|  |  |  |
| --- | --- | --- |
|  | **KONGU ENGINEERING COLLEGE**  **(Autonomous)**  **Perundurai, Erode – 638 060**  **DEPARTMENT OF INFORMATION TECHNOLOGY** | KEC | Kongu Engineering College |

**BANK MANAGEMENT SYSTEM**

**AN MICRO PROJECT REPORT**

**For**

**DATA STRUCTURES USING JAVA(22ITC32)**

**Submitted by**

**DHARANISH S [23ITR027]**

**GIRI VIGNESH P.S [23ITR046]**

**HANSIGA P [23ITR051]**

|  |  |  |
| --- | --- | --- |
|  | **KONGU ENGINEERING COLLEGE**  **(Autonomous)**  **Perundurai, Erode – 638 060**  **DEPARTMENT OF INFORMATION TECHNOLOGY** | KEC | Kongu Engineering College |

**BONAFIDE CERTIFICATE**

Name : **DHARANISH S [23ITR027]**

**: GIRI VIGNESH P.S [23ITR046]**

**: HANSIGA P [23ITR027]**

Course Code : **22ITC31**

Course Name : **DATA STRUCTURES USING JAVA**

Semester : **III**

Certified that this is a bonafide record of work for application project done by the above students for **22ITC32 – DATA STRUCTURES USING JAVA** during the academic year **2024 - 2025.**

Submitted for the Viva Voce Examination held on \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Faculty In-Charge Head of the Department

| **Chapter No.** | **Title** | **PageNo.** |
| --- | --- | --- |
|  |  |  |
| 1 | **ABSTRACT** | 6 |
| 2 | **PROBLEM STATEMENT** | 7 |
|  | 2.1 Background |  |
|  | 2.2 Problem Identification |  |
|  | 2.3 Past Solutions |  |
|  | 2.4 Current Status of Bank Management Systems |  |
|  | 2.5 Solution Overview |  |
|  | 2.6 Present Challenges and Limitations |  |
| 3 | **METHODOLOGY** | 12 |
|  | 3.1 Object-Oriented Design |  |
|  | 3.2 Graphical User Interface (GUI) |  |
|  | 3.3 Event Handling and User Interaction |  |
|  | 3.4 Bank Management and Logic |  |
|  | 3.5 Data Reporting and Analytics Page |  |
| 4 | **IMPLEMENTATION** | 19 |
|  | 4.1 Overview of the System Design |  |
|  | 4.2 Classes and Components |  |
|  | 4.3 Detailed Functionality |  |
|  | 4.4 User Interface (UI) Design |  |
|  | 4.5 Error Handling |  |
| 5 | **RESULTS AND DISCUSSION** | 24 |
|  | 5.1 System Overview and Key Features |  |
|  | 5.2 Customers Adding System |  |
|  | 5.3 Comparison with Existing Methods |  |
|  | 5.4 Limitations and Areas for Improvement |  |
| 6 | **CONCLUSION** | 28 |
| 7 | **SAMPLE CODING** | 30 |

**1.ABSTRACT**

In today’s rapidly evolving financial landscape, banks play an essential role in managing customer accounts, processing transactions, and ensuring data security. However, traditional banking systems often face challenges such as data redundancy, inefficient processing, and security risks, which can undermine the customer experience and expose financial institutions to vulnerabilities. This project presents a Bank Management System (BMS) developed in Java, using the Abstract Window Toolkit (AWT) for the graphical user interface (GUI) and MySQL as the database backend, to address these challenges effectively.

The primary objective of this project is to create a modular, secure, and user-friendly platform that simplifies bank operations by efficiently managing customer accounts, transactions, and data storage. Through Java’s object-oriented programming capabilities, the system employs a structured approach, organizing data into modular classes such as Customer, Account, and Transaction, which allows for streamlined data handling and retrieval. Additionally, the use of MySQL provides a lightweight, relational database that ensures data consistency and integrity, addressing common issues such as redundancy and data discrepancies found in traditional systems. The system also incorporates basic authentication and access control to secure sensitive customer information, protecting against unauthorized access.

Methodologically, this project involved designing and implementing core banking functions, including account creation, transaction processing (deposits, withdrawals, transfers), and balance inquiries. Each function was tested to ensure real-time processing and responsiveness, key requirements in modern banking systems. Java’s modular structure supports scalability, making it easy to integrate new features or scale the system to accommodate growing customer data without compromising performance.

The results of this project show a marked improvement over traditional systems in terms of transaction speed, security, and data management. The modular design enhances maintainability, while the use of SQL-based data storage promotes data integrity and quick retrieval. In comparison to legacy systems, which often rely on outdated methods for managing records, this Java-based BMS demonstrates the potential for an adaptable, scalable solution that meets the operational and security demands of today’s digital banking environment.

