温州肯恩大学

WENZHOU-KEAN UNIVERSITY

CPS 3320

Project Report

Billing and Inventory System (BIS)

Team member:

Xue Zhang	**1163317**	**1163317@wku.edu.cn**
Zhirong Yu	**1194166**	**yuzhi@kean.edu**
Jiayin Li	**1195011**	**lijiayin@kean.edu**

2024 Spring Semester

Section W04

List of Contents

Abstract

Our Billing and Inventory System (BIS) is a Python-based application that simplifies billing and inventory management processes within enterprises. It utilizes PyQt5 for the user interface and SQLite as a lightweight and serverless database system. The system offers separate interfaces for administrators, auditors, and customers, providing functionalities like customer management, transaction tracking, billing generation, payment records management, and more. It emphasizes simplicity, interactivity, and modularization for ease of use and code maintenance. While improvements are suggested for error handling, additional user attributes, and addressing SQLite limitations, the BIS system demonstrates practicality in enhancing operational efficiency and accuracy for enterprises.

Chapter 1 : Project Introduction

A Billing and Inventory System (BIS) is an application designed to simplify managing billing and inventory processes within an enterprise. It provides features like monitoring sales, generating invoices, and tracking customer transaction activity. The practical application significance of BIS is that it can improve operational efficiency and accuracy, ultimately benefiting enterprises in many aspects. Based on the PyQt5 framework, we built a billing and inventory system (BIS) with administrators, auditors, and customers as the main application groups. According to the classification of user attributes, we use Qt Designer to build a graphical user interface (GUI) for the BIS application's main interface, administrators/auditors, and customers. Users with different identities can log in to their own interface after system verification to complete the tasks of the corresponding module. The specific system operation procedures and application functions of our BIS are as follows:

1.1 System Operation Process

1. The user starts the application and enters the main interface.
2. User login and verification
    - The user enters the username and password.
    - The system authenticates and loads the appropriate user interface.
3. After successful login, the user enters the personal home interface (different user attributes include different visible application modules)
    - The administrator interface contains all customer information and each customer's bills and transactions. The interface that auditors can see is the same as that of administrators, but they cannot modify the information.
    - The Customer interface contains its own transactions and bills.

1.2 Module Task Operations

1. The Administrator Interface
    - Customer Module: Complete tasks of adding, deleting, searching, and updating customers and information
    - Trading Module: View transaction history and modify or delete transaction records.

- Billing Module: Generate new bills, edit bills, search and view bill details.
- Payment Module: Add payment records and modify or view payment details.
- Transaction History Module: View the history of all transactions, including detailed descriptions and related data.
- Report Generation Module: Generate Excel reports of customers, transactions, and bills for review or record keeping.
- Exit System Module: The administrator exits the system after completing the operation.

2. The Customer Interface:

- Bill Viewing Module: View details of the bill.
- Transaction Viewing Module: View the transaction history and details.
- Payment Module: Select the payment type.
- Exit System Module: The customer exits the system after completing the operation.

We utilize SQLite with application logic as local data storage in BIS applications. Better management of customers, administrators, transactions, bills, and other data. Implemented as a lightweight, serverless relational database management system (RDBMS). Achieve authentication and role-based access control.

Chapter 2 : Related Work

PyQt5 (*Overview of PyQt5 | SpringerLink*, n.d.) follows a widget-based approach, where GUI components are represented as widgets. Widgets are visual elements such as buttons, labels, text inputs, and containers that can be arranged and interacted with. PyQt5 provides a wide range of pre-built widgets that can be customized and combined to create complex GUI layouts. It utilizes the signals and slots mechanism provided by Qt. Signals are events emitted by widgets when specific actions occur, such as a button click. Slots are Python methods that are connected to signals and get executed when the corresponding signal is emitted. This mechanism enables event-driven programming, allowing developers to respond to user actions and update the GUI dynamically. For layout Management, PyQt5 (Peiming et al., 2020) offers flexible layout management options to arrange widgets within a window or container. Layout managers, such as QVBoxLayout, QHBoxLayout, and QGridLayout, help automatically handle widget positioning, resizing, and alignment. This simplifies the process of creating responsive and resizable interfaces that adapt to different screen sizes and orientations. PyQt5 provides extensive options for customizing the visual appearance of widgets. Using Qt's stylesheet mechanism, developers can apply CSS-like styles to widgets, change colors, fonts, and sizes, and define custom widget styles. This allows for the creation of visually appealing and branded GUIs that match specific design requirements.

SQLite is a popular open-source, embedded relational database management system that is widely used in various applications and platforms. Over the years, several studies and research works have focused on different aspects of SQLite, ranging from performance optimization to security and scalability. Numerous studies have focused on specific application domains that extensively use SQLite. Examples include research on SQLite in mobile applications, embedded systems, IoT devices, web browsers, and data-driven applications. These studies examine SQLite's suitability, performance, and optimization techniques tailored to the specific application requirements (Obradovic et al., 2019). A lot of research has been done to improve the performance of SQLite. This includes a survey of query optimization techniques, indexing strategies, caching mechanisms, and memory management. Researchers explored ways to enhance the execution speed and efficiency of SQLite, making it suitable for resource-constrained environments (Bi, 2009).

Chapter 3: Reports and Analysis

This project describes a simple and effective Python-based Billing and Inventory System (BIS). It uses the PyQt5 framework to implement the user interaction interface, and SQLite provides data management under different user attributes.

3.1 Good Characteristics

- Simplicity: Each function is clearly defined and uncomplicated, making the code simple to comprehend and modify.
- Interactivity: Integrating QtCore, QtGui, and QtWidgets in PyQt5 provides a clear and concise interactive interface for different user groups. This allows users to perform simple and easy-to-understand operations through visual icons, dialog boxes, menus, and other elements.

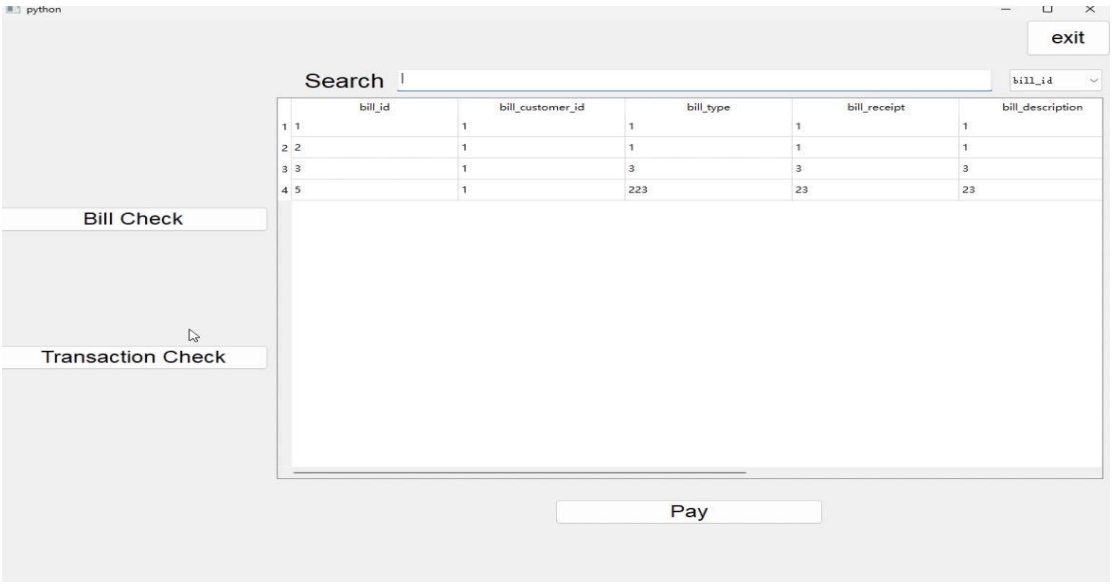


Figure 3.1. The Customer Interface

- Modularization: Modularize the functions of the Billing and Inventory System, using user attributes and corresponding overall functions as classification standards. Such integrated task classification facilitates code reading, modification, and maintenance.

- Database management: SQLite provides a lightweight, portable, and easy-to-use database solution for our project. It simplifies administration and eliminates the need for complex database server setup.

3.2 Weaknesses

- Error Handling: The system lacks robust error handling at the moment. It does not account for situations in which the user may enter incorrect information, which could result in unintended behavior.
- Single-User Mode: By default, SQLite operates in a single-user mode, where only one process can write to the database simultaneously. This can be a limitation in applications that require high-concurrency write operations.

3.3 Improvement

- Error Handling: This program can increase the number of confirmations of user login information and corresponding prompts to avoid user login errors and duplication.
- Function Improvements: Add related functions related to inventory management. Monitor inventory levels based on sales and purchases and generate reports on product movement.

Chapter 4 : Methodology

4.1 System Design and Planning

The project begins with designing the architecture of the Billing and Inventory System. The core components include user interfaces for administrators and customers, a database for storing data, and modules for handling different functionalities such as billing, inventory, and transaction management.

4.2 Development Environment Setup

The development environment includes setting up Python with necessary libraries, primarily PyQt5 for the GUI and SQLite for the database. The structure of the project is organized into directories and files for better management.

4.3 Graphical User Interface (GUI) Development

Tools and Frameworks:

- PyQt5: Used for creating the graphical user interface.
- Qt Designer: Employed to design the GUI visually, which is then converted into Python code.

The GUI components are divided into multiple windows and dialogs:

- loginWindow.py: Manages user authentication.
- adminWindow.py: Interface for administrators to manage customers, transactions, billing, and generate reports.
- customerWindow.py: Interface for customers to view their transactions and billing details.
- auditWindow.py: Interface for auditors (future enhancement).

4.4 Database Management

Tools and Technologies:

- SQLite: A lightweight, serverless RDBMS for storing and managing data.

The database schema includes tables for users, transactions, billing, and inventory. The database files (example.db, management.db) store the actual data, while

databases.py and dbmanager.py handle database operations.

4.5 Core Functionality Implementation

The core functionality is implemented in various Python scripts:

- main.py: The main entry point of the application.
- config.py: Contains configuration settings for the application.
- registerWindow.py: Handles user registration process.

Modules and Their Functions:

- Customer Management: Adding, updating, deleting, and searching for customer records.
- Transaction Management: Viewing and modifying transaction history.
- Billing Management: Generating, editing, and viewing bill details.
- Payment Records: Adding and viewing payment records.
- Report Generation: Exporting data to Excel for review or record-keeping.

