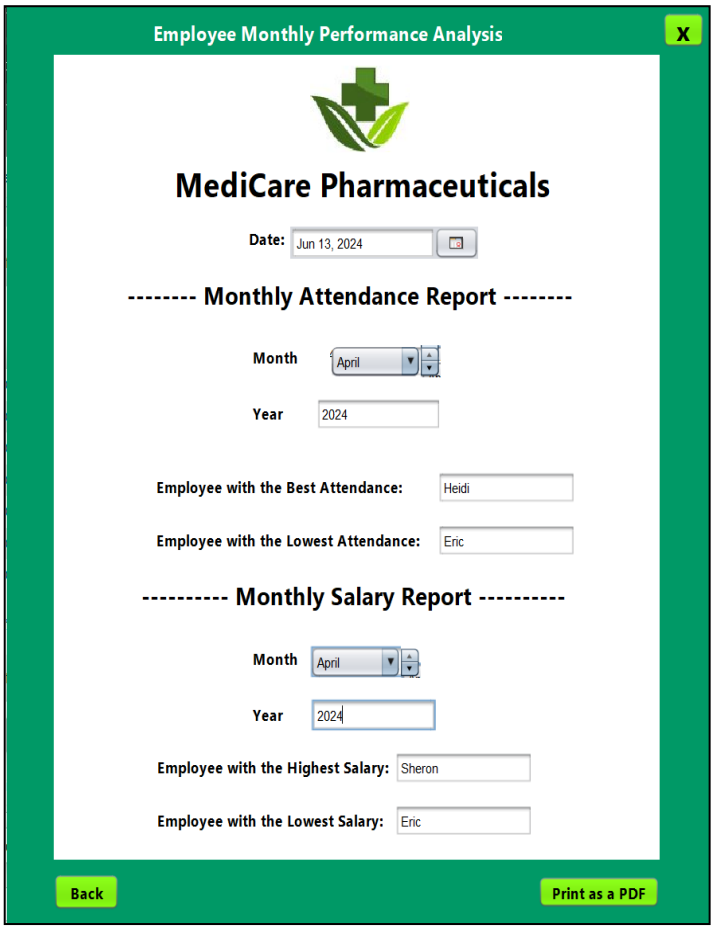

Figure 2.31: Monthly Salary & Attendance Reports Generation

❖ Customer Management – (IT22884138 – RATHNAYAKA R.M.T.D)