This BMS project not only offers a viable solution for streamlining banking operations but also provides a foundation for further expansion, such as the integration of advanced analytics, reporting, and enhanced security features. In summary, this project illustrates how modern programming tools and techniques can transform banking systems to provide better service, improve efficiency, and secure customer data, meeting the demands of a competitive financial industry.

**2.PROBLEM STATEMENT**

#### 2.1. Background

Banking institutions have historically depended on manual or semi-automated methods to manage customer accounts, transactions, and financial records. These methods, often involving extensive paperwork and outdated systems, are ill-suited for the modern banking environment, where accuracy, speed, and accessibility are paramount. As the volume of customers and transactions increases, handling such data manually becomes cumbersome and error-prone. Furthermore, banks are required to comply with strict financial regulations and ensure data security, adding to the complexity of data management. Effective banking systems are crucial not only for operational efficiency but also for enhancing customer satisfaction and ensuring regulatory compliance.

#### 2.2. Problem Identification

Traditional methods of managing student and teacher records present Banking institutions have historically depended on manual or semi-automated methods to manage customer accounts, transactions, and financial records. These methods, often involving extensive paperwork and outdated systems, are ill-suited for the modern banking environment, where accuracy, speed, and accessibility are paramount. As the volume of customers and transactions increases, handling such data manually becomes cumbersome and error-prone. Furthermore, banks are required to comply with strict financial regulations and ensure data security, adding to the complexity of data management. Effective banking systems are crucial not only for operational efficiency but also for enhancing customer satisfaction and ensuring regulatory compliance.

2.3. Past Solutions

In the past, banking institutions adopted various methods to address data management challenges, including:

* **Manual Record-Keeping**: Banks relied on physical ledgers to record transactions and account details. While simple, this method was prone to errors and made data retrieval labor-intensive.
* **Standalone Software**: Early banking software provided limited functionalities, such as basic transaction recording and account management, without advanced features like analytics or integration.
* **Spreadsheets**: Tools like Excel allowed for a level of digital record-keeping but required significant manual input and lacked automation, real-time updates, or advanced security.
* **Custom-Built Systems**: Banks developed proprietary systems to address specific needs. However, these systems were often costly to maintain, lacked scalability, and struggled to adapt to evolving regulatory and technological demands.

These solutions addressed certain challenges but failed to provide a holistic approach to banking data management.

#### 2.4. Current Status of Bank Management Systems

Modern database management systems, such as MySQL, have revolutionized banking operations by automating data management processes. However, several challenges persist:

* **Adoption of Technology**: Many banks face difficulties in transitioning to fully digital platforms due to the complexity of system migration and user training.
* **Feature Limitations**: Existing systems often focus on specific areas (e.g., transaction processing or account management) without offering an integrated platform for comprehensive banking operations.
* **Data Security Risks**: As cyber threats evolve, ensuring the security of sensitive customer information remains a significant concern.
* **High Costs for Advanced Systems**: Full-featured banking platforms can be expensive, making them inaccessible to smaller banks or credit unions.
* **Interoperability Issues**: Many systems struggle to integrate seamlessly with third-party services like mobile payment gateways or fraud detection tools.

#### 2.5. Solution Overview

The **Bank Management System** using MySQL addresses these challenges by offering:

* **User Authentication**: Ensures secure access for authorized users through robust login mechanisms.
* **Centralized Data Management**: Consolidates customer accounts, transactions, and financial records into a unified database.
* **Search and Filter Capabilities**: Enables quick retrieval of account and transaction data, improving operational efficiency.
* **Scalability**: Handles growing datasets efficiently, making it suitable for banks of all sizes.
* **Data Integrity and Security**: Implements strong validation and encryption to prevent data breaches and ensure compliance with regulations.
* **Integration Features**: Supports connectivity with external services like online banking portals, mobile apps, and compliance monitoring systems.

This system provides a comprehensive solution for managing banking operations while enhancing accuracy, efficiency, and security

.

#### 2.6. Present Challenges and Limitations

Despite its advantages, the current system has certain limitations:

* **Single-User Access**: The system currently supports single-user login, which may limit its usability in larger banking environments with multiple departments.
* **No Cloud Integration**: Without cloud support, remote access and disaster recovery capabilities are restricted.
* **Basic Analytics**: Advanced features like customer behavior analysis, fraud detection, and financial forecasting are not yet included.
* **User Adaptability**: Staff unfamiliar with digital systems may require additional training to use the platform effectively.
* **Limited Customization**: The system may need modifications to accommodate unique banking requirements, such as region-specific regulatory compliance or specialized financial products.

Addressing these challenges will enable future iterations of the system to support multi-user environments, cloud-based operations, and advanced financial analytics, paving the way for a fully integrated and scalable banking solution.

**3.METHODOLOGY**

#### 3.1. **Object-Oriented Design**

The **Bank Management System** is built using object-oriented programming (OOP) principles, ensuring modularity, scalability, and maintainability. Key components such as Customer, Account, and TransactionManager are designed as separate classes to encapsulate their respective attributes and methods.