4.6 Error Handling and Validation

The system includes basic error handling and validation to ensure data integrity and user input correctness. Future improvements are suggested for enhancing error handling mechanisms.

4.7 Testing and Debugging

The application is tested thoroughly to ensure all modules work as expected. Bugs and issues identified during testing are fixed to improve stability and performance.

4.8 Documentation and User Guide

Documentation is created to help users understand how to use the system. This includes:

- User Manual: Instructions on how to navigate and use different features of the system.
- Code Documentation: Comments and explanations within the code for

developers.

4.9 Deployment and Maintenance

The application is packaged and deployed for use within the enterprise. Regular maintenance is performed to address any issues and update features as required.

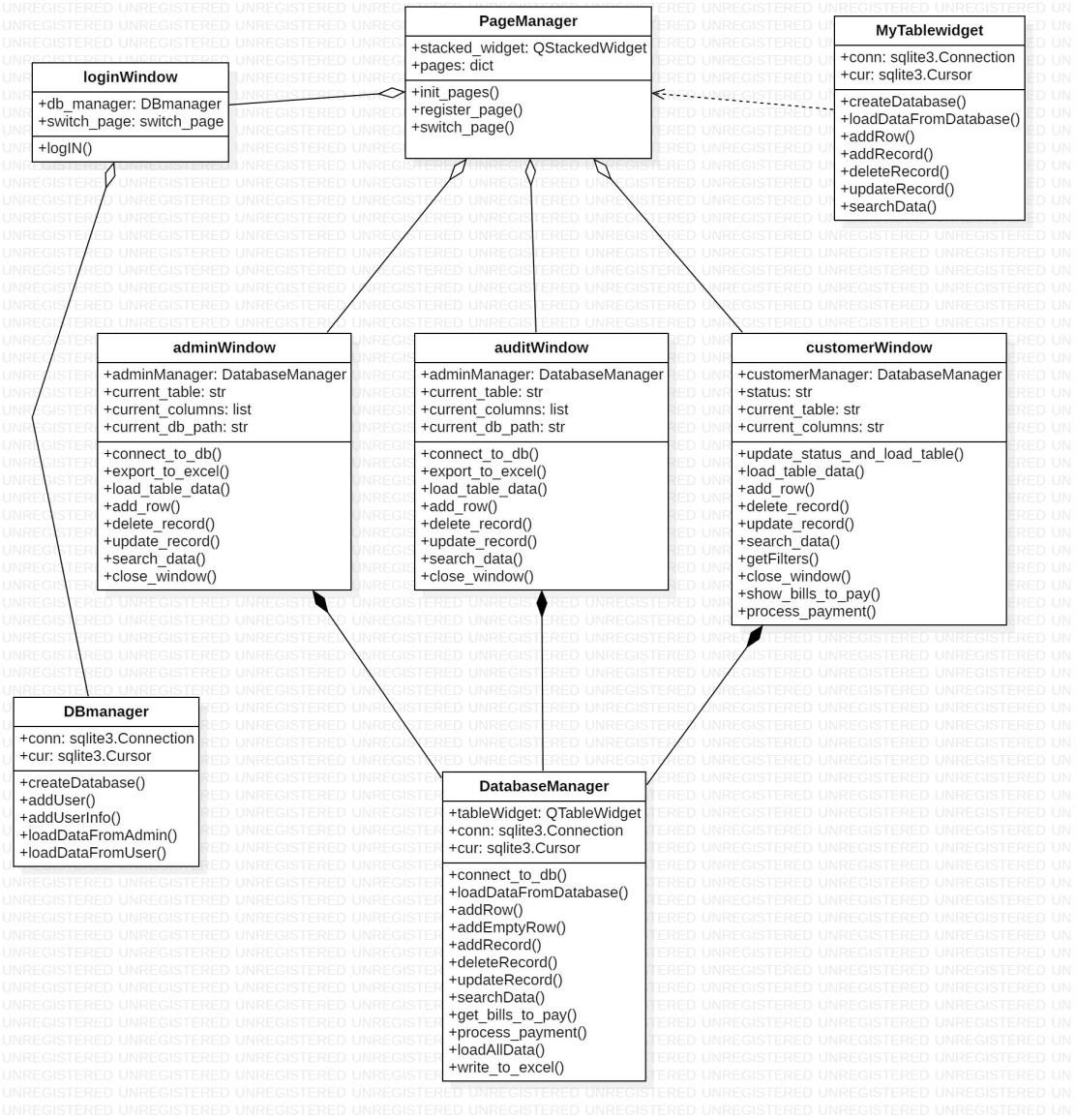
4.10 Future Enhancements

Suggested improvements for future versions include:

- Robust error handling mechanisms.
- Additional user roles (e.g., auditor).
- Enhanced inventory management features.
- Multi-user support for concurrent database access.

Chapter 5 : UML Diagrams

5.1 Class Diagram



- **PageManager**:

  This class is responsible for managing the different pages within our application using a QStackedWidget. It holds a dictionary of pages and includes methods for initializing, registering, and switching between pages. The key methods here are init_pages(), register_page(), and switch_page().

- **MyTableWidget:**

This class manages the interaction with the SQLite database. It includes methods for creating the database, loading data, adding, deleting, and updating records. The critical methods include createDatabase(), loadDataFromDatabase(), addRecord(), deleteRecord(), and updateRecord().

- **adminWindow:**

  Managed by adminManager, this class allows administrators to interact with the database. It includes methods for connecting to the database, loading and updating data, adding and deleting records, and exporting data to Excel. Key methods include connect_to_db(), load_table_data(), add_row(), delete_record(), and export_to_excel().

- **auditWindow:**

  Managed by auditManager, this class is similar to adminWindow but focuses on audit-related tasks. It also includes methods for database interactions, data loading, updating, and exporting. The important methods are connect_to_db(), update_status_and_load_table(), search_data(), and close_window().

- **customerWindow:**

  Managed by customerManager, this class is tailored for customer interactions. It includes methods for updating statuses, loading and searching data, and processing payments. Key methods include update_status_and_load_table(), search_data(), process_payment(), and close_window().

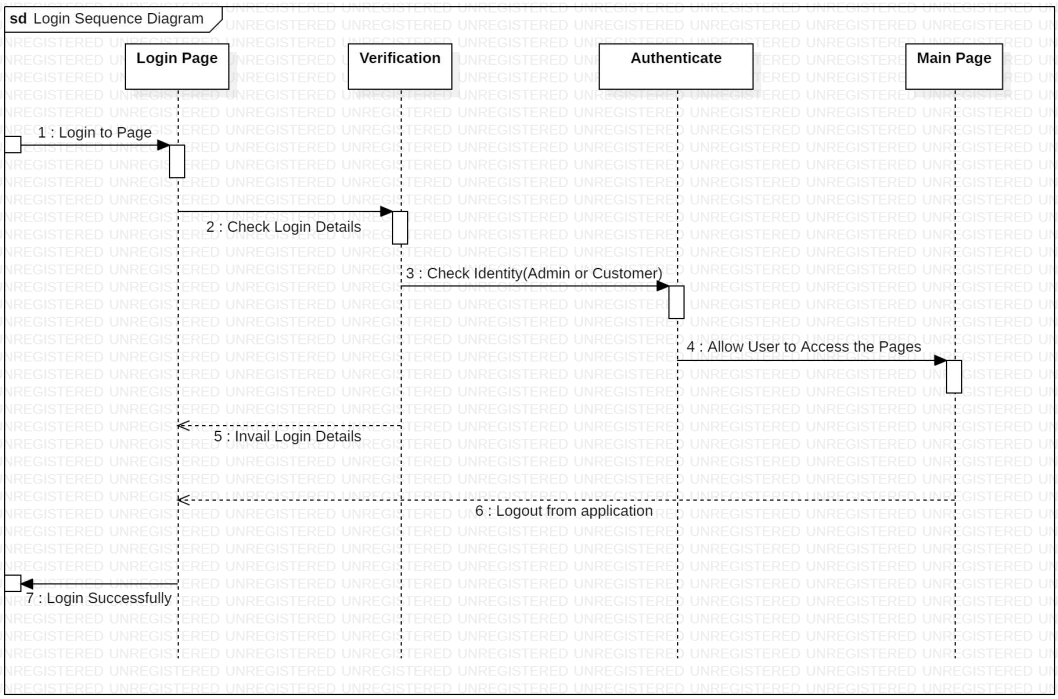
- **DBManager:**

  This class provides methods for creating the database, adding users, loading data from different sources, and performing CRUD operations. The critical methods are createDatabase(), addUserInfo(), loadDataFromAdmin(), loadDataFromUser(), addRecord(), deleteRecord(), and updateRecord().

- **DatabaseManager:**

As a general manager for the database operations. It includes methods for connecting to the database, loading data, adding, deleting, and updating records, as well as exporting data to Excel. Key methods are connect_to_db(), loadDataFromDatabase(), addEmptyRow(), deleteRecord(), updateRecord(), get_bills_to_pay(), show_bills_to_pay(), and write_to_excel().

5.2 Login Sequence Diagram



- **Login Page:**

  The process begins at the Login Page, where the user inputs their credentials.

  Upon submission, the credentials are sent for verification.

- **Verification:**

  The Verification component receives the credentials and checks them against the stored data.

  If the credentials are valid, the process moves to the next step, which is authentication.

- **Authenticate:**

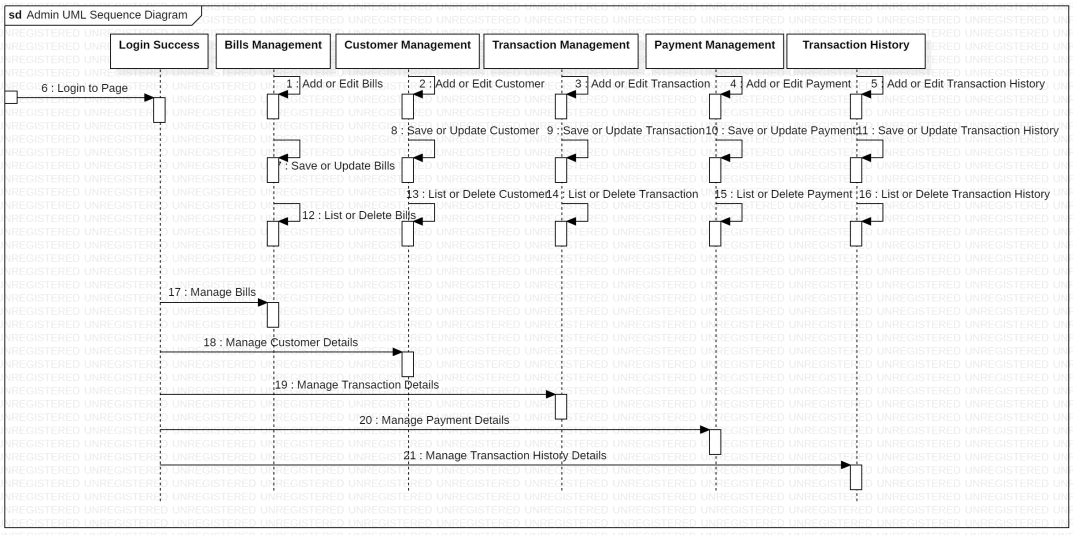
The Authenticate component handles the validation of the user's session.