- **Add details of a New Customer**

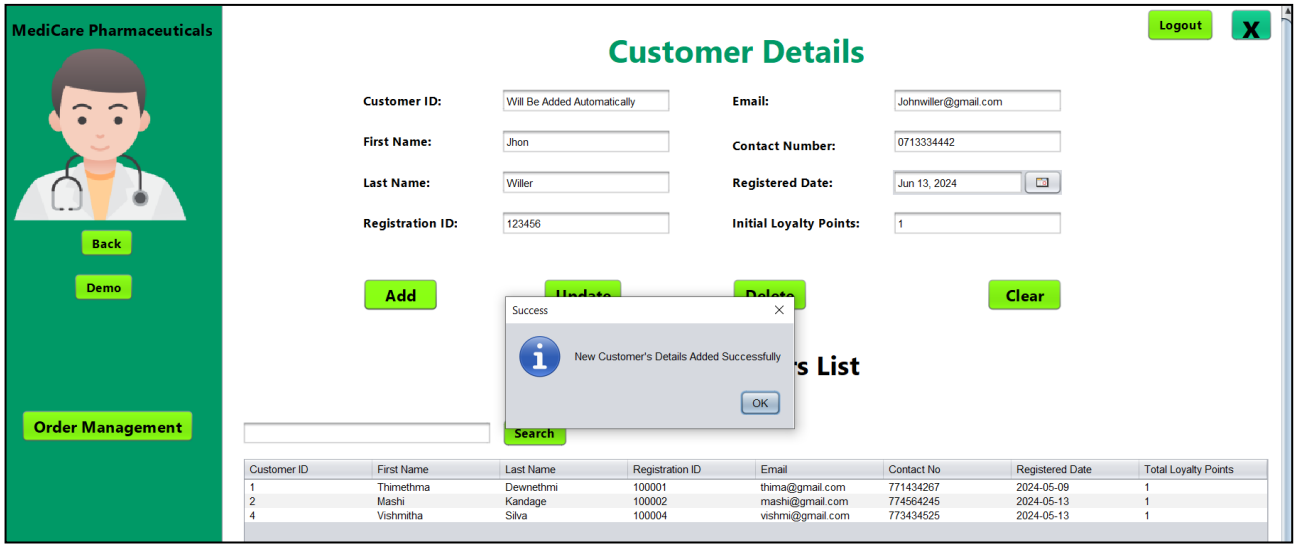

Figure 2.32: Add New Customer

- **Update the details of a Registered Customer**

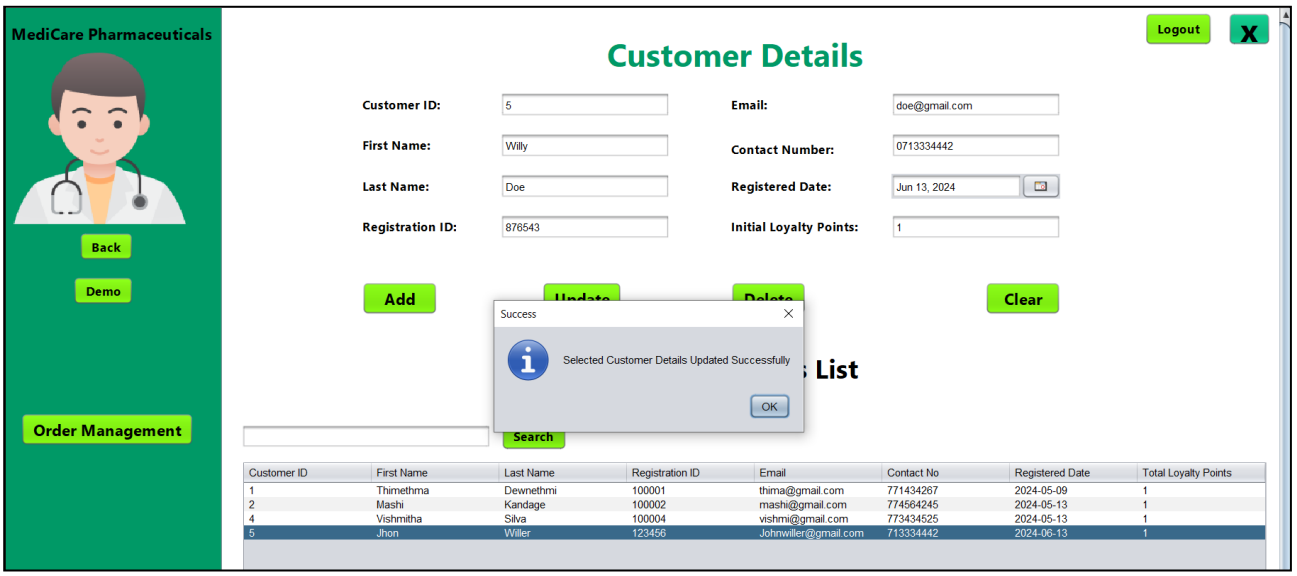

Figure 2.33: Update the details of a customer

- **Delete the details of a Registered Customer**

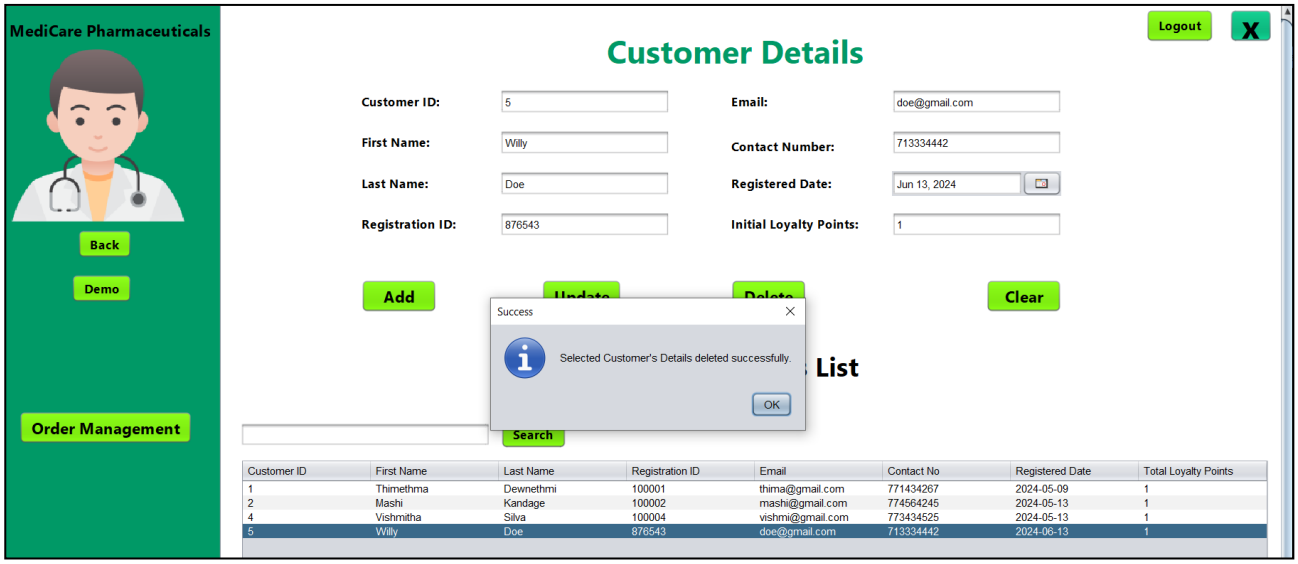

Figure 2.34: Delete a customer from system database

- **Search the details of a Registered Customer**

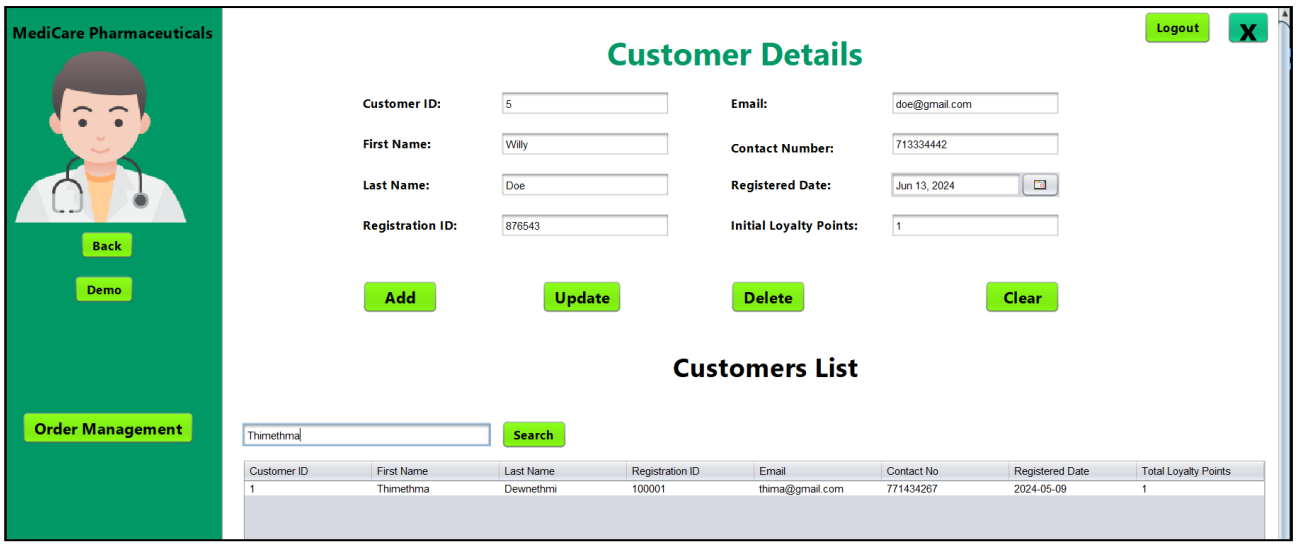

Figure 2.35: Search for a Registered Customer

❖ Order Management – (IT22884138 – RATHNAYAKA R.M.T.D)