**Benefits:**

* **Modularity**: The system is divided into independent modules, enabling updates or replacement of components without affecting the overall functionality.
* **Reusability**: Classes and methods are reusable across different parts of the application, reducing code redundancy.
* **Scalability**: New features, such as loan management or fraud detection, can be added seamlessly.
* **Data Abstraction**: Internal details are hidden, providing users with a clean and intuitive interface.

**Drawbacks:**

* **Complexity**: Designing and implementing OOP-based systems may initially be more challenging than procedural approaches.
* **Overhead**: The creation and management of objects can increase memory and processing requirements.
* **Learning Curve**: Developers unfamiliar with OOP principles may find the system design difficult to grasp

.

#### 3.2. **Graphical User Interface (GUI)**

The GUI is developed using **Java’s Abstract Window Toolkit (AWT)**, providing a platform-independent interface for users to interact with the system. Core GUI components such as TextField, Label, and Button enable efficient data entry and navigation.

**Core GUI Components:**

* **Customer Management**: Add, view, and update customer details.
* **Transaction Dashboard**: Display account transactions and balances.
* **Search and Filter Options**: Quickly find accounts, customers, or transactions based on criteria.

**Benefits:**

* **Usability**: The user-friendly GUI ensures accessibility, even for non-technical users.
* **Platform Independence**: Java AWT supports multiple operating systems without requiring additional configurations.
* **Customizability**: The interface can be tailored to meet specific requirements, such as custom transaction filters.
* **Real-Time Interaction**: Users receive immediate feedback on their actions, such as confirmation dialogs or updated transaction views.

**Drawbacks:**

* **Performance**: AWT’s performance may lag when handling complex interfaces or large datasets compared to modern frameworks like JavaFX.
* **Limited Advanced Features**: AWT lacks the advanced UI components found in newer libraries.
* **Learning Curve**: Developers need a strong understanding of layout managers and event handling to create effective GUIs.

#### 3.3. **Event Handling and User Interaction**

The system uses event-driven programming to manage user interactions, linking GUI components like buttons and menus to event listeners.

**Event-Driven Methods:**

* **ActionListener**: Executes operations such as adding a new customer or performing a transaction when a button is clicked.
* **ItemListener**: Handles dropdown or checkbox selections, such as filtering transaction types.
* **MouseListener**: Detects user actions like double-clicking a transaction to view its details.
* **KeyListener**: Processes keyboard input for shortcuts or account searches.

**Benefits:**

* **Interactivity**: Event-driven programming creates a dynamic and responsive application.
* **Simplicity**: Predefined listeners simplify interaction handling, improving code readability.

**Example Workflow:**

* A user clicks the "Transfer" button, triggering the corresponding ActionListener, which collects input data, validates it, and processes the transaction.
* After completing the transaction, a success message or error notification is displayed to the user.

**Drawbacks:**

* **Scalability**: Managing a growing number of event listeners in a large application may complicate development and maintenance.

#### 3.4. Bank Management System Logic

The core logic of the system revolves around managing customer accounts and transactions. Algorithms handle operations like adding, retrieving, updating, and deleting records using efficient data structures such as arrays and linked lists.

**Core Algorithms:**

1. **Search Functionality:**
   * **Inputs**: User provides search terms (e.g., account number or customer name).
   * **Process**: The system performs a linear search or queries the database for matching results.
   * **Output**: Displays relevant accounts or transactions in the GUI.
   * **Efficiency**: Indexing can enhance search performance in large datasets.
2. **Sorting:**
   * **Inputs**: Users select a sorting parameter (e.g., account balance or transaction date).
   * **Process**: Implements sorting algorithms such as quicksort or mergesort.
   * **Output**: Displays sorted data in the transaction table.
   * **Scalability**: Optimized for moderate datasets but adaptable for larger ones.
3. **CRUD Operations:**
   * **Add Customer**: Appends a new customer record to the database, ensuring unique identifiers.
   * **Update Customer**: Locates a record by ID and updates the required fields.
   * **Delete Customer**: Removes a record after confirming the action.
4. **Transaction Processing:**
   * **Validation**: Ensures sufficient funds for withdrawals or transfers.
   * **Balance Update**: Adjusts account balances dynamically.
5. **File Exporting:**
   * **Outputs**: Allows exporting data such as account statements or customer lists to CSV or PDF.
   * **Process**: Iterates through datasets, formatting data into structured rows and columns.

**Benefits:**

* **Simplicity**: Well-defined algorithms ensure clarity and ease of implementation.
* **Flexibility**: Dynamic structures adapt to changes in account or transaction data.

**Drawbacks:**

* **Search Efficiency**: Linear search in linked lists can become slow for large datasets; indexing could mitigate this issue.