If authentication is successful, the user is granted access and redirected to the main page.

- **Main Page:**

  Finally, the user is directed to the Main Page, where they can access the application's functionalities.

5.3 Admin UML Sequence Diagram



- **Login Success:**

  After a successful login, the admin is authenticated and gains access to the admin functionalities.

- **Bills Management:**

  The admin can manage bills, including adding, updating, and viewing bill details.

  This is crucial for maintaining accurate financial records.

- **Customer Management:**

The admin can manage customer information, including adding new customers, updating customer details, and viewing customer records.

This ensures that customer data is kept up-to-date and organized.

- **Transaction Management:**

  The admin can oversee transactions, ensuring that all financial activities are properly recorded and managed.

  This includes viewing transaction histories and ensuring their accuracy.

- **Payment Management:**

  The admin can manage payments, including processing and verifying payments.

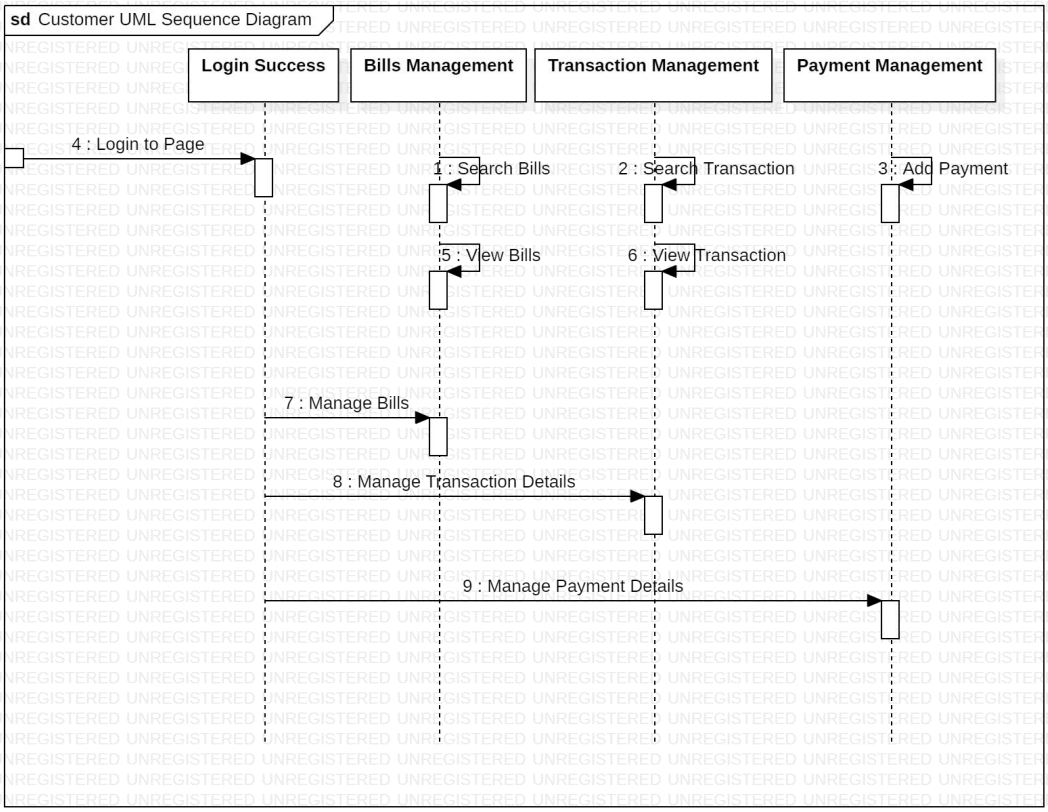
  This is essential for ensuring that all payments are handled correctly and efficiently.

- **Transaction History:**

  The admin can access transaction history, providing a detailed overview of all past transactions.

  This helps in auditing and tracking financial activities over time.

5.4 Customer UML Sequence Diagram



- **Login Success:**

  Similar to the admin process, the customer starts with a login.

  Upon successful authentication, the customer is granted access to the system and directed to their dashboard.

- **Bills Management:**

  The customer can view and manage their bills.

  This includes viewing detailed bill information and ensuring that all their bills are up-to-date.

- **Transaction Management:**

  Customers can manage their transactions, which includes viewing transaction details and histories.

  This feature helps customers keep track of their financial activities within the

application.

- **Payment Management:**

  This functionality allows customers to manage their payments.

  They can make payments, view payment histories, and ensure that all payments are processed correctly.

Chapter 6 : Conclusion

The Python-based Billing and Inventory System (BIS) is a complex project that aims to provide customers and managers with valuable insights into sales performance and inventory management in a streamlined operational process. The system uses the PyQt5 framework to build a user interaction interface to visualize and simplify operations. This project uses SQLite to handle authentication and role-based access control, making it more convenient to operate different user attributes. Effectively improves data management and enables scenario-based practical applications. The core function of the system is to use managers and customers as classification standards to detect and manage elements (such as bills and payment records) and information generated during the transaction process between the two.

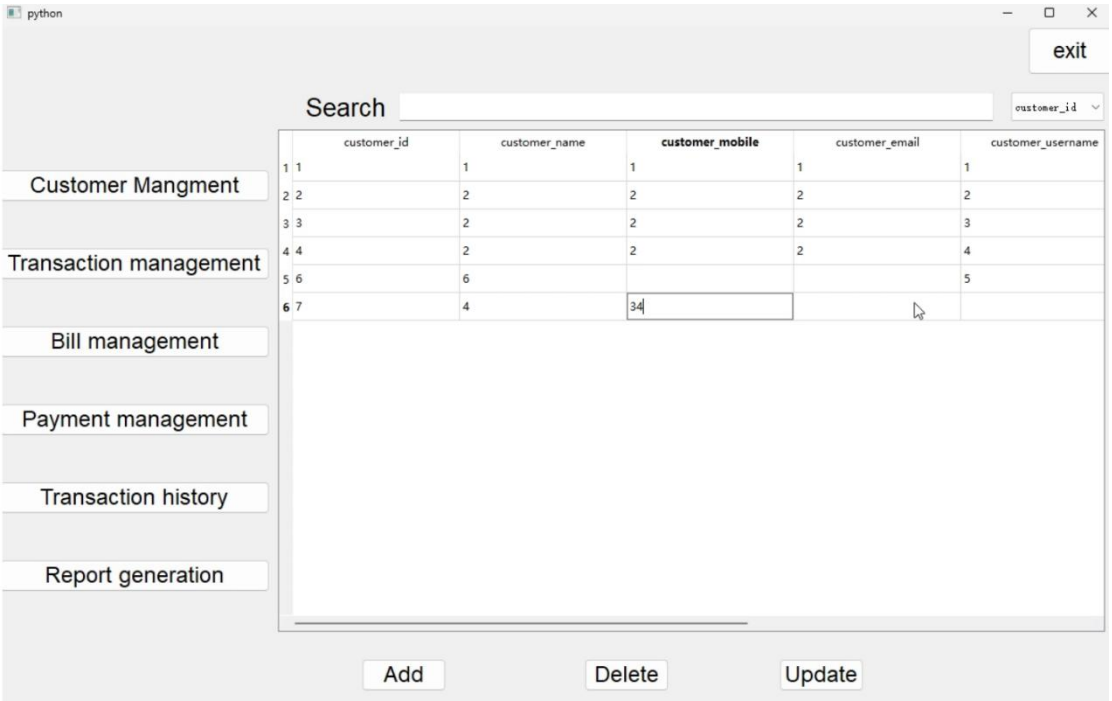


Figure 5.2. The Administrator Interface

The customer interaction aspects of the system are designed with user-friendliness in mind. Users can automatically jump to the interface that matches their user attributes by entering their username and password. Administrators, auditors and customers can easily complete different tasks through various visual buttons. For example, managers can enter this module by clicking the "Customer Management" button. He can click "Add," "Delete," and "Update" to complete the addition, deletion, and update of user information. At the same time, the eye-catching "Search" key above also makes it easier for managers to search for customer information. Additionally, managers can add and modify parameters of different customer attributes (eg. customer_id,

customer_name) directly on the interface. If the user wants to exit the system, he only needs to click the "exit" key to complete the operation efficiently. This streamlined process is intuitive and efficient, enhancing the overall user experience.

Overall, the Python-based Billing and Inventory System (BIS) presented in the document demonstrates the practical application of managing billing and inventory processes. With its user-friendly interface, efficient task completion, and effective data management, the BIS system can improve operational efficiency and accuracy within enterprises. While the system offers valuable features, there are areas for improvement. The document highlights the need for robust error handling, additional user attributes beyond managers and customers, and addressing limitations related to SQLite's single-user mode. To enhance the system, suggestions include implementing better error handling, adding inventory management functions, and generating reports on product movement based on sales and purchases.

References

Bi, C. (2009). *Research and Application of SQLite Embedded Database Technology*. *8*(1).