- **Insert a New Order's details**

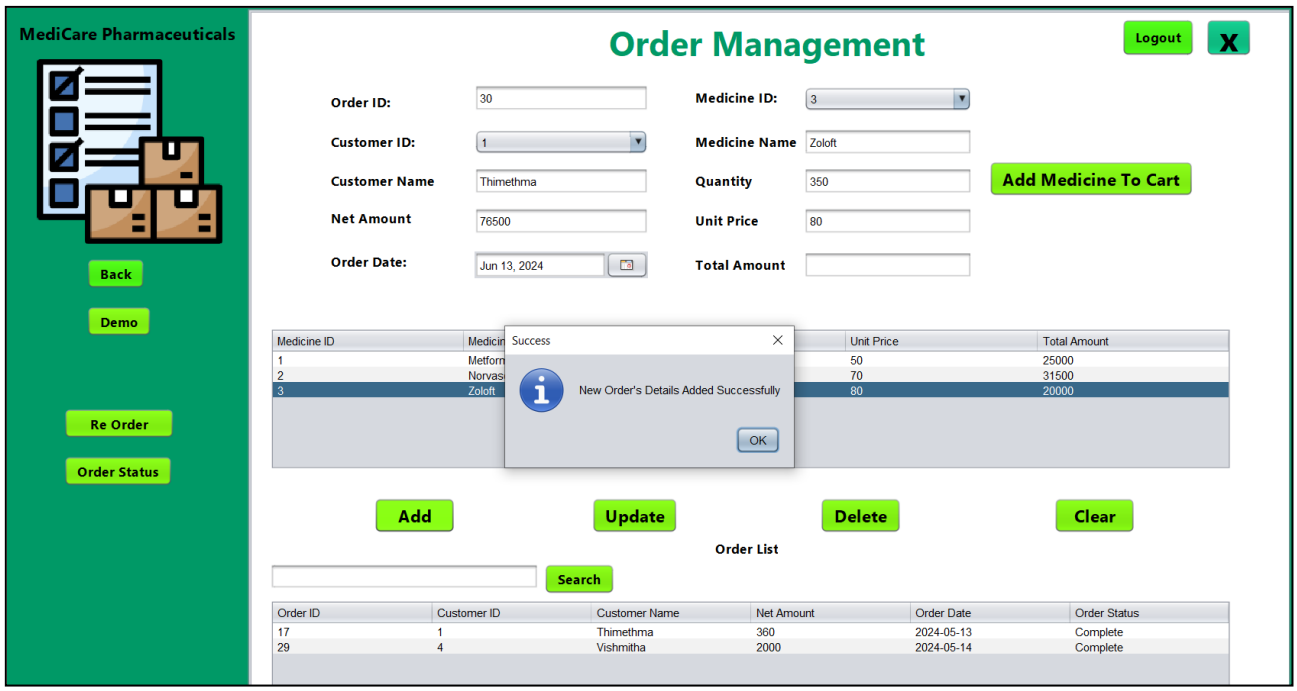

Figure 2.36: Add New Order

- **Update an Order's details**

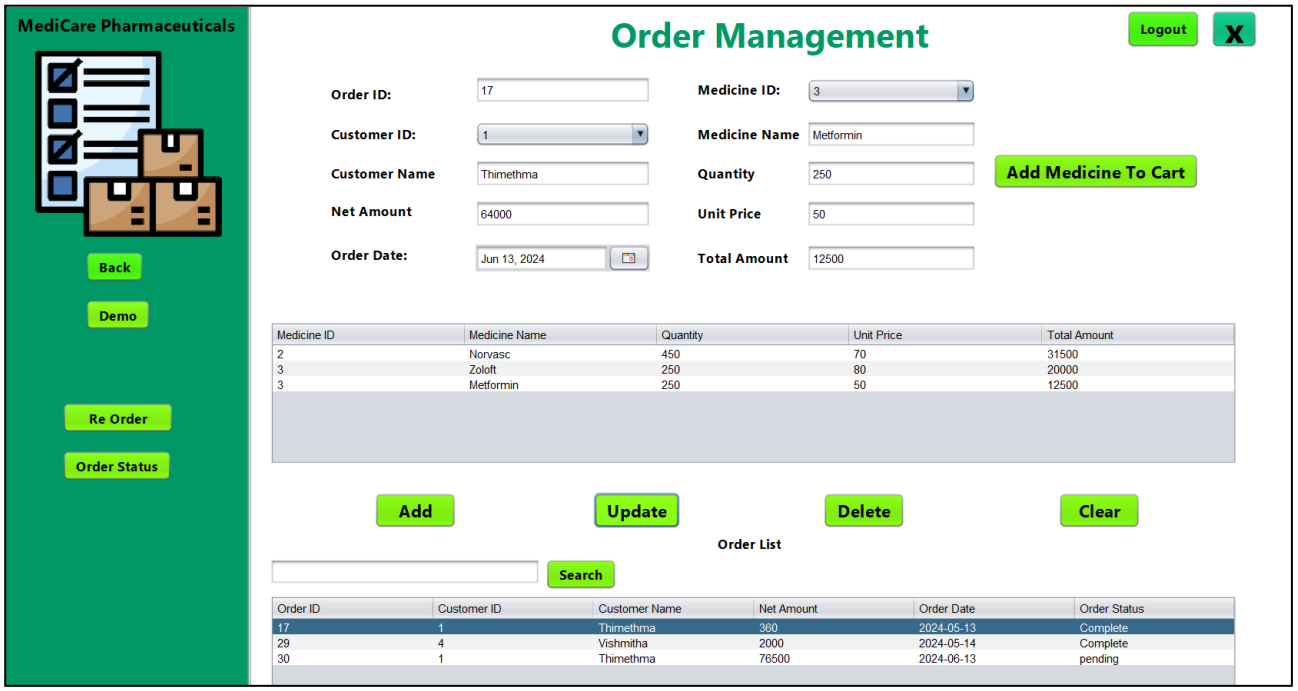

Figure 2.37: Modify Order Details

- **Delete an Order**

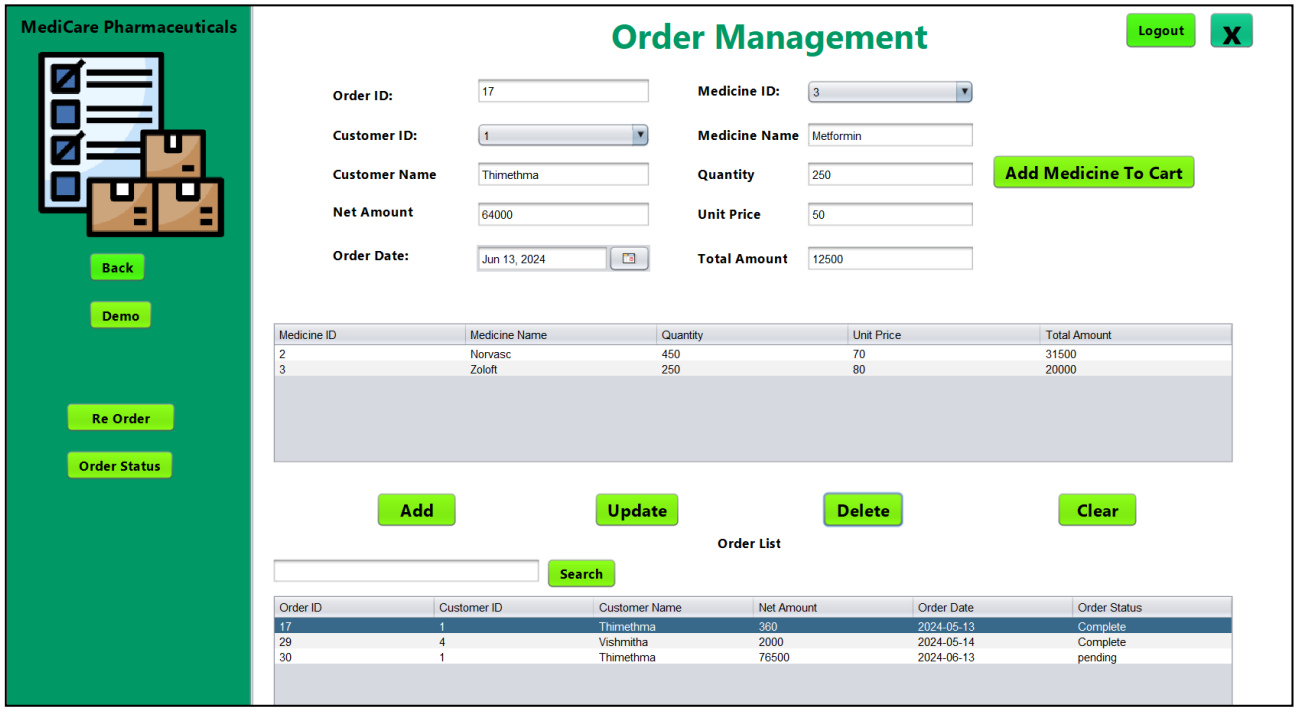

Figure 2.38: Delete an Order

- **Search an Order's details**

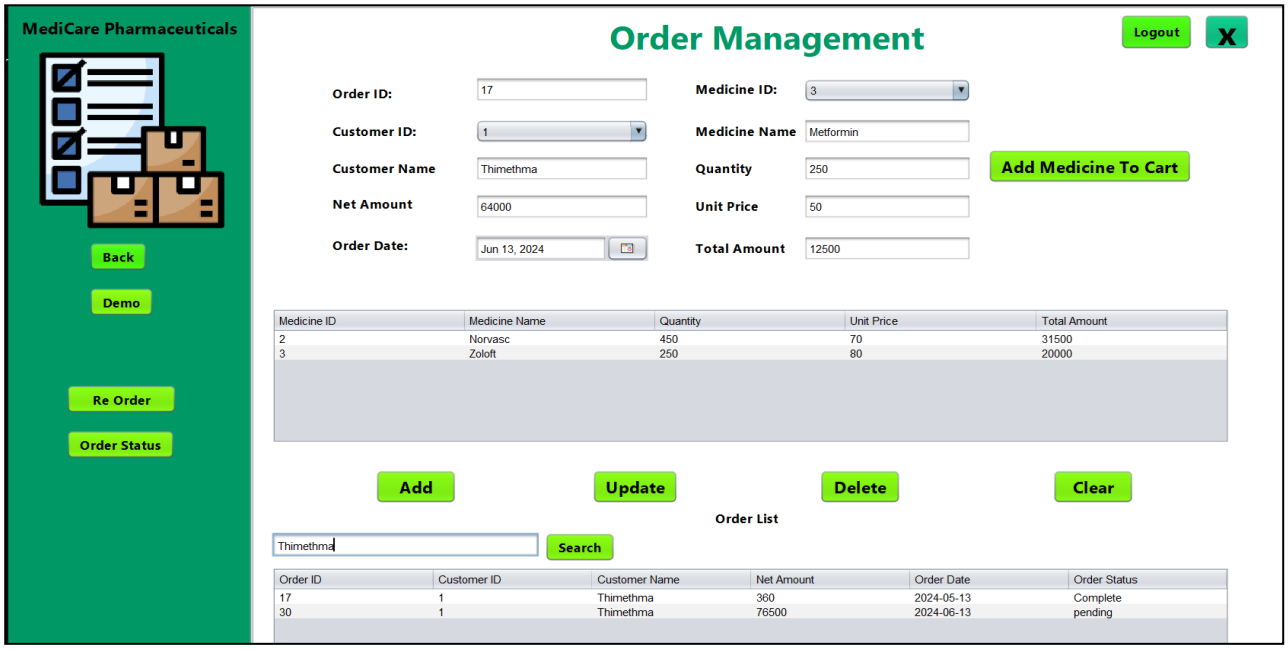