3.5. Data Reporting and Analytics Page

**Features:**

1. **Customer Reports:**
   * Generate summaries of customers by region or account type.
   * Highlight high-value customers based on balances or transactions.
2. **Transaction Reports:**
   * Provide summaries of daily, weekly, or monthly transactions.
   * Highlight anomalies, such as unusually large withdrawals.
3. **Account Summary:**
   * Display account activity trends, such as deposit growth or transaction frequency.
   * Highlight inactive or dormant accounts.
4. **Statistics and Insights:**
   * Total number of accounts, customers, and transactions over time.
   * Graphical trends for deposits, withdrawals, and transfers.
5. **Export Options:**
   * Allow downloading reports in PDF or CSV format for offline use.

**Benefits:**

* **Enhanced Usability**: Provides actionable insights to bank administrators.
* **Time-Saving**: Reduces manual effort through automated reporting.
* **Professional Presentation**: Reports can be used for board meetings or regulatory compliance.

**Drawbacks:**

* **Advanced Analytics Gaps**: While effective for basic reporting, advanced features like predictive analytics may require additional development.

By employing this methodology, the **Bank Management System** ensures a robust, efficient, and user-friendly platform capable of addressing the modern banking sector's operational needs.

.

**4.IMPLEMENTATION**

### IMPLEMENTATION

#### 4.1. Overview of the System Design

The **Bank Management System** is organized into three primary layers:

1. **User Interface Layer**: Built using Java AWT, it provides a simple and user-friendly interface for interacting with the system.
2. **Business Logic Layer**: Handles the core functionalities such as account management, transaction processing, and reporting.
3. **Data Layer**: Utilizes MySQL as the backend database for efficient data storage and retrieval, ensuring data integrity and scalability.

This modular design ensures flexibility, allowing updates or changes to one layer without disrupting the others.

#### 4.2. Classes and Components

The system is implemented using well-defined classes, each responsible for a specific functionality:

* **Customer**: Represents individual customers with attributes such as CustomerID, Name, Address, and Contact.
* **Account**: Manages bank account details, including AccountNumber, AccountType, Balance, and LinkedCustomerID.
* **TransactionManager**: Handles deposits, withdrawals, transfers, and transaction histories.
* **ReportGenerator**: Generates summaries and analytical reports for administrative purposes.
* **AuthenticationManager**: Ensures secure user authentication through login and sign-up functionalities.
* **DatabaseManager**: Facilitates interaction with the MySQL database, executing CRUD operations.

**Key Features:**

* **Sign-Up and Login**: Prompts users to create accounts or securely log in.
* **Customer Management**: Allows adding, updating, and deleting customer profiles.
* **Account Management**: Facilitates operations like account creation, modification, and closure.
* **Transaction Processing**: Supports deposits, withdrawals, transfers, and viewing transaction histories.
* **Analytics and Reporting**: Provides insights through visual reports and data summaries.

**Implementation of Reporting System:**  
The **ReportGenerator** interacts with the **DatabaseManager** to retrieve necessary data, process it, and present it as tables or graphs.

### 4.3. ****Detailed Functionality****

##### 4.3.1. User Authentication

Authentication ensures that only authorized users access the system.

* **Sign-Up**:
  + Captures unique usernames and passwords.
  + Validates inputs to prevent duplicate accounts.
* **Login**:
  + Verifies credentials against the database.
  + Displays error messages for incorrect username-password combinations.

**Example Workflow:**

1. A user enters a new username and password during sign-up.
2. The system checks for duplicates and saves the credentials in the MySQL database.
3. On login, the entered credentials are matched with stored values. Successful authentication grants system access.

##### 4.3.2. Customer and Account Management

* **Add Customer**:
  + Captures details like Name, Contact, and Address.
  + Validates inputs before saving them in the database.
* **Add Account**:
  + Associates an account with an existing customer.
  + Accepts parameters such as AccountType and InitialBalance.
* **Update Details**:
  + Retrieves existing records using a unique ID and allows modifications.
* **Delete Record**:
  + Removes customer or account details, ensuring referential integrity.
* **View All Records**:
  + Displays data in a paginated table with sorting and filtering options.

##### 4.3.3. Transaction Processing

* **Deposit**:
  + Updates the balance of the selected account after validating inputs.
* **Withdrawal**:
  + Checks for sufficient balance before deducting the amount.
* **Transfer**:
  + Transfers funds between two accounts after validating both accounts and balances.
* **Transaction History**:
  + Retrieves and displays all transactions for a specific account.

**Implementation Workflow:**

1. A user initiates a deposit.
2. The system validates the account details and input amount.
3. The balance is updated in the database, and a transaction record is created.

#### 4.4.Interface (UI) Design

The user interface is developed using **Java AWT**, focusing on clarity and usability.