Obradovic, N., Kelec, A., & Dujlovic, I. (2019). Performance analysis on Android SQLite database. *2019 18th International Symposium INFOTEH-JAHORINA (INFOTEH)*, 1–4. https://doi.org/10.1109/INFOTEH.2019.8717652

Overview of PyQt5 | SpringerLink. (n.d.). Retrieved May 20, 2024, from https://link.springer.com/chapter/10.1007/978-1-4842-6603-8_1

Peiming, G., Shiwei, L., Liyin, S., Xiyu, H., Zhiyuan, Z., Mingzhe, C., & Zhenzhen, L. (2020). A PyQt5-based GUI For Operational Verification Of Wave Forcasting System. *2020 International Conference on Information Science, Parallel and Distributed Systems (ISPDS)*, 204–211. https://doi.org/10.1109/ISPDS51347.2020.00049

Chapter 7 : Appendix (Code)

7.1 adminWindow.py

```
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtCore import pyqtSlot
from PyQt5.QtGui import QCloseEvent
from PyQt5.QtWidgets import QWidget, QTableWidgetItem, QTableWidget,
QVBoxLayout, QPushButton, QLabel, QTextEdit, \
    QMessageBox, QApplication, QFileDialog
from dbmanager import DatabaseManager,DBmanager
import config
class adminWindow(QWidget):
    def __init__(self, switch_page, parent=None):
        super(adminWindow, self).__init__(parent)
        self.ui = Ui_adminWindow()
        self.ui.setupUi(self)
        self.adminManager = DatabaseManager(self.ui.tableWidget)
        self.current_table = None
        self.current_columns = None
        self.current_db_path = None
        self.current_db_path = 'management.db'
        self.connect_to_db()
        self.bind_main_buttons()
        self.bind_crud_buttons()

        self.ui.textEdit.textChanged.connect(self.search_data)

    def bind_main_buttons(self):

        self.ui.pushButton_3.clicked.connect(lambda:
self.load_table_data('management.db', 'user_info', ['customer_id',
'customer_name',
'customer_mobile','customer_email','customer_username','customer_pass
word','customer_address']))
        self.ui.pushButton_5.clicked.connect(lambda:
self.load_table_data('management.db', 'transaction_info', [
            'transaction_id','transaction_customer_id',
'transaction_amount', 'transaction_bill',
            'transaction_number', 'transaction_type',
'transaction_history', 'transaction_description'
        ]))
        self.ui.pushButton.clicked.connect(lambda:
self.load_table_data('management.db', 'bill_info', ['bill_id',
'bill_customer_id',
'bill_number','bill_type','bill_receipt','bill_description','bill_amo
unt','unpaid_bill']))
        self.ui.pushButton_2.clicked.connect(lambda:
self.load_table_data('management.db', 'payment_info', ['payment_id',
'payment_customer_id', 'payment_des','payment_status']))
        self.ui.pushButton_6.clicked.connect(lambda:
self.load_table_data('management.db', 'transaction_history_info',
['transaction_history_id', 'transaction_history_customer_id',
'transaction_history_amount', 'transaction_history_bill',
'transaction_history_number', 'transaction_history_type',
'transaction_history_description']))
        self.ui.pushButton_7.clicked.connect(self.export_to_excel)
        self.ui.pushButton_4.clicked.connect(self.close_window)
```

```
    def connect_to_db(self):
        self.adminManager.connect_to_db(self.current_db_path)
    def export_to_excel(self):
        file_path, _ = QFileDialog.getSaveFileName(self, "Save Excel
File", "", "Excel Files (*.xlsx)")
        if file_path:
            self.adminManager.write_to_excel(file_path)

    def bind_crud_buttons(self):
        self.ui.addButton.clicked.connect(self.add_row)
        self.ui.delButton.clicked.connect(self.delete_record)
        self.ui.pushButton_10.clicked.connect(self.update_record)

    def load_table_data(self, db_path, table_name, columns):
        self.current_db_path = db_path
        self.current_table = table_name
        self.current_columns = columns
        self.adminManager.connect_to_db(db_path)
        self.adminManager.loadDataFromDatabase(table_name, columns)

        self.ui.comboBox.clear()
        self.ui.comboBox.addItems(columns)

    def add_row(self):
        self.adminManager.addEmptyRow()
    def delete_record(self):
        if self.current_table:
            self.adminManager.deleteRecord(self.current_table,
self.current_columns[0])
    def update_record(self):
        if self.current_table and self.current_columns:
            self.adminManager.updateRecord(self.current_table,
self.current_columns[0], self.current_columns)

    @pyqtSlot()
    def search_data(self):
        if self.current_table and self.current_columns:
            keyword = self.ui.textEdit.toPlainText()
            selected_column = self.ui.comboBox.currentIndex()
            print(f"Selected column index: {selected_column}, column
name: {self.current_columns[selected_column]}")  # Debug information
            self.adminManager.searchData(self.current_table,
self.current_columns[selected_column], keyword)

    def close_window(self):
        event = QCloseEvent()
        self.closeEvent(event)

    def closeEvent(self, event):
        reply = QMessageBox.question(self, 'Message',
                                     "Are you sure you want to quit?",
QMessageBox.Yes | QMessageBox.No, QMessageBox.No)
        if reply == QMessageBox.Yes:
            self.adminManager.close_connection()
            QApplication.quit()
        else:
            event.ignore()

class Ui_adminWindow(object):
    def setupUi(self, adminWindow):
```

```
        adminWindow.setObjectName("adminWindow")
        adminWindow.resize(1600, 900)  # 调整窗口大小
       adminWindow.setMinimumSize(QtCore.QSize(1600, 900))
        adminWindow.setMaximumSize(QtCore.QSize(1600, 900))

        self.tableWidget = QTableWidget(adminWindow)
        self.tableWidget.setGeometry(QtCore.QRect(350, 150, 1200,
600))  # 调整表格的大小
       self.tableWidget.setObjectName("tableWidget")
        self.tableWidget.setColumnCount(0)
        self.tableWidget.setRowCount(0)

        self.verticalLayoutWidget = QWidget(adminWindow)
        self.verticalLayoutWidget.setGeometry(QtCore.QRect(10, 150,
330, 600))

self.verticalLayoutWidget.setObjectName("verticalLayoutWidget")
        self.verticalLayout = QVBoxLayout(self.verticalLayoutWidget)
        self.verticalLayout.setContentsMargins(0, 0, 0, 0)
        self.verticalLayout.setObjectName("verticalLayout")

        # 调整按钮大小和位置
       self.pushButton_3 = QPushButton(self.verticalLayoutWidget)
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(14)
        self.pushButton_3.setFont(font)
        self.pushButton_3.setObjectName("pushButton_3")
        self.verticalLayout.addWidget(self.pushButton_3)

        self.pushButton_5 = QPushButton(self.verticalLayoutWidget)
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(14)
        self.pushButton_5.setFont(font)
        self.pushButton_5.setObjectName("pushButton_5")
        self.verticalLayout.addWidget(self.pushButton_5)

        self.pushButton = QPushButton(self.verticalLayoutWidget)
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(14)
        self.pushButton.setFont(font)
        self.pushButton.setObjectName("pushButton")
        self.verticalLayout.addWidget(self.pushButton)

        self.pushButton_2 = QPushButton(self.verticalLayoutWidget)
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(14)
        self.pushButton_2.setFont(font)
        self.pushButton_2.setObjectName("pushButton_2")
        self.verticalLayout.addWidget(self.pushButton_2)

        self.pushButton_6 = QPushButton(self.verticalLayoutWidget)
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(14)
        self.pushButton_6.setFont(font)
        self.pushButton_6.setObjectName("pushButton_6")
        self.verticalLayout.addWidget(self.pushButton_6)
```

```
        self.pushButton_7 = QPushButton(self.verticalLayoutWidget)
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(14)
        self.pushButton_7.setFont(font)
        self.pushButton_7.setObjectName("pushButton_7")
        self.verticalLayout.addWidget(self.pushButton_7)

        self.pushButton_4 = QPushButton(adminWindow)
        self.pushButton_4.setGeometry(QtCore.QRect(1450, 10, 120, 50))
# 调整按钮的位置和大小
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(14)
        self.pushButton_4.setFont(font)
        self.pushButton_4.setObjectName("pushButton_4")

        # 调整搜索标签的位置和大小
        self.label = QLabel(adminWindow)
        self.label.setGeometry(QtCore.QRect(350, 70, 150, 31))
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(20)
        self.label.setFont(font)
        self.label.setObjectName("label")

        # 调整搜索框的位置和大小
        self.textEdit = QTextEdit(adminWindow)
        self.textEdit.setGeometry(QtCore.QRect(510, 70, 600, 31))
        font = QtGui.QFont()
        font.setFamily("Arial")
        self.textEdit.setFont(font)
        self.textEdit.setObjectName("textEdit")

        self.comboBox = QtWidgets.QComboBox(adminWindow)
        self.comboBox.setGeometry(QtCore.QRect(1120, 70, 150, 31))
        self.comboBox.setObjectName("comboBox")

        # 调整底部按钮的位置和大小
        self.addButton = QPushButton(adminWindow)
        self.addButton.setGeometry(QtCore.QRect(450, 800, 120, 50))
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(14)
        self.addButton.setFont(font)
        self.addButton.setObjectName("addButton")

        self.delButton = QPushButton(adminWindow)
        self.delButton.setGeometry(QtCore.QRect(650, 800, 120, 50))
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(14)
        self.delButton.setFont(font)
        self.delButton.setObjectName("delButton")

        self.pushButton_10 = QPushButton(adminWindow)
        self.pushButton_10.setGeometry(QtCore.QRect(850, 800, 120,
50))
        font = QtGui.QFont()
```

```
        font.setFamily("Arial")
        font.setPointSize(14)
        self.pushButton_10.setFont(font)
        self.pushButton_10.setObjectName("pushButton_10")

        self.retranslateUi(adminWindow)
        QtCore.QMetaObject.connectSlotsByName(adminWindow)

    def retranslateUi(self, adminWindow):
        _translate = QtCore.QCoreApplication.translate
        adminWindow.setWindowTitle(_translate("adminWindow", "Admin
Panel"))
        self.pushButton_3.setText(_translate("adminWindow", "Customer
Management"))
        self.pushButton_5.setText(_translate("adminWindow",
"Transaction Management"))
        self.pushButton.setText(_translate("adminWindow", "Bill
Management"))
        self.pushButton_2.setText(_translate("adminWindow", "Payment
Management"))
        self.pushButton_6.setText(_translate("adminWindow",
"Transaction History"))
        self.pushButton_7.setText(_translate("adminWindow", "Report
Generation"))
        self.pushButton_4.setText(_translate("adminWindow", "Exit"))
        self.label.setText(_translate("adminWindow", "Search"))
        self.addButton.setText(_translate("adminWindow", "Add"))
        self.delButton.setText(_translate("adminWindow", "Delete"))
        self.pushButton_10.setText(_translate("adminWindow",
"Update"))
```