Figure 2.39: Search an Order

- **Order Status Page**

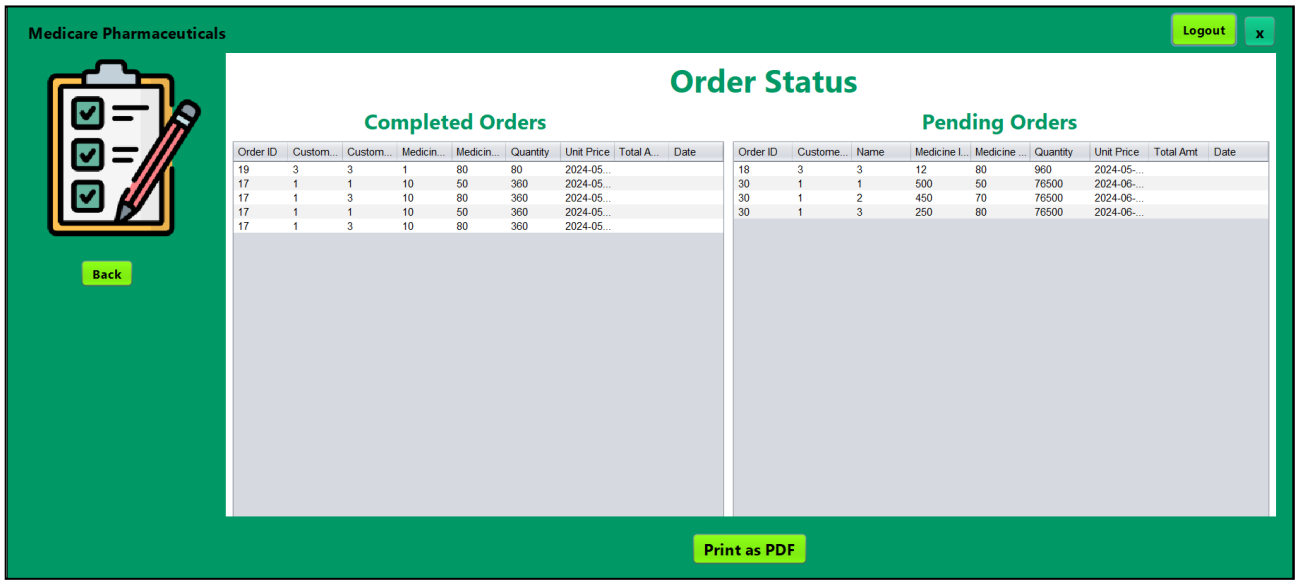

Figure 2.40: Order Status Page

- **Generate Order Status Report**

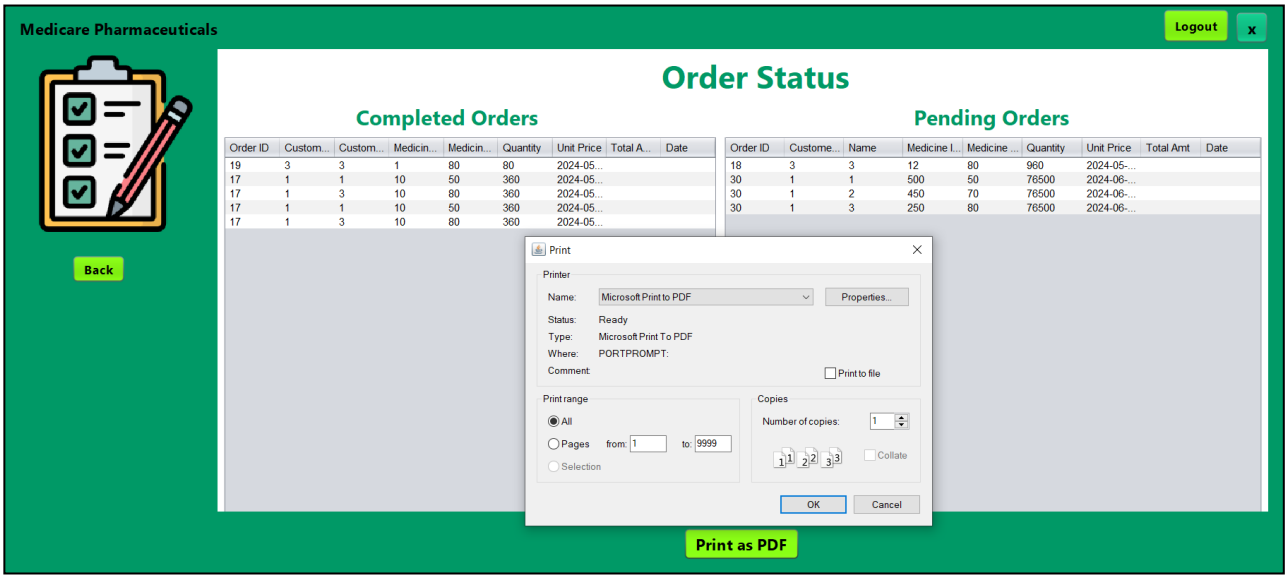

Figure 2.41: Order Status Report Generation

- **Order Status Report**

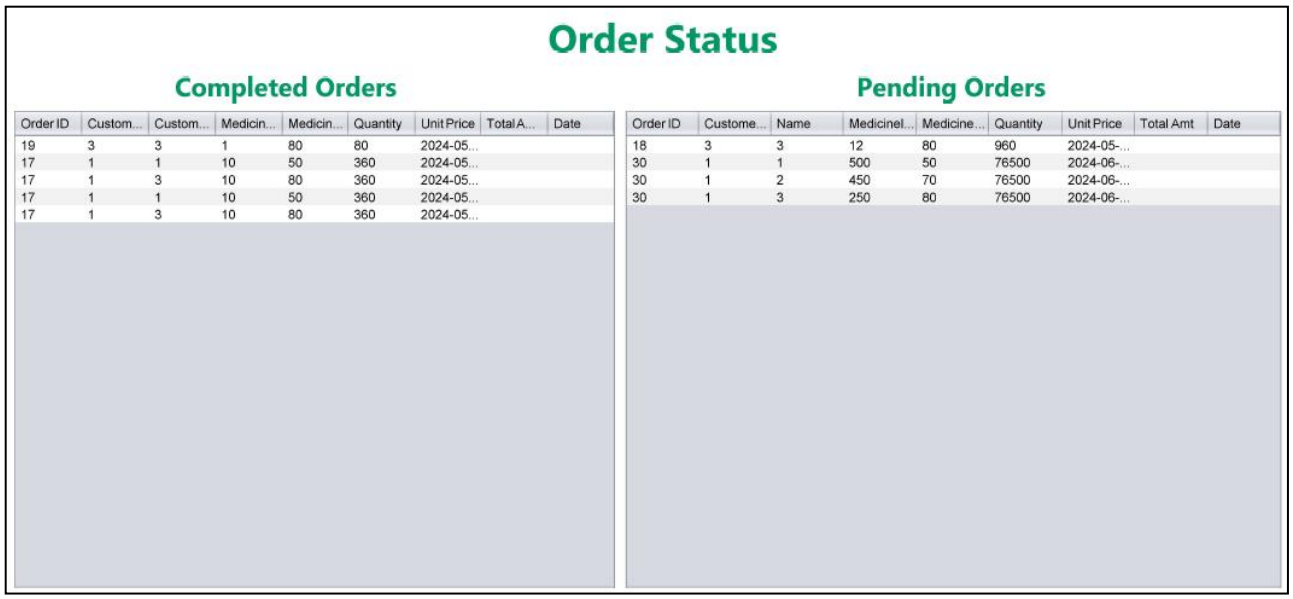

Figure 2.42: Order Status Report

❖ Billing Management – (IT22310996 – THENNAKOON T.A.C.S)