**Key Components:**

* **Login and Sign-Up Forms**:
  + Uses TextField for input and Button for actions.
* **Dashboard**:
  + Provides options for managing customers, accounts, transactions, and reports.
* **Transaction Page**:
  + Includes input fields for amounts and dropdowns for selecting account types.
* **Reporting Page**:
  + Displays data summaries with graphical components like pie charts or bar graphs.

**Design Philosophy:**

* **Clarity**: Clear labels and tooltips guide users through various actions.
* **Consistency**: Ensures uniform layouts and themes across the application.
* **Efficiency**: Optimized workflows minimize the number of steps for common tasks.

#### 4.5.**Handling Errors**

Robust error-handling mechanisms ensure a smooth user experience and maintain system integrity.

**Examples of Errors and Solutions:**

* **Invalid Inputs**:
  + Checks for incorrect or missing data (e.g., non-numeric account numbers).
  + Displays appropriate error messages using JOptionPane.
* **Insufficient Balance**:
  + Prevents withdrawal or transfer if funds are inadequate.
  + Alerts users with a detailed message.
* **Duplicate Entries**:
  + Validates unique fields like CustomerID or AccountNumber to avoid duplicates.
* **Authentication Failures**:
  + Displays alerts for incorrect credentials or unauthorized access attempts.

**Implementation:**

* **Try-Catch Blocks**: Handle runtime exceptions such as null pointers or database connection failures.
* **Validation Methods**: Sanitize user inputs before processing.

**5.RESULTS AND DISCUSSION**

### 5.1. System Overview and Key Features

The **Bank Management System** provides a robust and efficient platform for managing customers, accounts, and financial transactions. By automating core banking operations, the system minimizes errors, enhances security, and offers improved user experiences.

**Key Features:**

* **Centralized Data Storage**: Organizes customer and account information in a secure and efficient manner.
* **Transaction Automation**: Facilitates deposits, withdrawals, and transfers with minimal user intervention.
* **Secure Access**: Implements authentication to restrict unauthorized usage.
* **Reporting and Analytics**: Offers comprehensive insights into financial data and trends.
* **Intuitive Interface**: The AWT-based GUI ensures accessibility for users of varying technical expertise.

#### 5.1.1. User Authentication and Interface

The system emphasizes both security and usability, ensuring users can access services reliably.

**Authentication Features:**

* **Sign-Up**: Ensures each user creates a unique account with strong password policies.
* **Login**: Authenticates users with accurate validation of credentials and provides informative error messages for failed attempts.

**Interface Features:**

* **User-Friendly Design**: A clean, well-organized layout ensures that users can navigate the system with ease.
* **Feedback Mechanisms**: Confirmation dialogs and error messages guide users through actions like transaction processing or account management, reducing potential mistakes.

### 5.2. Student Adding System

The transaction module is a cornerstone of the **Bank Management System**, streamlining operations like deposits, withdrawals, and transfers.

**Functionality:**

* **Input Fields**: Accepts details such as account number, transaction type, and amount.
* **Validation**: Ensures accounts exist, inputs are valid, and balances are sufficient for withdrawals or transfers.
* **Database Integration**: Records transactions dynamically, updating balances in real-time.
* **Feedback**: Provides success notifications or error alerts, depending on transaction outcomes.

**Example Scenario:**

1. A customer initiates a transfer by entering source and target account numbers and the amount.
2. The system validates the accounts and checks the source balance.
3. If validation passes, the system processes the transfer, updates balances, and notifies the customer of the success.

### 5.3. Comparison with Existing Methods

##### Traditional Banking Methods:

1. **Manual Processes**:
   * Transactions recorded in ledgers or spreadsheets.
   * Prone to human error and delays in processing.
2. **Legacy Software**:
   * Offers limited features with outdated user interfaces.
   * Often lacks robust security and remote accessibility.

##### Proposed System:

1. **Automation and Efficiency**:
   * Streamlines core banking operations, reducing manual workload.
   * Built-in validation mechanisms minimize errors.
2. **Enhanced Security**:
   * User authentication and data validation ensure secure operations.
3. **Improved Usability**:
   * A modern GUI enables seamless interaction for users of all technical levels.
4. **Scalability**:
   * Capable of handling increased transaction volumes as banks expand.

#### 5.4. Limitations and Areas for Improvement

While the system is efficient, it has some limitations that can be addressed to improve its applicability and user experience:

**Limitations:**

1. **Single-User Access**:
   * The current implementation supports only one user at a time, limiting its usability for larger banking environments.
2. **Local Storage**:
   * The absence of cloud integration restricts remote access to account and transaction data.
3. **Basic Analytics**:
   * While the system includes basic reports, it lacks advanced tools for predictive insights or trend analysis.
4. **Outdated Interface**:
   * The AWT-based GUI, though functional, lacks modern aesthetics and usability standards.