7.2 auditWindow.py

```
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtCore import pyqtSlot
from PyQt5.QtGui import QCloseEvent
from PyQt5.QtWidgets import QWidget, QTableWidgetItem, QTableWidget,
QVBoxLayout, QPushButton, QLabel, QTextEdit, \
    QMessageBox, QApplication, QFileDialog
from dbmanager import DatabaseManager
import config
class auditWindow(QWidget):
    def __init__(self, switch_page, parent=None):
        super(auditWindow, self).__init__(parent)
        self.ui = Ui_adminWindow()
        self.ui.setupUi(self)
        self.adminManager = DatabaseManager(self.ui.tableWidget)
        self.current_table = None
        self.current_columns = None
        self.current_db_path = None
        self.current_db_path = 'management.db'
        self.connect_to_db()
        self.bind_main_buttons()
        # self.bind_crud_buttons()
        self.ui.textEdit.textChanged.connect(self.search_data)

    def bind_main_buttons(self):

        self.ui.pushButton_3.clicked.connect(lambda:
```

```
self.load_table_data('management.db', 'user_info', ['customer_id',
'customer_name',
'customer_mobile','customer_email','customer_username','customer_pass
word','customer_address']))
        self.ui.pushButton_5.clicked.connect(lambda:
self.load_table_data('management.db', 'transaction_info', [
            'transaction_id','transaction_customer_id',
'transaction_amount', 'transaction_bill',
            'transaction_number', 'transaction_type',
'transaction_history', 'transaction_description'
        ]))
        self.ui.pushButton.clicked.connect(lambda:
self.load_table_data('management.db', 'bill_info', ['bill_id',
'bill_customer_id',
'bill_number','bill_type','bill_receipt','bill_description','bill_amo
unt','unpaid_bill']))
        self.ui.pushButton_2.clicked.connect(lambda:
self.load_table_data('management.db', 'payment_info', ['payment_id',
'payment_customer_id', 'payment_des','payment_status']))
        self.ui.pushButton_6.clicked.connect(lambda:
self.load_table_data('management.db', 'transaction_history_info',
['transaction_history_id', 'transaction_history_customer_id',
'transaction_history_amount', 'transaction_history_bill',
'transaction_history_number', 'transaction_history_type',
'transaction_history_description']))
        self.ui.pushButton_7.clicked.connect(self.export_to_excel)
        self.ui.pushButton_4.clicked.connect(self.close_window)

    def connect_to_db(self):
        self.adminManager.connect_to_db(self.current_db_path)
    def export_to_excel(self):
        file_path, _ = QFileDialog.getSaveFileName(self, "Save Excel
File", "", "Excel Files (*.xlsx)")
        if file_path:
            self.adminManager.write_to_excel(file_path)

    def bind_crud_buttons(self):

        self.ui.addButton.clicked.connect(self.add_row)
        self.ui.delButton.clicked.connect(self.delete_record)
        self.ui.pushButton_10.clicked.connect(self.update_record)

    def load_table_data(self, db_path, table_name, columns):
        self.current_db_path = db_path
        self.current_table = table_name
        self.current_columns = columns
        self.adminManager.connect_to_db(db_path)
        self.adminManager.loadDataFromDatabase(table_name, columns)

        self.ui.comboBox.clear()
        self.ui.comboBox.addItems(columns)

    def add_row(self):
        self.adminManager.addEmptyRow()

    def delete_record(self):
        if self.current_table:
            self.adminManager.deleteRecord(self.current_table,
self.current_columns[0])

    def update_record(self):
```

```
        if self.current_table and self.current_columns:
            self.adminManager.updateRecord(self.current_table,
self.current_columns[0], self.current_columns)

    @pyqtSlot()
    def search_data(self):
        if self.current_table and self.current_columns:
            keyword = self.ui.textEdit.toPlainText()
            selected_column = self.ui.comboBox.currentIndex()
            print(f"Selected column index: {selected_column}, column
name: {self.current_columns[selected_column]}")  # Debug information
            self.adminManager.searchData(self.current_table,
self.current_columns[selected_column], keyword)

    def close_window(self):

        event = QCloseEvent()
        self.closeEvent(event)

    def closeEvent(self, event):
        reply = QMessageBox.question(self, 'Message',
                                     "Are you sure you want to quit?",
QMessageBox.Yes | QMessageBox.No, QMessageBox.No)
        if reply == QMessageBox.Yes:
            self.adminManager.close_connection()
            QApplication.quit()
        else:
            event.ignore()
class Ui_adminWindow(object):
    def setupUi(self, adminWindow):
        adminWindow.setObjectName("adminWindow")
        adminWindow.resize(1400, 800)  # 增加窗口宽度
       adminWindow.setMinimumSize(QtCore.QSize(1400, 800))  # 增加窗口
最小宽度
       adminWindow.setMaximumSize(QtCore.QSize(1400, 800))  # 增加窗口
最大宽度

       button_width = 300  # 增加按钮宽度
       button_height = 60  # 增加按钮高度
       button_font_size = 16  # 增加按钮字体大小

       self.tableWidget = QTableWidget(adminWindow)
        self.tableWidget.setGeometry(QtCore.QRect(320, 110, 1050,
541))  # 调整位置和大小
       self.tableWidget.setObjectName("tableWidget")
        self.tableWidget.setColumnCount(0)
        self.tableWidget.setRowCount(0)

        self.verticalLayoutWidget = QWidget(adminWindow)
        self.verticalLayoutWidget.setGeometry(QtCore.QRect(10, 110,
291, 541))

self.verticalLayoutWidget.setObjectName("verticalLayoutWidget")
        self.verticalLayout = QVBoxLayout(self.verticalLayoutWidget)
        self.verticalLayout.setContentsMargins(0, 0, 0, 0)
        self.verticalLayout.setObjectName("verticalLayout")

        self.pushButton_3 = QPushButton(self.verticalLayoutWidget)
        font = QtGui.QFont()
```

```
        font.setFamily("Arial")
        font.setPointSize(button_font_size)
        self.pushButton_3.setFont(font)
        self.pushButton_3.setObjectName("pushButton_3")
        self.pushButton_3.setMinimumSize(QtCore.QSize(button_width,
button_height))  # 设置按钮最小尺寸
      self.verticalLayout.addWidget(self.pushButton_3)

        self.pushButton_5 = QPushButton(self.verticalLayoutWidget)
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(button_font_size)
        self.pushButton_5.setFont(font)
        self.pushButton_5.setObjectName("pushButton_5")
        self.pushButton_5.setMinimumSize(QtCore.QSize(button_width,
button_height))  # 设置按钮最小尺寸
      self.verticalLayout.addWidget(self.pushButton_5)

        self.pushButton = QPushButton(self.verticalLayoutWidget)
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(button_font_size)
        self.pushButton.setFont(font)
        self.pushButton.setObjectName("pushButton")
        self.pushButton.setMinimumSize(QtCore.QSize(button_width,
button_height))  # 设置按钮最小尺寸
      self.verticalLayout.addWidget(self.pushButton)

        self.pushButton_2 = QPushButton(self.verticalLayoutWidget)
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(button_font_size)
        self.pushButton_2.setFont(font)
        self.pushButton_2.setObjectName("pushButton_2")
        self.pushButton_2.setMinimumSize(QtCore.QSize(button_width,
button_height))  # 设置按钮最小尺寸
      self.verticalLayout.addWidget(self.pushButton_2)

        self.pushButton_6 = QPushButton(self.verticalLayoutWidget)
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(button_font_size)
        self.pushButton_6.setFont(font)
        self.pushButton_6.setObjectName("pushButton_6")
        self.pushButton_6.setMinimumSize(QtCore.QSize(button_width,
button_height))  # 设置按钮最小尺寸
      self.verticalLayout.addWidget(self.pushButton_6)

        self.pushButton_7 = QPushButton(self.verticalLayoutWidget)
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(button_font_size)
        self.pushButton_7.setFont(font)
        self.pushButton_7.setObjectName("pushButton_7")
        self.pushButton_7.setMinimumSize(QtCore.QSize(button_width,
button_height))  # 设置按钮最小尺寸
      self.verticalLayout.addWidget(self.pushButton_7)

        self.pushButton_4 = QPushButton(adminWindow)
        self.pushButton_4.setGeometry(QtCore.QRect(1280, 10, 100, 51))
```

```
# 调整退出按钮的位置和大小
        font = QtGui.QFont()
         font.setFamily("Arial")
         font.setPointSize(button_font_size)
         self.pushButton_4.setFont(font)
         self.pushButton_4.setObjectName("pushButton_4")

         self.label = QLabel(adminWindow)
         self.label.setGeometry(QtCore.QRect(320, 70, 91, 31))  # 调整
搜索标签的位置
        font = QtGui.QFont()
         font.setFamily("Arial")
         font.setPointSize(20)
         self.label.setFont(font)
         self.label.setObjectName("label")

         self.textEdit = QTextEdit(adminWindow)
         self.textEdit.setGeometry(QtCore.QRect(400, 70, 641, 31))  #
调整搜索输入框的位置和大小
        font = QtGui.QFont()
         font.setFamily("Arial")
         self.textEdit.setFont(font)
         self.textEdit.setObjectName("textEdit")

         self.comboBox = QtWidgets.QComboBox(adminWindow)
         self.comboBox.setGeometry(QtCore.QRect(1090, 70, 100, 31))  #
调整下拉框的位置
        self.comboBox.setObjectName("comboBox")

         self.retranslateUi(adminWindow)
         QtCore.QMetaObject.connectSlotsByName(adminWindow)

    def retranslateUi(self, adminWindow):
         _translate = QtCore.QCoreApplication.translate
         adminWindow.setWindowTitle(_translate("adminWindow", "Form"))
         self.pushButton_3.setText(_translate("adminWindow", "Customer
Management"))
         self.pushButton_5.setText(_translate("adminWindow",
"Transaction Management"))
         self.pushButton.setText(_translate("adminWindow", "Bill
Management"))
         self.pushButton_2.setText(_translate("adminWindow", "Payment
Management"))
         self.pushButton_6.setText(_translate("adminWindow",
"Transaction History"))
         self.pushButton_7.setText(_translate("adminWindow", "Report
Generation"))
         self.pushButton_4.setText(_translate("adminWindow", "Exit"))
         self.label.setText(_translate("adminWindow", "Search"))
```