- **Add New Bill**

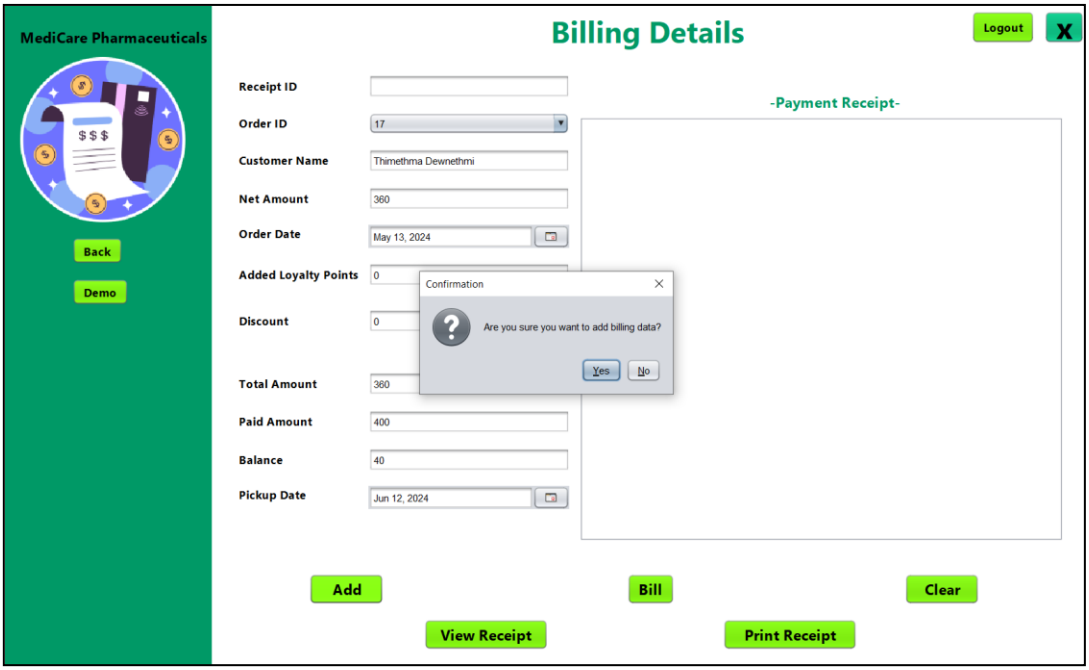

Figure 3.43: Add New Bill

- **View the Receipt**

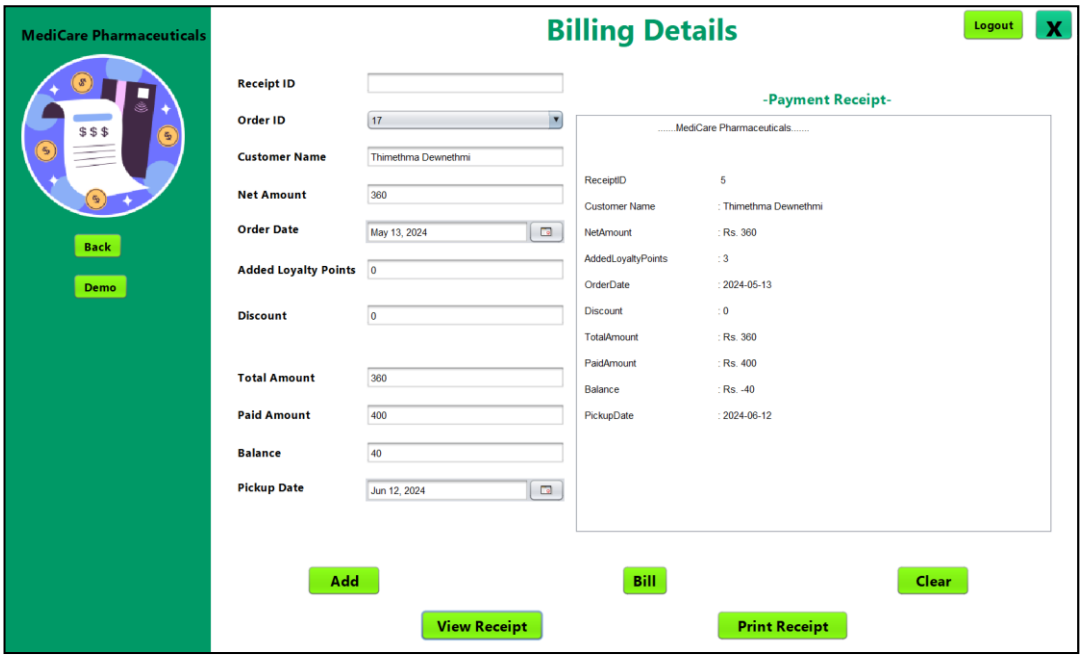

Figure 3.44: View the Bill

- **Print the Invoice**

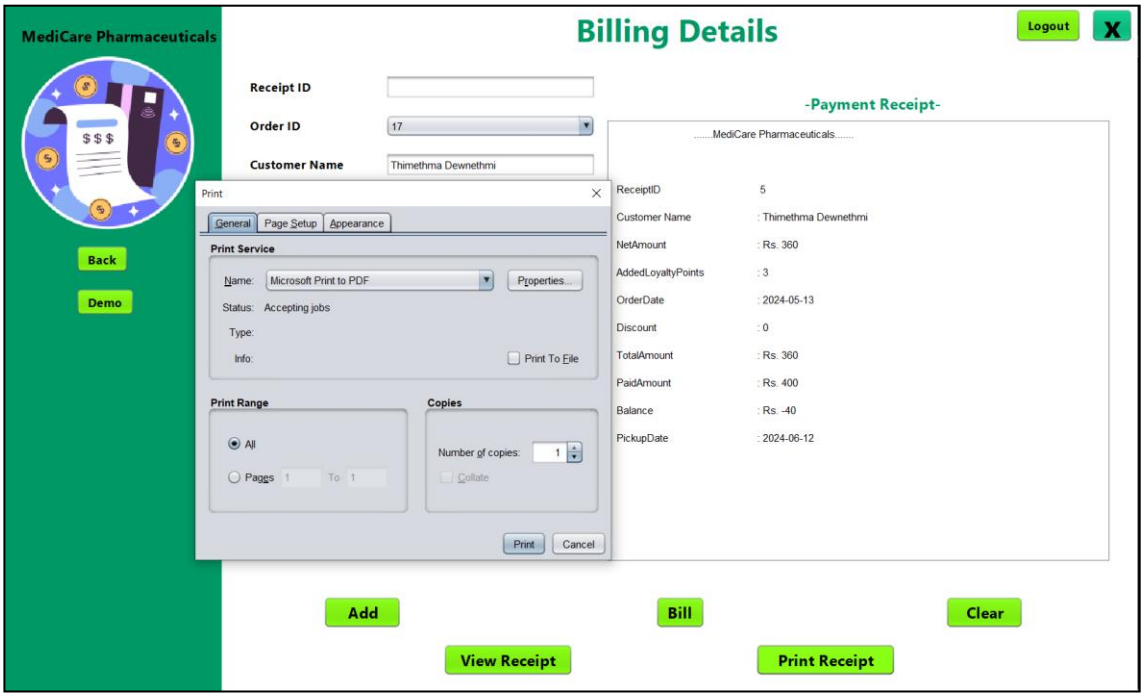

Figure 3.45: Print Invoice

- **Select an Invoice**

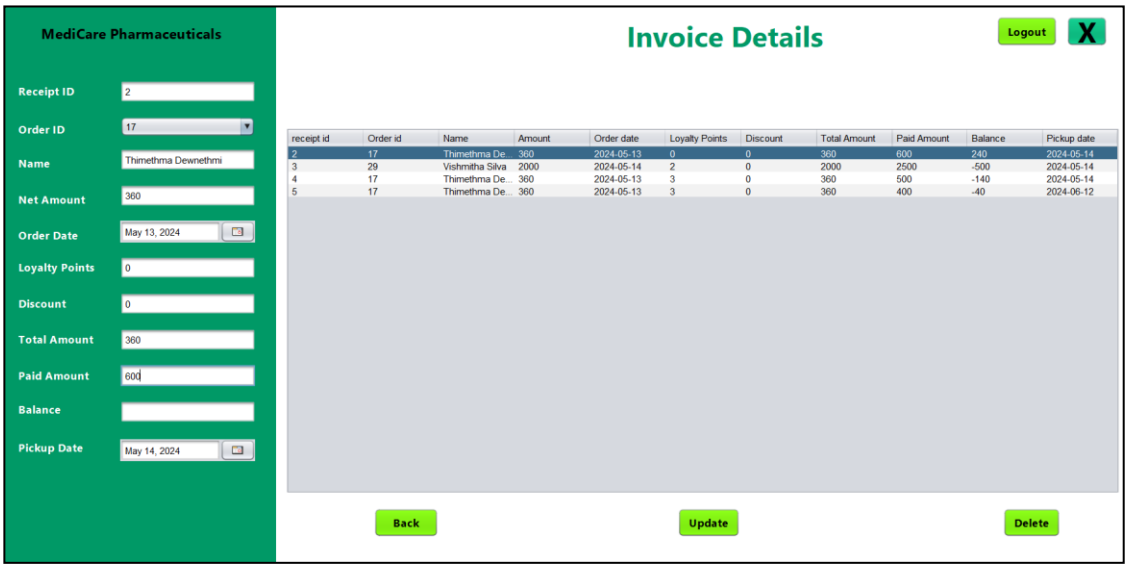

Figure 3.46: Selecting one of the invoices

- **Update the details of an Invoice**

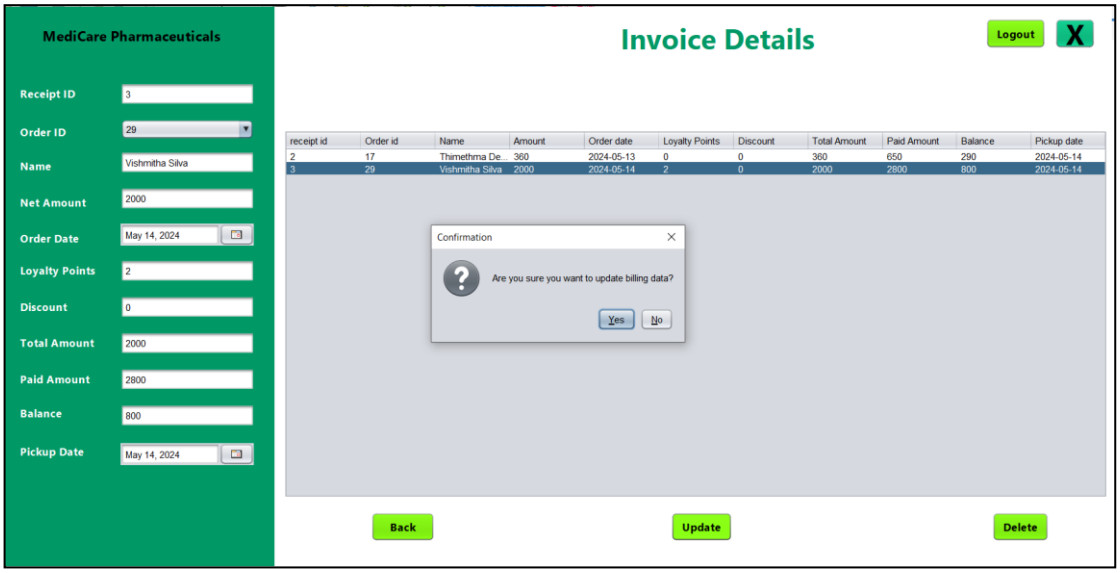

Figure 3.47: Update Invoice

- **Delete an Invoice**

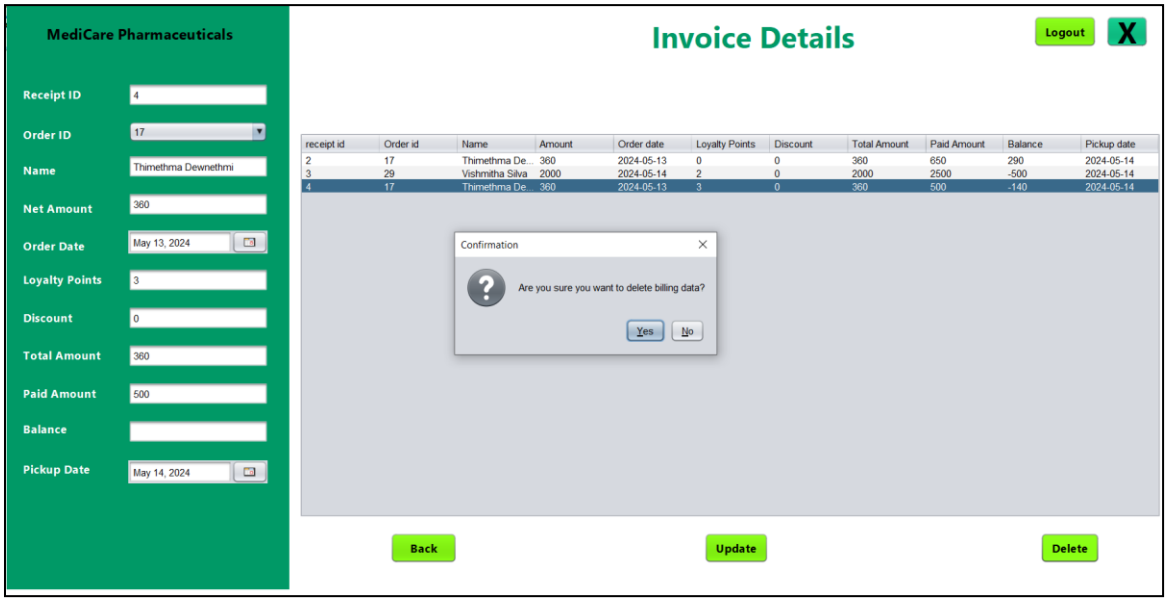

Figure 3.48: Delete an Invoice

❖ Sales Management – (IT22310996 – THENNAKOON T.A.C.S)