**Areas for Improvement:**

1. **Multi-User Support**:
   * Incorporate concurrent access with role-based permissions for tellers, administrators, and auditors.
2. **Cloud Integration**:
   * Enable remote database access and synchronization via cloud-based solutions.
3. **Advanced Analytics**:
   * Add features such as customer spending patterns, revenue forecasting, and fraud detection.
4. **Modern Interface**:
   * Upgrade the GUI to JavaFX or a web-based design for better usability and visual appeal.
5. **Mobile and Web Accessibility**:
   * Extend the platform to mobile apps and web interfaces to enhance user convenience.

**6.CONCLUSION**

The **Bank Management System** represents a transformative solution in the domain of financial services, addressing the challenges and inefficiencies associated with traditional banking operations. By utilizing modern software development principles, the system provides a comprehensive and efficient platform for managing customer accounts, transactions, and reporting. Its secure and intuitive design ensures a seamless experience for users, catering to both technical and non-technical stakeholders.

#### Key Contributions

1. **Operational Efficiency**  
   The system significantly streamlines core banking operations, automating tasks like deposits, withdrawals, and transfers, thereby reducing manual effort and processing time.
2. **Data Accuracy and Security**  
   Through robust validation mechanisms and user authentication, the system ensures the accuracy and confidentiality of sensitive financial data, mitigating risks associated with errors and unauthorized access.
3. **Scalability**  
   Designed to adapt to increasing transaction volumes, the system is scalable to meet the needs of growing banks and financial institutions.
4. **Enhanced Usability**  
   With its user-friendly interface, the system simplifies complex banking operations, making it accessible to users regardless of their technical expertise.

#### Achievements

The successful development of this system highlights the effective application of programming concepts such as **object-oriented design**, **event-driven programming**, and the integration of **data structures** for efficient record management. The use of Java's **AWT framework** demonstrates its capability to deliver a functional and interactive graphical interface.

#### Future Scope

While the current implementation meets its objectives, there are opportunities for further enhancement:

1. **Cloud Integration**  
   Incorporating cloud storage will enable remote access and synchronization across multiple branches, ensuring real-time data availability and updates.
2. **Multi-User Access**  
   Adding role-based permissions for tellers, administrators, and auditors will expand the system's utility for collaborative environments.
3. **Advanced Analytics**  
   Leveraging machine learning and AI can enable predictive insights, fraud detection, and deeper financial trend analyses, empowering better decision-making.
4. **Modern User Interface**  
   Upgrading to frameworks like **JavaFX** or transitioning to a web-based design will enhance the aesthetic appeal and usability of the system, making it more engaging for end-users.
5. **Mobile and Web Accessibility**  
   Extending the platform to mobile apps and web interfaces will improve accessibility, aligning with the demands of modern banking services.

#### Final Remarks

In conclusion, the **Bank Management System** is a robust, secure, and scalable platform that addresses the operational challenges of traditional banking processes. This project not only demonstrates technical proficiency but also underscores the importance of aligning software solutions with practical, real-world needs. By automating financial operations and enhancing data security, the system positions itself as a valuable tool for banks aiming to improve efficiency, customer satisfaction, and overall service delivery.

The system's foundation ensures adaptability to future advancements, making it a sustainable and impactful solution in the evolving financial industry landscape.

**7.SAMPLECODING**

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.\*;

import java.sql.\*;

import java.util.\*;

class BankSystem {

private Connection conn;

public BankSystem() { try {

conn = DriverManager.getConnection("jdbc:mysql://localhost:3307/bank\_db", "root", "21-Feb-05");

} catch (SQLException e) { e.printStackTrace(); }}

public void updateAccount(String query, Object... params) {

try (PreparedStatement stmt = conn.prepareStatement(query)) {

for (int i = 0; i < params.length; i++) stmt.setObject(i + 1, params[i]);

stmt.executeUpdate();

} catch (SQLException e) { e.printStackTrace(); }}

public Map<String, Object> getAccount(int accNo) {

try (PreparedStatement stmt = conn.prepareStatement("SELECT \* FROM bank WHERE accountNumber = ?")) {

stmt.setInt(1, accNo);

ResultSet rs = stmt.executeQuery();

if (rs.next()) return Map.of("number", rs.getInt(1), "name", rs.getString(2), "balance", rs.getDouble(3), "cibil", rs.getInt(4));

} catch (SQLException e) { e.printStackTrace(); }

return null;}

public List<Map<String, Object>> getAllAccounts() {

List<Map<String, Object>> accounts = new ArrayList<>();

try (Statement stmt = conn.createStatement(); ResultSet rs = stmt.executeQuery("SELECT \* FROM bank")) {

while (rs.next()) accounts.add(Map.of("number", rs.getInt(1), "name", rs.getString(2), "balance", rs.getDouble(3), "cibil", rs.getInt(4)));

} catch (SQLException e) { e.printStackTrace(); }

return accounts;