7.3 config.py

```
current_user_id = None
current_prem = None
```

7.4 customerWindow.py

```
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtCore import pyqtSlot
from PyQt5.QtGui import QCloseEvent
from PyQt5.QtWidgets import QWidget, QTableWidgetItem, QTableWidget,
QVBoxLayout, QPushButton, QLabel, QTextEdit, \
    QMessageBox, QApplication, QInputDialog, QLineEdit, QSpinBox,
QDialog, QComboBox
from dbmanager import DatabaseManager
import config
class customerWindow(QWidget):
    def __init__(self, switch_page, parent=None):
        super(customerWindow, self).__init__(parent)

        self.ui = Ui_customerWindow()
        self.ui.setupUi(self)
        self.bind_main_buttons()
        self.customerManager = DatabaseManager(self.ui.tableWidget)
        self.ui.textEdit.textChanged.connect(self.search_data)
        self.customerManager.connect_to_db('management.db')
        self.ui.pushButton_4.clicked.connect(self.close_window)
        self.ui.payButton_2.clicked.connect(self.show_bills_to_pay)
        self.status = None
    def bind_main_buttons(self):
        self.ui.billButton.clicked.connect(lambda:
self.update_status_and_load_table(
            'management.db',
            'bill_info',
            ['bill_id', 'bill_customer_id', 'bill_type',
'bill_receipt', 'bill_description', 'bill_amount', 'unpaid_bill'],
            {'bill_customer_id': config.current_user_id},
            'bill'
        ))
        self.ui.transactionButton.clicked.connect(lambda:
self.update_status_and_load_table(
            'management.db',
            'transaction_info',
            [
                'transaction_id','transaction_customer_id',
'transaction_amount', 'transaction_bill','transaction_number',
                'transaction_type', 'transaction_history',
'transaction_description'
            ],
            {'transaction_customer_id': config.current_user_id},
            'transaction'
        ))

        self.ui.pushButton_4.clicked.connect(self.close)

    def bind_crud_buttons(self):

        self.ui.addButton.clicked.connect(self.add_row)
        self.ui.delButton.clicked.connect(self.delete_record)
        self.ui.pushButton_10.clicked.connect(self.update_record)
```

```
    def update_status_and_load_table(self, db_path, table_name,
columns, filters, status):
        self.status = status
        print("status is", self.status)
        self.load_table_data(db_path, table_name, columns, filters)

    def load_table_data(self, db_path, table_name, columns,
filters=None):
        self.current_db_path = db_path
        self.current_table = table_name
        self.current_columns = columns
        self.customerManager.connect_to_db(db_path)
        self.customerManager.loadDataFromDatabase(table_name, columns,
filters)
        print("current_id is",config.current_user_id)
        print(filters)

        self.ui.comboBox.clear()
        self.ui.comboBox.addItems(columns)

    def add_row(self):
        self.customerManager.addEmptyRow()

    def delete_record(self):
        if self.current_table:
            self.customerManager.deleteRecord(self.current_table,
self.current_columns[0])

    def update_record(self):
        if self.current_table and self.current_columns:
            self.customerManager.updateRecord(self.current_table,
self.current_columns[0], self.current_columns)

    @pyqtSlot()
    def search_data(self):
        if self.current_table and self.current_columns:
            keyword = self.ui.textEdit.toPlainText()
            selected_column = self.ui.comboBox.currentIndex()

            # Debug information
            print(f"Selected column index: {selected_column}, column
name: {self.current_columns[selected_column]}")

            # Assuming you have a method `getFilters` that retrieves
the filter conditions from the UI
            filters = self.getFilters()

            # Pass the filters to the searchData method
            self.customerManager.searchData(self.current_table,
self.current_columns[selected_column], keyword, filters)

    def getFilters(self):
        filters = {}
        if self.status == 'bill':
            filters['bill_customer_id'] = config.current_user_id
        elif self.status == 'transaction':
            filters['transaction_customer_id'] =
```

```
config.current_user_id
        print("filters                    is",                    filters)
        return                                                     filters

    def                                                close_window(self):

        event                          =                        QCloseEvent()
        self.closeEvent(event)

    def                        closeEvent(self,                        event):
        reply       =       QMessageBox.question(self,       'Message',
                                    "Are you sure you want to quit?",
QMessageBox.Yes        |        QMessageBox.No,        QMessageBox.No)
        if             reply             ==             QMessageBox.Yes:
            self.customerManager.close_connection()
            QApplication.quit()
        else:
            event.ignore()

    def                                              show_bills_to_pay(self):

        bills                                                              =
self.customerManager.get_bills_to_pay(config.current_user_id)
        if                              not                                bills:
            QMessageBox.warning(self,  "No  Bill  to  Pay",  "No  bill  to
pay                for                this                customer.")
            return


        dialog                         =                        QDialog(self)
        dialog.setWindowTitle("Bills              to              Pay")
        layout                         =                        QVBoxLayout()


        bill_selector_label   =   QLabel("Select   a   bill   to   pay:")
        layout.addWidget(bill_selector_label)

        bill_selector                      =                     QComboBox()
        bill_dict                             =                          {}
        for                bill                in                bills:
            bill_id,       bill_amount,       unpaid       =       bill
            bill_amount   =   float(bill_amount)     # 转换为浮点数
              unpaid    =    float(unpaid)       #    转    换    为    浮    点    数
              amount_due        =        bill_amount        -        unpaid
              bill_desc    =    f"Bill    ID:    {bill_id},    Amount    Due:
{amount_due},                      Unpaid:                          {unpaid}"
            bill_selector.addItem(bill_desc,                   bill_id)
            bill_dict[bill_id]       =       (bill_amount,       unpaid)
        layout.addWidget(bill_selector)


        amount_label     =     QLabel("Enter     amount     to     pay:")
        amount_input                       =                     QSpinBox()
        amount_input.setMaximum(1000000)
        layout.addWidget(amount_label)
        layout.addWidget(amount_input)
```

```
        button = QPushButton("Pay")
        layout.addWidget(button)
        button.clicked.connect(lambda:
self.process_payment(bill_selector.currentData(),
amount_input.value(), bill_dict))
        dialog.setLayout(layout)
        dialog.exec_()

    def process_payment(self, bill_id, amount, bill_dict):
        bill_amount, unpaid = bill_dict[bill_id]
        if amount <= bill_amount:

            new_unpaid = unpaid - amount
            # Update unpaid_bill in bill_info table
            update_query_bill_info = "UPDATE bill_info SET
unpaid_bill = ? WHERE bill_id = ?"
            self.customerManager.cur.execute(update_query_bill_info,
(new_unpaid, bill_id))

            # Update transaction_history_bill in
transaction_history_info table
            update_query_transaction_history = "UPDATE
transaction_history_info SET transaction_history_bill = ? WHERE
transaction_history_id = ?"

self.customerManager.cur.execute(update_query_transaction_history,
(new_unpaid, bill_id))

            # Commit the changes
            self.customerManager.conn.commit()

            QMessageBox.information(None, "Payment Processed",
                                    f"Payment of {amount} has been
processed. New unpaid amount: {new_unpaid}")
        else:
            QMessageBox.warning(None, "Payment Error", "Amount
exceeds the bill amount.")


class Ui_customerWindow(object):
    def setupUi(self, customerWindow):
        customerWindow.setObjectName("customerWindow")
        customerWindow.resize(1200, 800)
        customerWindow.setMinimumSize(QtCore.QSize(1200, 800))
        customerWindow.setMaximumSize(QtCore.QSize(1200, 800))

        self.tableWidget = QtWidgets.QTableWidget(customerWindow)
        self.tableWidget.setGeometry(QtCore.QRect(300, 110, 891, 541))
        self.tableWidget.setObjectName("tableWidget")
        self.tableWidget.setColumnCount(0)
        self.tableWidget.setRowCount(0)

        self.verticalLayoutWidget = QtWidgets.QWidget(customerWindow)
        self.verticalLayoutWidget.setGeometry(QtCore.QRect(0, 110,
291, 541))
```

```
self.verticalLayoutWidget.setObjectName("verticalLayoutWidget")

        self.verticalLayout                                                =
QtWidgets.QVBoxLayout(self.verticalLayoutWidget)
        self.verticalLayout.setContentsMargins(0,     0,     0,     0)
        self.verticalLayout.setObjectName("verticalLayout")

        self.billButton                                                    =
QtWidgets.QPushButton(self.verticalLayoutWidget)
        font                        =                        QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(18)
        self.billButton.setFont(font)
        self.billButton.setObjectName("billButton")
        self.verticalLayout.addWidget(self.billButton)

        self.transactionButton                                             =
QtWidgets.QPushButton(self.verticalLayoutWidget)
        font                        =                        QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(18)
        self.transactionButton.setFont(font)
        self.transactionButton.setObjectName("transactionButton")
        self.verticalLayout.addWidget(self.transactionButton)

        self.pushButton_4    =    QtWidgets.QPushButton(customerWindow)
        self.pushButton_4.setGeometry(QtCore.QRect(1108,  0,  91,  51))
        font                        =                        QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(18)
        self.pushButton_4.setFont(font)
        self.pushButton_4.setObjectName("pushButton_4")

        self.textEdit  =  QtWidgets.QLineEdit(customerWindow)   # Using
QLineEdit
        self.textEdit.setGeometry(QtCore.QRect(430,   70,   641,   31))
        font                        =                        QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(18)          #     Adjusting     font     size
        self.textEdit.setFont(font)
        self.textEdit.setPlaceholderText("Enter search text here...")
#                Adding                placeholder                text
        self.textEdit.setAlignment(QtCore.Qt.AlignLeft)    #  Aligning
text                    to                    the                    left
        self.textEdit.setObjectName("textEdit")

        self.comboBox       =       QtWidgets.QComboBox(customerWindow)
        self.comboBox.setGeometry(QtCore.QRect(1090,  70,  100,  31))
        self.comboBox.setObjectName("comboBox")

        self.label          =          QtWidgets.QLabel(customerWindow)
        self.label.setGeometry(QtCore.QRect(330,    70,    91,    31))
        font                        =                        QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(20)
        self.label.setFont(font)
```

```
        self.label.setObjectName("label")

        self.payButton_2    =    QtWidgets.QPushButton(customerWindow)
        self.payButton_2.setGeometry(QtCore.QRect(600, 680, 289, 35))
        font                        =                        QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(18)
        self.payButton_2.setFont(font)
        self.payButton_2.setObjectName("payButton_2")

        self.retranslateUi(customerWindow)
        QtCore.QMetaObject.connectSlotsByName(customerWindow)

    def              retranslateUi(self,              customerWindow):
        _translate         =         QtCore.QCoreApplication.translate
        customerWindow.setWindowTitle(_translate("customerWindow",
"Form"))
        self.billButton.setText(_translate("customerWindow",     "Bill
Check"))
        self.transactionButton.setText(_translate("customerWindow",
"Transaction                                                    Check"))
        self.pushButton_4.setText(_translate("customerWindow",
"exit"))
        self.label.setText(_translate("customerWindow",     "Search"))
        self.payButton_2.setText(_translate("customerWindow", "Pay"))
```