=

Figure 3.49: Sales Page UI

- **Generate Sales Report**

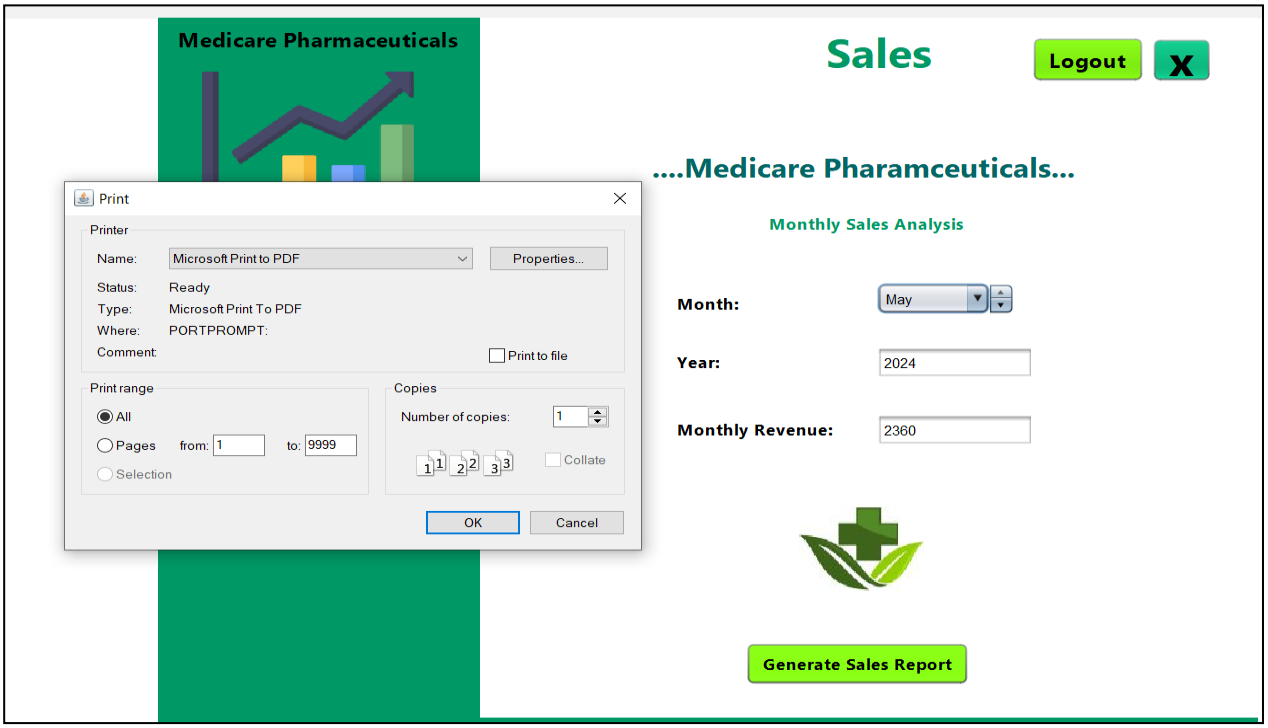

Figure 3.50: Sales Report Generation

- **Sales Report**

Figure 3.51: Sales Report

3.2 Lessons Learned

Several important lessons were learned during the creation of Medicare Pharmaceuticals' Desktop Pharmaceutical Management Application, which will improve our approach to future projects. First and foremost, it became clear how crucial thorough requirements elicitation is. It was crucial to make sure that the needs of the stakeholders were thoroughly understood at the beginning of the project to prevent misunderstandings and limit scope creeps later.

Second, it became clear that good communication was essential to the success of the project. Establishing consistent and lucid channels of communication among involved parties promoted agreement on project goals and nurtured a cooperative atmosphere. This made it easier to make quick changes in response to input from stakeholders, guaranteeing that the finished product satisfied their changing requirements.

Finally, the project demonstrated the value of documentation, risk management, user-centric design, agility, and continuous improvement. Agile development processes made it possible to adjust to changing needs, while thorough testing and user-centric design principles made sure the application was safe, dependable, and easy to use. Complete documentation, open communication of expertise, and proactive risk management reduced disruptions while promoting team cohesiveness and future improvements. Our culture of growth and innovation was promoted by constant reflection and improvement throughout the project lifecycle, setting us up for future success

3.3 Future Work

Future development on the Medicare Pharmaceuticals Desktop Pharmaceutical Management Application should give priority to several important topics. In the beginning, the user experience should be the main priority. This may be accomplished by adding user-friendly interfaces, customized dashboards, and enhanced navigation. To ensure that the application stays user-centric and effective, frequent user input sessions and thorough usability testing will be crucial in identifying and resolving areas for refinement.

Additionally, for the purpose of expediting data transmission and enhancing overall efficiency, it will be important to integrate the pharmaceutical management application with external systems, such as accounting software or electronic health records (EHR).[6] Facilitating smooth interaction with third-party systems through the development of strong APIs or middleware can streamline operations and improve data integrity on all platforms.

Furthermore, to give stakeholders a better understanding of pharmaceutical operations, future versions of the program ought to place a higher priority on advanced reporting and analytics features. To assist with well-informed decision-making and strategic planning, this may entail putting in place data visualization tools, predictive analytics, and configurable reporting templates. The Desktop Pharmaceutical Management Application may continue to develop and fulfill Medicare Pharmaceuticals' changing needs while being flexible to shifts in the pharmaceutical market environment by addressing these areas of improvement.

4. Conclusion

Upon completion of Medicare Pharmaceuticals' Desktop Pharmaceutical Management Application, the project has clearly accomplished its initial aims and objectives. The application streamlines and automates critical business activities, including inventory management, customer assistance, and decision-making, to successfully solve the shortcomings and inefficiencies of the company's manual techniques. The application provides a complete solution that improves productivity, accuracy, and efficiency in pharmaceutical company processes by utilizing the Java programming language.

To make the program even better, there are still flaws and restrictions, just like in any project. The lack of real-time data synchronization between the desktop application and other systems like supplier or regulatory databases is one of these weaknesses. By putting real-time data integration solutions into place or creating APIs for smooth data flow, this restriction can be removed. Furthermore, even though the program has extensive reporting and analytics functions, predictive analytics and machine learning algorithms should be improved to deliver more insightful and capable predictions.

To further enhance pharmaceutical operations, future development on the project may concentrate on integrating cutting-edge technology like artificial intelligence and machine learning. This can entail creating forecasting models for inventories, automating decision-making procedures, and putting in place recommendation engines with intelligence to provide individualized customer care. Long-term success will also depend on improving the application's scalability and adaptability to meet the changing needs of Medicare Pharmaceuticals and the pharmaceutical sector.

Medicare Pharmaceuticals can gain a lot from developing the Desktop Pharmaceutical Management Application. First off, by automating repetitive chores and optimizing corporate procedures, it greatly boosts operational efficiency and reduces costs while boosting production. Second, by lowering human error in data entry and processing, the application improves accuracy and data integrity. Thirdly, the program enables the client company to make informed decisions based on real-time insights by offering complete reporting and analytics capabilities. All things considered, the project gives Medicare Pharmaceuticals real value by updating their processes and setting them up for success in the rapidly shifting pharmaceutical market.