}}

class BankGUI extends JFrame implements ActionListener {

private BankSystem bank;

private boolean isAdmin;

public BankGUI(BankSystem bank, boolean isAdmin) {

this.bank = bank;

this.isAdmin = isAdmin;

setLayout(new FlowLayout());

String[] options = isAdmin ? new String[]{"Add Account", "View Accounts", "Delete Account"} : new String[]{"View Account", "Deposit", "Withdraw", "Check CIBIL", "Loan Eligibility"};

for (String opt : options) addButton(opt);

addButton("Back");

setTitle(isAdmin ? "Admin Panel" : "User Panel");

setSize(300, 200);

setDefaultCloseOperation(JFrame.DISPOSE\_ON\_CLOSE);

setVisible(true);

}

private void addButton(String label) {

JButton button = new JButton(label);

button.addActionListener(this);

add(button);

}

public void actionPerformed(ActionEvent e) {

String cmd = e.getActionCommand();

if (cmd.equals("Back")) dispose();

else if (isAdmin) adminActions(cmd);

else userActions(cmd);

}

private void adminActions(String cmd) {

switch (cmd) {

case "Add Account" -> addAccount();

case "View Accounts" -> viewAllAccounts();

case "Delete Account" -> deleteAccount();

}

}

private void userActions(String cmd) {

int accNo = inputInt("Enter Account Number:");

Map<String, Object> account = bank.getAccount(accNo);

if (account == null) { showMessage("Account not found!"); return; }

switch (cmd) {

case "View Account" -> showMessage(account.toString());

case "Deposit" -> deposit(account);

case "Withdraw" -> withdraw(account);

case "Check CIBIL" -> showMessage("CIBIL Score: " + account.get("cibil"));

case "Loan Eligibility" -> showMessage((int) account.get("cibil") >= 750 ? "Eligible for Loan" : "Not Eligible for Loan");

}

}

private void addAccount() {

int accNo = inputInt("Enter Account Number:");

String name = input("Enter Name:");

double balance = inputDouble("Enter Balance:");

bank.updateAccount("INSERT INTO bank VALUES (?, ?, ?, ?)", accNo, name, balance, 650);

showMessage("Account Added!");

}

private void viewAllAccounts() {

List<Map<String, Object>> accounts = bank.getAllAccounts();

StringBuilder sb = new StringBuilder();

for (Map<String, Object> acc : accounts) sb.append(acc).append("\n");

showMessage(sb.length() > 0 ? sb.toString() : "No Accounts.");

}

private void deleteAccount() {

int accNo = inputInt("Enter Account Number to Delete:");

bank.updateAccount("DELETE FROM bank WHERE accountNumber = ?", accNo);

showMessage("Account Deleted!");

}

private void deposit(Map<String, Object> account) {

double amount = inputDouble("Enter Deposit Amount:");

bank.updateAccount("UPDATE bank SET balance = balance + ?, cibilScore = cibilScore + ? WHERE accountNumber = ?",

amount, (int) (amount / 1000) \* 10, account.get("number"));

showMessage("Deposit Successful!");

}

private void withdraw(Map<String, Object> account) {

double amount = inputDouble("Enter Withdrawal Amount:");

if ((double) account.get("balance") >= amount) {

bank.updateAccount("UPDATE bank SET balance = balance - ?, cibilScore = cibilScore - 10 WHERE accountNumber = ?",

amount, account.get("number"));

showMessage("Withdrawal Successful!");

} else {

showMessage("Insufficient Funds!");

}

}

private int inputInt(String msg) { return Integer.parseInt(JOptionPane.showInputDialog(this, msg)); }

private double inputDouble(String msg) { return Double.parseDouble(JOptionPane.showInputDialog(this, msg)); }

private String input(String msg) { return JOptionPane.showInputDialog(this, msg); }

private void showMessage(String msg) { JOptionPane.showMessageDialog(this, msg); }

}

public class BankApp {

public static void main(String[] args) {

BankSystem bank = new BankSystem();

JFrame frame = new JFrame("Bank System");

frame.setLayout(new FlowLayout());

addButton(frame, "Admin Login", e -> new BankGUI(bank, true));

addButton(frame, "User Login", e -> new BankGUI(bank, false));

frame.setSize(300, 200);

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setVisible(true);

}

private static void addButton(JFrame frame, String label, ActionListener action) {

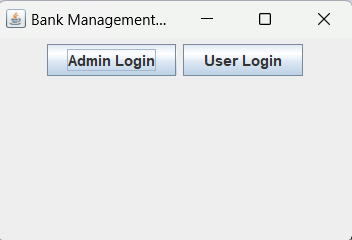
JButton button = new JButton(label);