7.5 databases.py

```
import sqlite3
conn = sqlite3.connect('management.db')
c = conn.cursor()
c.execute('''CREATE TABLE IF NOT EXISTS users
             (id INTEGER PRIMARY KEY, user TEXT, password TEXT, perm
INTEGER)''')

c.execute('''CREATE TABLE IF NOT EXISTS user_info
             (id INTEGER PRIMARY KEY, customer_id TEXT, customer_name
TEXT, customer_mobile TEXT, customer_email TEXT, customer_username
TEXT, customer_password TEXT, customer_address TEXT)''')

c.execute('''CREATE TABLE IF NOT EXISTS transaction_info
             (id INTEGER PRIMARY KEY,
             transaction_id TEXT,
              transaction_customer_id TEXT,
              transaction_amount TEXT,
              transaction_bill TEXT,
              transaction_number TEXT,
              transaction_type TEXT,
              transaction_history TEXT,
              transaction_description TEXT)''')

c.execute('''CREATE TABLE IF NOT EXISTS bill_info
             (id INTEGER PRIMARY KEY,
              bill_id TEXT,
              bill_customer_id TEXT,
              bill_number TEXT,
              bill_type TEXT,
```

```
                bill_receipt TEXT,
                bill_description TEXT,
                bill_amount TEXT,
                unpaid_bill TEXT)''')

c.execute('''CREATE TABLE IF NOT EXISTS payment_info
               (id INTEGER PRIMARY KEY,
                payment_id TEXT,
                payment_customer_id TEXT,
                payment_des TEXT,
                payment_status TEXT)''')

c.execute('''CREATE TABLE IF NOT EXISTS transaction_history_info
               (id INTEGER PRIMARY KEY,
                transaction_history_id TEXT,
                transaction_history_customer_id TEXT,
                transaction_history_amount TEXT,
                transaction_history_bill TEXT,
                transaction_history_number TEXT,
                transaction_history_type TEXT,
                transaction_history_description TEXT,
                unpaid_bill TEXT)''')



conn.commit()


conn.close()
```

7.6 dbmanager.py

```
"""

"""
import sqlite3
from PyQt5.QtWidgets import QMessageBox, QTableWidgetItem
import pandas as pd
current_user_id = None
current_prem = None
class DBmanager:
    def __init__(self):
        self.createDatabase()

    def createDatabase(self):
        # 创建数据库连接并创建表格
       self.conn = sqlite3.connect('management.db')
        self.cur = self.conn.cursor()
        self.cur.execute('''CREATE TABLE IF NOT EXISTS users
                            (id INTEGER PRIMARY KEY, name TEXT, age
INTEGER)''')
        self.conn.commit()

    def addUser(self, name, password, perm):

        self.cur.execute("INSERT INTO users (user, password, perm)
VALUES (?, ?, ?)", (name, password, perm))
        self.conn.commit()
        return self.cur.lastrowid
```

```
    def addUserInfo(self,customer_id, name, password):
        self.cur.execute("INSERT INTO user_info (customer_id,
customer_username, customer_password) VALUES (?, ?, ?)", (customer_id,
name, password))
        self.conn.commit()


    def loadDataFromAdmin(self):
        self.cur.execute("SELECT * FROM users")
        data = self.cur.fetchall()
        return data

    def loadDataFromUser(self):
        self.cur.execute("SELECT * FROM user_info")
        data = self.cur.fetchall()
        return data

class DatabaseManager:
    def __init__(self, tableWidget):
        self.tableWidget = tableWidget
        self.conn = None
        self.cur = None
    def connect_to_db(self, db_path):
        self.conn = sqlite3.connect(db_path)
        self.cur = self.conn.cursor()

    def loadDataFromDatabase(self, table_name, columns, filters=None):
        self.tableWidget.setRowCount(0)
        self.tableWidget.setColumnCount(len(columns))
        self.tableWidget.setHorizontalHeaderLabels(columns)


        all_columns = ["id"] + columns


        query = f"SELECT {', '.join(all_columns)} FROM {table_name}"


        if filters:
            filter_conditions = []
            for key, value in filters.items():
                filter_conditions.append(f"{key} = '{value}'")
            filter_clause = " AND ".join(filter_conditions)
            query += f" WHERE {filter_clause}"

        self.cur.execute(query)
        data = self.cur.fetchall()

        for row, record in enumerate(data):
            self.addRow(row, record[1:])
        self.adjustColumnWidths()

    def addRow(self, row, record):
        self.tableWidget.insertRow(row)
        for col, value in enumerate(record):
            self.tableWidget.setItem(row, col,
QTableWidgetItem(str(value)))

    def addEmptyRow(self):
        row_position = self.tableWidget.rowCount()
```

```
        self.tableWidget.insertRow(row_position)

        for col in range(self.tableWidget.columnCount()):
            self.tableWidget.setItem(row_position, col,
QTableWidgetItem(""))

    def addRecord(self, table_name, column_names, values):
        columns_str = ', '.join(column_names)
        placeholders = ', '.join('?' * len(values))
        self.cur.execute(f"INSERT INTO {table_name} ({columns_str})
VALUES ({placeholders})", values)
        self.conn.commit()
        self.loadDataFromDatabase(table_name, column_names)

    def deleteRecord(self, table_name, id_column):
        current_row = self.tableWidget.currentRow()
        if current_row != -1:
            id_value = self.tableWidget.item(current_row, 0).text()
            self.cur.execute(f"DELETE FROM {table_name} WHERE
{id_column}=?", (id_value,))
            self.conn.commit()
            self.tableWidget.removeRow(current_row)
        else:
            QMessageBox.warning(self.tableWidget, "Warning", "Please
select a row to delete.")

    def updateRecord(self, table_name, id_column, columns):
        current_row = self.tableWidget.currentRow()
        if current_row != -1:

            values = [self.tableWidget.item(current_row, col).text()
for col in range(len(columns))]

            query = f"SELECT id FROM {table_name} WHERE {columns[0]}
= ?"
            self.cur.execute(query,
(self.tableWidget.item(current_row, 0).text(),))
            result = self.cur.fetchone()

            if result:
                id_value = result[0]
                set_clause = ', '.join(f"{col}=?" for col in columns)
                update_query = f"UPDATE {table_name} SET {set_clause}
WHERE id=?"
                self.cur.execute(update_query, (*values, id_value))
            else:
                columns_str = ', '.join(columns)
                placeholders = ', '.join('?' * len(values))
                insert_query = f"INSERT INTO {table_name}
({columns_str}) VALUES ({placeholders})"
                self.cur.execute(insert_query, values)

            # 更新 transaction_history_info 表
          if table_name == 'bill_info':
                print(values)
                transaction_history_data = {
                    'transaction_history_id': values[0],
                    'transaction_history_customer_id': values[1],  #
使用 bill_info 的某些字段更新 transaction_history_info
                    'transaction_history_amount': values[6],
                    'transaction_history_bill': values[6],  # 假设
```

```
bill_info 的第 7 列是未支付的账单信息
                    'transaction_history_number': values[2],  # 假设
bill_info 的第 3 列是账单号
                    'transaction_history_type': values[3],  # 假设
bill_info 的第 4 列是账单类型
                    'transaction_history_description': values[5]  # 假设
bill_info 的第 6 列是账单描述
                }
                    self.cur.execute(
                        "INSERT INTO transaction_history_info
(transaction_history_id, transaction_history_customer_id,
transaction_history_amount, transaction_history_bill,
transaction_history_number, transaction_history_type,
transaction_history_description) VALUES (?, ?, ?, ?, ?, ?, ?)",

(transaction_history_data['transaction_history_id'],

transaction_history_data['transaction_history_customer_id'],

transaction_history_data['transaction_history_amount'],

transaction_history_data['transaction_history_bill'],

transaction_history_data['transaction_history_number'],

transaction_history_data['transaction_history_type'],

transaction_history_data['transaction_history_description']))

            self.conn.commit()
            self.loadDataFromDatabase(table_name, columns)
            QMessageBox.information(self.tableWidget, "Update",
"Record updated successfully.")
        else:
            QMessageBox.warning(self.tableWidget, "Warning", "Please
select a row to update.")

    def searchData(self, table_name, search_column, keyword,
filters=None):
        print(f"Searching in table: {table_name}, column:
{search_column}, for keyword: {keyword}")  # Debug information

        query = f"SELECT {',
'.join(self.exclude_id_column(table_name))} FROM {table_name} WHERE
{search_column} LIKE ?"
        params = ['%' + keyword + '%']

        # If filters are provided, add them to the query
        if filters:
            for filter_column, filter_value in filters.items():
                query += f" AND {filter_column} = ?"
                params.append(filter_value)

        print(f"Query: {query}, Params: {params}")  # Debug
information
        self.cur.execute(query, params)
        data = self.cur.fetchall()
        print(f"Search results: {data}")
        self.tableWidget.setRowCount(0)
        for row, record in enumerate(data):
```

```
                self.addRow(row, record)
            self.adjustColumnWidths()

    def exclude_id_column(self, table_name):
        self.cur.execute(f"PRAGMA table_info({table_name})")
        columns_info = self.cur.fetchall()
        return [info[1] for info in columns_info if info[1] != 'id']

    def adjustColumnWidths(self):

        for i in range(self.tableWidget.columnCount()):
            self.tableWidget.setColumnWidth(i, 180)

    def close_connection(self):
        if self.conn:
            self.conn.close()

    def get_bills_to_pay(self, bill_customer_id):

        query = "SELECT bill_id, bill_amount, unpaid_bill FROM
bill_info WHERE bill_customer_id = ? AND (unpaid_bill - bill_amount)
<= 0"
        self.cur.execute(query, (bill_customer_id,))
        return self.cur.fetchall()

    def process_payment(self, bill_id, amount):

        query = "SELECT bill_amount, unpaid_bill FROM bill_info WHERE
bill_id = ?"
        self.cur.execute(query, (bill_id,))
        result = self.cur.fetchone()

        if result:
            bill_amount, unpaid = result
            bill_amount = float(bill_amount)
            unpaid = float(unpaid)
            amount = float(amount)

            if unpaid + amount <= bill_amount:

                new_unpaid = amount - unpaid
                update_query = "UPDATE bill_info SET unpaid_bill = ?
WHERE bill_id = ?"
                self.cur.execute(update_query, (new_unpaid, bill_id))
                self.conn.commit()
                QMessageBox.information(None, "Payment Processed",
                                        f"Payment of {amount} has
been processed. New unpaid amount: {new_unpaid}")
            else:
                QMessageBox.warning(None, "Payment Error", "Amount
exceeds the bill amount.")
        else:
            QMessageBox.warning(None, "Payment Error", "Bill ID not
found.")

    def loadAllData(self):
        tables = {
            'bill_info': ['bill_id', 'bill_customer_id',
'bill_number', 'bill_type', 'bill_receipt', 'bill_description',
                          'bill_amount', 'unpaid_bill'],
            'transaction_info': ['transaction_id',
```

```
'transaction_customer_id', 'transaction_amount', 'transaction_bill',
                                'transaction_number',
'transaction_type', 'transaction_history',
                                'transaction_description'],
            'user_info': ['customer_id', 'customer_name',
'customer_mobile', 'customer_email', 'customer_username',
                          'customer_password', 'customer_address']
        }

        all_data = {}
        for table, columns in tables.items():
            query = f"SELECT {', '.join(columns)} FROM {table}"
            self.cur.execute(query)
            data = self.cur.fetchall()
            all_data[table] = (columns, data)

        return all_data

    def write_to_excel(self, file_path):
        all_data = self.loadAllData()
        with pd.ExcelWriter(file_path) as writer:
            for table_name, (columns, data) in all_data.items():
                df = pd.DataFrame(data, columns=columns)
                df.to_excel(writer, sheet_name=table_name,
index=False)
        QMessageBox.information(None, "Excel Export", f"Data
successfully exported to {file_path}")
```