5. References

[1] Kirill Fakhroutdinov, "UML 2.4 Diagrams Overview.", www.uml-diagrams.org, https://www.uml-diagrams.org/uml-24-diagrams.html

[2] Kirill Fakhroutdinov," Use case diagrams are UML diagrams describing units of useful functionality (use cases) performed by a system in collaboration with external users (actors).", www.uml-diagrams.org, https://www.uml-diagrams.org/use-case-diagrams.html

[3] Pandey, S., & Pandey, S. (2024, June 6). How to Write a Project Report (with Best Practices Templates for Microsoft 365). BrightWork.com. https://www.brightwork.com/blog/7-steps-effective-report-writing

[4] Anusha Mysore Keerthi, Soujanya Ramapriya, Sriraksha Bharadwaj Kashyap, Praveen Kumar Gupta & B. S. Rekha," Pharmaceutical Management Information Systems: A Sustainable Computing Paradigm in the Pharmaceutical Industry and Public Health Management," www.simplerqms.com. ,20 Sept 2020. [online] Available: https://link.springer.com/chapter/10.1007/978-3-030-51070-1_2

[5] Staff, C. (2024, March 29). How to Write Test Cases: A Step-by-Step QA Guide. Coursera. https://www.coursera.org/articles/how-to-write-test-cases

[6] What is an electronic health record (EHR)? | HealthIT.gov. (n.d.). https://www.healthit.gov/faq/what-electronic-health-record-ehr

[7] Simple Guide to Desktop Application Testing. (n.d.). TestDevLab Blog.

https://www.testdevlab.com/blog/simple-guide-to-desktop-application-testing

[8] Chi, C. (2023, September 6). A Beginner's Guide to Data Flow Diagrams.

https://blog.hubspot.com/marketing/data-flow-diagram

Appendix A: Design Diagrams

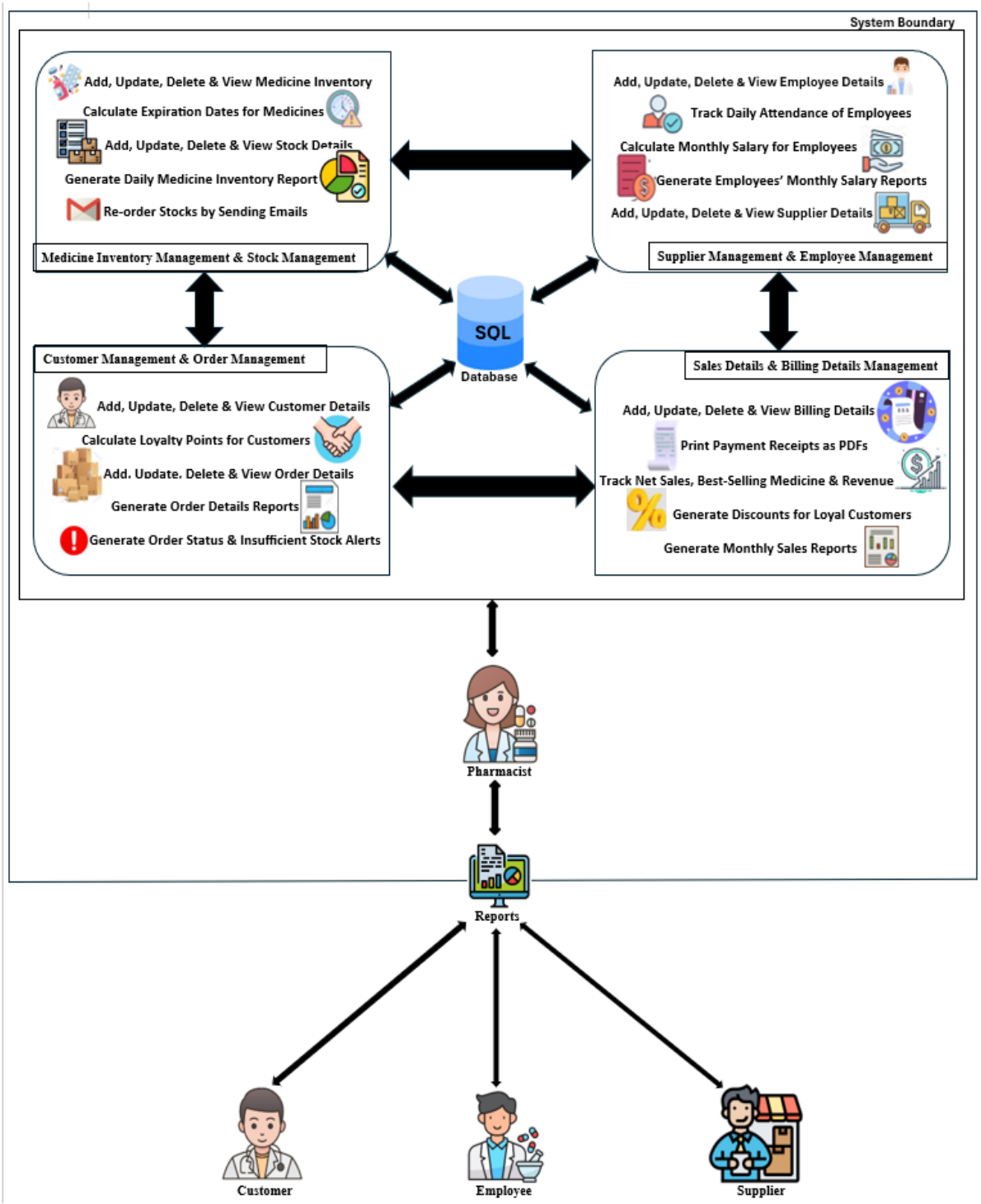

Figure 6.1: High – Level Architecture Diagram

Appendix B: Test Results

Assumptions:

- Only the pharmacist has access to the pharmaceutical management system and all the business functions will be managed by the pharmacist.
- The company will sell the medicines only.
- Main customers of this system will be doctors who run medical centers and local pharmacies.
- Only registered customers are allowed to place orders through the system.
- To supply medicines, a new supplier must be registered in the system first.
- A specific supplier supplies each medicine.
- Loyalty customers will be given 5% of the total amount of the order as a discount. (Initially, every customer will be given one loyalty point when they get registered to the system).
- Customers will be able to return their medicines within the first two weeks after the order is dispatched.
- Salary calculation of an employee will be done based on the daily rate and the monthly attendance.

Appendix C: Selected Code Listings

❖ Medicine Inventory Management Page - Codes of Special Algorithms

(IT22364692 – KANDAGE K.T.S)

- 'UnitPriceKeyPressed' and 'qtyKeyPressed' input validation examples make sure that only numeric values are entered.

```
private void unitPriceKeyPressed(java.awt.event.KeyEvent evt) {
    //Ensures that the user enters only the numerical values for price
    char c = evt.getKeyChar();

    //Validating the user inputs for the unit price
    if(Character.isLetter(ch: c))
    {
        qty.setEditable(b: false);
        JOptionPane.showMessageDialog(parentComponent: null, message: "Enter Numerical Values for the Unit Price",title: "Error",messageType: JOptionPane.ERROR_MESSAGE);
        qty.setEditable(b: true);
    }
}
```

Figure 8.1: Input Validations in Inventory

- **Interaction of Prepared Statements and Databases:** To avoid SQL injection in 'updateBtnActionPerformed', use 'PreparedStatement'.

```
// Use PreparedStatement to avoid SQL injection
java.sql.PreparedStatement prepSt = st.getConnection().prepareStatement(string:query);

prepSt.setString(i: 1, string:med_Name);
prepSt.setString(i: 2, string:medicineType);
prepSt.setString(i: 3, string:dosage);
prepSt.setInt(i: 4, il: unit_Price);
prepSt.setString(i: 5, string:storage_Location);
prepSt.setString(i: 6, string:date_Added);
prepSt.setInt(i: 7, il: quantity);
prepSt.setInt(i: 8, il: x);

int count = prepSt.executeUpdate();
```

Figure 8.2: Interaction of Prepared Statements and Databases

❖ Medicine Stock Management Page - Codes of Special Algorithms

(IT22364692 – KANDAGE K.T.S)

- Generate the Stock IDs : By asking the maximum stock ID currently in use and increasing it by one, this method creates the subsequent stock ID.

```
ResultSet rs = st.executeQuery(string:"SELECT MAX(stock_ID) FROM stock");

int nextStockID = 1;


if (rs.next()) {
    nextStockID = rs.getInt(i: 1) + 1;
}
```

Figure 8.3: Generate the Stock IDs

- Functionality of Search: Case-insensitive search is made possible by this method, which filters the table rows according to the user's inputted search string.

```
private void stockSearchBarKeyPressed(java.awt.event.KeyEvent evt) {
    // TODO add your handling code here:
    // Get the text from the search bar
String searchText = stockSearchBar.getText().trim();

// Get the table model of your medicine table
DefaultTableModel model = (DefaultTableModel) stock_table.getModel();

// Create a row sorter for the table model
TableRowSorter<DefaultTableModel> rowSorter = new TableRowSorter<>(model);

// Set the row sorter to the table
stock_table.setRowSorter(sorter:rowSorter);

// Apply the filter to show rows that contain the search text
if (searchText.length() == 0) {
    // If the search text is empty, show all rows
    rowSorter.setRowFilter(filter:null);
} else {
    // Otherwise, show rows that contain the search text
    rowSorter.setRowFilter(filter:RowFilter.regexFilter("(?i)" + searchText)); // Case insensitive filter
}
}
```