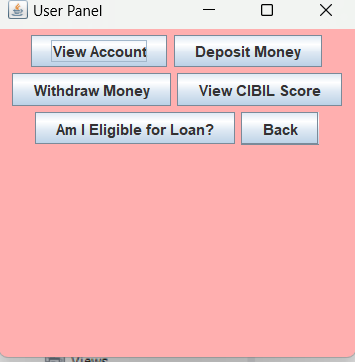
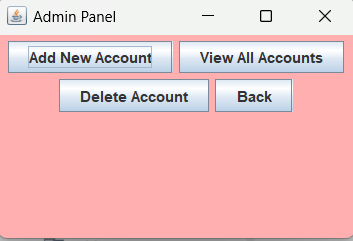
button.addActionListener(action);

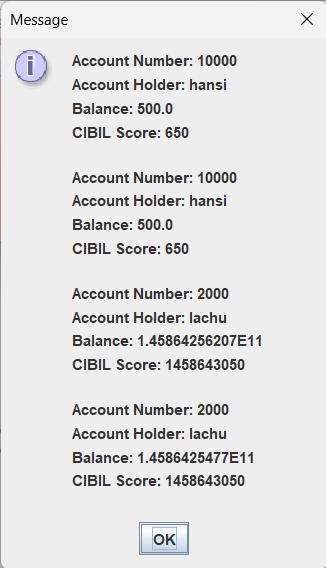
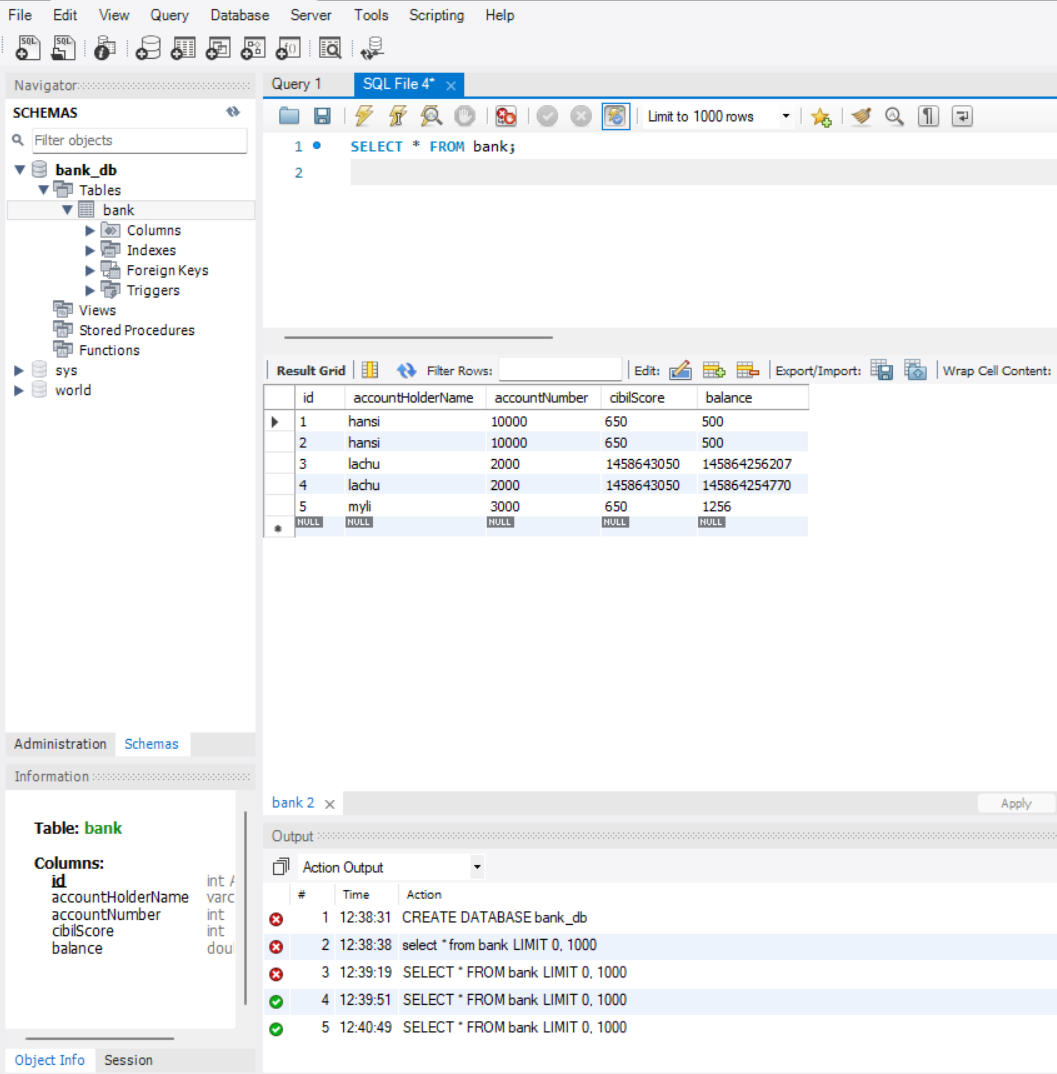
frame.add(button);

}}

**OUTPUT**

****

** **

****