7.7 loginWindow.py

```
"""

"""
from      PyQt5      import      QtCore,      QtGui,      QtWidgets
from            PyQt5.QtGui            import            QCloseEvent
from   PyQt5.QtWidgets   import   QWidget,   QMessageBox,   QApplication
from          dbmanager          import          DBmanager,current_prem
import                                                          config
class                                                  loginWindow(QWidget):
    def      __init__(self,      switch_page,      parent      =None):
        super(loginWindow,                          self).__init__(parent)

        self.ui                    =                   Ui_loginWindow()
        self.ui.setupUi(self)
        #                  self.ui.pushButton.clicked.connect(lambda:
switch_page("register"))
        self.ui.pushButton_2.clicked.connect(self.logIN)
        self.db_manager                   =                   DBmanager()
        self.switch_page                   =                   switch_page
        #     self.ui.pushButton_4.clicked.connect(self.close_window)

    def                                                        logIN(self):
        username           =           self.ui.textEdit.toPlainText()
        password          =          self.ui.textEdit_2.toPlainText()

        Admin_info        =        self.db_manager.loadDataFromAdmin()
        User_info         =         self.db_manager.loadDataFromUser()
```

```
        for admin_info in Admin_info:
            db_username, db_password, perm = admin_info[1],
admin_info[2], admin_info[3]
            if username == db_username and password == db_password:
                if perm == 1:
                    config.current_prem = perm
                    print("prem is",config.current_prem)
                    self.switch_page("admin")
                    return
                elif perm == 2:
                    config.current_prem = perm
                    print("prem is",config.current_prem)
                    self.switch_page("audit")
                    return
        for user_info in User_info:
            C_username, C_password = user_info[5], user_info[6]
            if username == C_username and password == C_password:
                config.current_user_id = user_info[1]
                print(config.current_user_id)
                self.switch_page("customer")
                return
        QMessageBox.warning(self, "Login", "Invalid username or
password.")

class Ui_loginWindow(object):
    def setupUi(self, loginWindow):
        loginWindow.setObjectName("loginWindow")
        loginWindow.resize(1600, 900)
        loginWindow.setMinimumSize(QtCore.QSize(1600, 900))
        loginWindow.setMaximumSize(QtCore.QSize(1600, 900))

        # 设置用户名标签的位置和大小
        self.label = QtWidgets.QLabel(loginWindow)
        self.label.setGeometry(QtCore.QRect(500, 300, 200, 50))
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(20)
        self.label.setFont(font)# 将字体应用到标签
        self.label.setObjectName("label")

        # 设置用户名文本框的位置和大小
        self.textEdit = QtWidgets.QTextEdit(loginWindow)
        self.textEdit.setGeometry(QtCore.QRect(700, 300, 400, 50))
        self.textEdit.setObjectName("textEdit")
        font = QtGui.QFont()
        font.setPointSize(20)
        self.textEdit.setFont(font)      # 设置字体和大小

        # 设置密码标签的位置和大小
        self.label_2 = QtWidgets.QLabel(loginWindow)
        self.label_2.setGeometry(QtCore.QRect(500, 400, 200, 50))
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(20)
        self.label_2.setFont(font)
```

```
        self.label_2.setObjectName("label_2")

        # 设置密码文本框的位置和大小
        self.textEdit_2 = QtWidgets.QTextEdit(loginWindow)
        self.textEdit_2.setGeometry(QtCore.QRect(700, 400, 400, 50))
        self.textEdit_2.setObjectName("textEdit_2")
        font = QtGui.QFont()
        font.setPointSize(20)
        self.textEdit_2.setFont(font)      # 设置字体和大小

        # 设置登录按钮的位置和大小
        self.pushButton_2 = QtWidgets.QPushButton(loginWindow)
        self.pushButton_2.setGeometry(QtCore.QRect(850, 500, 100, 50))
        font = QtGui.QFont()
        font.setFamily("Arial")
        font.setPointSize(20)
        self.pushButton_2.setFont(font)
        self.pushButton_2.setObjectName("pushButton_2")

        self.retranslateUi(loginWindow)#cccc
        QtCore.QMetaObject.connectSlotsByName(loginWindow)#ccc

    def retranslateUi(self, loginWindow):
        _translate = QtCore.QCoreApplication.translate
        loginWindow.setWindowTitle(_translate("loginWindow", "Form"))
        self.label.setText(_translate("loginWindow", "username"))
        self.label_2.setText(_translate("loginWindow", "password"))
        self.pushButton_2.setText(_translate("loginWindow", "LogIn"))
```

7.8 main.py

```
import sys
import sqlite3
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout,
QHBoxLayout, QPushButton, QTableWidget, QTableWidgetItem, QMessageBox,
QLineEdit
from PyQt5.QtWidgets import QApplication, QDialog, QStackedWidget,
QMainWindow
from adminWindow import adminWindow
from customerWindow import customerWindow
from loginWindow import loginWindow
from registerWindow import registerWindow
from auditWindow import auditWindow
class PageManager(QMainWindow):
    def __init__(self):
        super().__init__()
        self.pages = {}
        self.stacked_widget = QStackedWidget(self)
        self.init_pages()
        self.setCentralWidget(self.stacked_widget)

    def init_pages(self):
        self.register_page("admin", adminWindow)
        self.register_page("register", registerWindow)
        self.register_page("customer", customerWindow)
        self.register_page("login", loginWindow)
```

```
        self.register_page("audit",auditWindow)

        self.switch_page("login")

    def register_page(self, name, page_class):
        page = page_class(self.switch_page, self)
        self.pages[name] = page
        self.stacked_widget.addWidget(page)


    def switch_page(self, name):
        print(f"Attempting to switch to page: {name}")
        page = self.pages.get(name)
        if page:
            self.stacked_widget.setCurrentWidget(page)

class MyTableWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.createDatabase()

    def createDatabase(self):

        self.conn = sqlite3.connect('management.db')
        self.cur = self.conn.cursor()
        self.cur.execute('''CREATE TABLE IF NOT EXISTS users
                            (id INTEGER PRIMARY KEY, name TEXT, age
INTEGER)''')
        self.conn.commit()

    def loadDataFromDatabase(self):

        self.tableWidget.setRowCount(0)  # 清空表格内容
       self.cur.execute("SELECT * FROM users")
        data = self.cur.fetchall()
        for row, (id, name, age) in enumerate(data):
            self.addRow(row, id, name, age)

    def addRow(self, row, id, name, age):

        self.tableWidget.insertRow(row)
        self.tableWidget.setItem(row, 0, QTableWidgetItem(str(id)))
        self.tableWidget.setItem(row, 1, QTableWidgetItem(name))
        self.tableWidget.setItem(row, 2, QTableWidgetItem(str(age)))

    def addRecord(self):

        self.cur.execute("INSERT INTO users (name, age) VALUES
(?, ?)", ("", 0))
        self.conn.commit()
        self.loadDataFromDatabase()

    def addUser(self):

        self.cur.execute("INSERT INTO users (user, password,perm)
VALUES (?, ?, ?)", ("", 0))
        self.conn.commit()
        self.loadDataFromDatabase()

    def deleteRecord(self):
```

```
        current_row = self.tableWidget.currentRow()

        if current_row != -1:
            id = self.tableWidget.item(current_row, 0).text()
            self.cur.execute("DELETE FROM users WHERE id=?", (id,))
            self.conn.commit()
            self.loadDataFromDatabase()
        else:
            QMessageBox.warning(self, "Warning", "Please select a row
to delete.")

    def updateRecord(self):

        current_row = self.tableWidget.currentRow()
        if current_row != -1:
            id = self.tableWidget.item(current_row, 0).text()
            name = self.tableWidget.item(current_row, 1).text()
            age = self.tableWidget.item(current_row, 2).text()
            self.cur.execute("UPDATE users SET name=?, age=? WHERE
id=?", (name, age, id))
            self.conn.commit()
            QMessageBox.information(self, "Update", f"ID: {id}, Name:
{name}, Age: {age}")
        else:
            QMessageBox.warning(self, "Warning", "Please select a row
to update.")

    def searchData(self):

        keyword = self.searchInput.text()
        self.cur.execute("SELECT * FROM users WHERE name LIKE ?",
('%'+keyword+'%',))
        data = self.cur.fetchall()
        self.tableWidget.setRowCount(0)  # 清空表格内容
       for row, (id, name, age) in enumerate(data):
            self.addRow(row, id, name, age)

    def closeEvent(self, event):
        reply = QMessageBox.question(self, 'Message',
                                     "Are you sure you want to quit?",
QMessageBox.Yes | QMessageBox.No, QMessageBox.No)
        if reply == QMessageBox.Yes:
            self.customerManager.close_connection()
            QApplication.quit()
        else:
            event.ignore()


if __name__ == "__main__":
    app = QApplication(sys.argv)
    tableApp = PageManager()
    tableApp.show()
    sys.exit(app.exec_())
```