Figure 8.4: Search Functionality of Stocks

- Update of Quantity: This algorithm determines the new amount of medicine based on the amount that is already there as well as the amount that is being added or removed, updating the total amount of medicines in the stock.

```
int availableQTY = 0;
 java.sql.PreparedStatement prepSt2 = st.getConnection().prepareStatement(string:selectQuery1);

ResultSet result = prepSt2.executeQuery();

if(result.next()){
    availableQTY = result.getInt(string:"qty");
}
int temp = (availableQTY-availableQTYinStock)+Quantity;

String query1 = "UPDATE medicine SET qty = "+temp+" WHERE med_ID = "+med_id+" ";

java.sql.PreparedStatement prepSt1 = st.getConnection().prepareStatement(string:query1);

int count1 = prepSt1.executeUpdate();
```

Figure 8.5: Quantity Update of Stocks

❖ Customer Management Page - Codes of Special Algorithms

(IT22884138 – RATHNAYAKA R.M.T.D)

- Search algorithm - Although the application does not directly provide a search mechanism, it does allow customers to be found using their ID. Based on the user's input, it dynamically filters the rows in the JTable using a regex filter. This feature can be thought of as a basic type of search.

```
            try
            {
                //Handle the parsing of the Pickup Date
                Date pickup_Date = sfd.parse(source: pickupDate);
                date.setDate(date:pickup_Date);
            }
            catch (ParseException e)
            {
                // Handle the parsing exception,by displaying an error message.
                e.printStackTrace();

            }
        }
        catch (Exception e)
        {
            System.out.println(x: e.getMessage());
        }
    }
```

Figure 8.6: Customer Search Algorithm

- Sorting algorithm - In this segment, a TableRowSorter object (obj1) is created and initialized with the DefaultTableModel (obj) representing the data displayed in the JTable (customerTable). This sorter is then set as the row sorter for the table using the setRowSorter method. Once this is done, clicking on any column header in the JTable will trigger sorting based on the data in that column. Under the hood, the TableRowSorter class likely utilizes efficient sorting algorithms such as quicksort or mergesort to perform the sorting operation. However, the specifics of the sorting algorithm used by TableRowSorter are abstracted away from the developer and handled internally by the Swing framework.

```
DefaultTableModel obj = (DefaultTableModel) customerTable.getModel();
TableRowSorter<DefaultTableModel> obj1 = new TableRowSorter<>(model: obj);
customerTable.setRowSorter(sorter: obj1);
```

Figure 8.7: Customer Data Sorting Algorithm

❖ Employee Management Page - Codes of Special Algorithms (IT22319142)

(IT22319142 – WIJESINGHE A.G.T)

- Date Validation: In order to prevent future dates from being chosen as birth dates, the code limits the date chooser for the date of birth (dob) to only accept past dates. Additionally, it ensures that no previous or future dates can be chosen by setting the start Date to the current date and restricting it to only accept the current date.

Eg:

```
// Restricting the dob JDateChooser to only allow past dates
dob.setDate(date: null); // Clear any default date set

// Set maximum selectable date to yesterday's date
Calendar cal = Calendar.getInstance();
// Subtracting 1 day from today's date
cal.add(field:Calendar.DAY_OF_MONTH, amount: -1);
Date yesterday = cal.getTime();

dob.setMaxSelectableDate(date: yesterday);
```

Figure 8.8: Employee Management Page Date Validation

- NIC Validation: The code verifies that the National Identity Card (NIC) number is precisely 12 characters long, as required by the NIC format.

Eg:

```
if (N_I_C.length() != 12)
{
    JOptionPane.showMessageDialog(parentComponent: null, message: "Please enter 12 numerical values for the NIC!", title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    return; // Stop processing,if NIC length is incorrect
}

// Check whether the NIC already exists in the database
try
{
    Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
    ResultSet existingNIC = st.executeQuery("SELECT * FROM employee WHERE NIC = '" + N_I_C + "'");

    if (existingNIC.next())
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Please check whether the NIC is correct! There is another registered employee with the same NIC you are trying to insert!",
                title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        return; // Stop processing further if the NIC already exists in employee table
    }
}
```

Figure 8.9: Employee NIC Validation

- Unique NIC Check: In order to maintain data integrity and avoid duplicate entries, the algorithm verifies whether the NIC already exists in the database before adding a new employee.

Eg:

```
// Check whether the NIC already exists in the database
try
{
    Statement st = pharmacy.DBconnection.createDBconnection().createStatement();
    ResultSet existingNIC = st.executeQuery("SELECT * FROM employee WHERE NIC = '" + N_I_C + "'");

    if (existingNIC.next())
    {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Please check whether the NIC is correct! There is another registered employee with the same NIC you are trying to insert!",
                title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
        return; // Stop processing further if the NIC already exists in employee table
    }
}
```

Figure 8.10: Employee Unique NIC Validation

Input Validation: Several input validations are included in the code, such as ensuring that the daily rate and contact number are entered numerically and that names are entered alphabetically. These checks protect against incorrect entries and preserve consistency in the data.

❖ Daily Attendance Tracking Page - Codes of Special Algorithms

(IT22319142 – WIJESINGHE A.G.T)

- Checking whether the Attendance is Already Marked

```
private boolean isAttendanceMarkedToday(int employeeID)
{
    boolean attendanceMarked = false;

    try
    {
        SimpleDateFormat sdf = new SimpleDateFormat(pattern: "yyyy-MM-dd");
        String todayDate = sdf.format(new Date());

        String query = "SELECT * FROM attendance WHERE empID = " + employeeID + " AND date = '" + todayDate + "'";
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery(string: query);

        if (rs.next())
        {
            attendanceMarked = true;
        }

        rs.close();
        st.close();
        DBconnection.createDBconnection().close();
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error checking attendance: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
    return attendanceMarked;
}
```

Figure 8.11: Duplicate Attendance Record Validation

- Refreshing the Attendance Table after Adding/Updating/Deleting a Record

```java
// Refresh the Attendance Table
private void refreshAttendanceTable()
{
    try
    {
        Statement st = DBconnection.createDBconnection().createStatement();
        ResultSet rs = st.executeQuery(string: " SELECT * FROM attendance ");

        DefaultTableModel model = new DefaultTableModel();
        model.addColumn(columnName:"Attendance ID");
        model.addColumn(columnName:"Date");
        model.addColumn(columnName:"Employee ID");
        model.addColumn(columnName:"Employee Name");
        model.addColumn(columnName:"Attendance");

        while (rs.next())
        {
            Object[] row =
            {
                rs.getInt(string: "attendanceID"),
                rs.getString(string: "date"),
                rs.getInt(string: "empID"),
                rs.getString(string: "empName"),
                rs.getString(string: "attendance")
            };
            model.addRow(rowData: row);
        }

        attendanceTable.setModel(dataModel: model);
    }
    catch (SQLException ex)
    {
        JOptionPane.showMessageDialog(parentComponent: null, "Error occurred while refreshing Table: " + ex.getMessage(), title:"Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 8.12: Refresh the Attendance Table

❖ Monthly Salary Calculation Page - Codes of Special Algorithms

(IT22319142 – WIJESINGHE A.G.T)

- The name and daily rate of the employee are retrieved from the database when an employee ID is selected using the code below:

```java
employID.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(ActionEvent e)
    {
        try
        {
            // Get the selected employee ID from the combo box
            int selectedEmpID = Integer.parseInt(s: employID.getSelectedItem().toString());

            // Retrieve the employee's name and daily rate based on the selected employee ID
            String employee_Name = getEmployeeNameByEmployeeID(employeeID:selectedEmpID);
            int employeeDailyRate = getEmployeeDailyRateByEmployeeID(employeeID:selectedEmpID);

            // Display the employee's name and daily rate in respective text fields or labels
            employeeName.setText(t: employee_Name);
            daily_rate.setText(t: String.valueOf(i: employeeDailyRate));
        }
        catch (NumberFormatException ex)
        {
            // Handle number format exception if selectedEmpID is invalid
            ex.printStackTrace();
        }
    }
});
```

Figure 8.13: Retrieve Daily Rate and the Employee Name

- The AddBtnActionPerformed method includes logic for validating input, calculating net salary, and inserting a new salary record into the database:

```
employID.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(ActionEvent e)
    {
        try
        {
            // Get the selected employee ID from the combo box
            int selectedEmpID = Integer.parseInt(s: employID.getSelectedItem().toString());

            // Retrieve the employee's name and daily rate based on the selected employee ID
            String employee_Name = getEmployeeNameByEmployeeID(employeeID:selectedEmpID);
            int employeeDailyRate = getEmployeeDailyRateByEmployeeID(employeeID:selectedEmpID);

            // Display the employee's name and daily rate in respective text fields or labels
            employeeName.setText(t: employee_Name);
            daily_rate.setText(t: String.valueOf(i: employeeDailyRate));
        }
        catch (NumberFormatException ex)
        {
            // Handle number format exception if selectedEmpID is invalid
            ex.printStackTrace();
        }
    }
});
```

Figure 8.14: Validate input in Salary Table