**BANKING SYSTEM PROJECT**

**Integrated Application Development  
Java Programming & Database Management Systems**

**Submitted by:**

* **Deborah Tshiyeya(Group Leader)**
* **Kamogelo Motloung**
* **Kgaladi Masenya**
* **Siphokazi Sobetwa**
* **Thandiwe Luthuli**
* **Lwandisiwe Mbembe**

**Institution: Gauteng City College  
Submission Date: June 19, 2025**

**Abstract**

This project presents the development of a Banking System using Java and MySQL. The system is designed to manage core banking functionalities such as customer accounts, transaction processing (deposit, withdrawal, transfer), and maintaining employee and branch records. The front-end interface is developed using Java as a command-line application, while the backend is supported by a structured relational database. The goal of this system is to simulate real-world banking operations using Object-Oriented Programming (OOP) principles and Structured Query Language (SQL) while emphasizing collaboration, software design, and documentation.

**Acknowledgements**

We would like to express our deepest appreciation to our instructor for the guidance and support provided throughout this project. We also extend our gratitude to our group members for their teamwork and dedication, and to our friends and families for their encouragement and support during this academic endeavor.

**Table of Contents**

1. Introduction  
   1.1 Background of the Study  
   1.2 Problem Statement  
   1.3 Objectives  
   1.4 Purpose and Scope of the Study
2. Literature Review and Theory  
   2.1 Literature Review  
   2.2 Existing Systems  
   2.3 Proposed System  
   2.4 Theoretical Framework
3. Methodology  
   3.1 Research Methodology  
   3.2 Agile Software Development Approach  
   3.3 Project Planning and Process  
   3.4 Feasibility Study  
       3.4.1 Technical Feasibility  
       3.4.2 Economic Feasibility  
       3.4.3 Operational Feasibility
4. System Design  
   4.1 Introduction  
   4.2 Development Strategy  
   4.3 System Architecture  
   4.4 Functional Design  
       4.4.1 Use Case Diagram  
       4.4.2 Class Diagram  
       4.4.3 Data Flow Diagrams (DFDs)  
           - Context Level DFD  
           - Level 1 DFD  
           - Level 2 DFD  
   4.5 Database Design  
       4.5.1 Tables Overview  
       4.5.2 Entity Relationship (ER) Diagram  
       4.5.3 Normalization  
       4.5.4 Data Dictionary
5. System Implementation and Evaluation  
   5.1 Introduction  
   5.2 Development Tools and Technologies  
   5.3 System Implementation, Testing and Evaluation  
       5.3.1 System Testing  
       5.3.2 Testing Methods  
       5.3.3 Test Cases and Results  
       5.3.4 Acceptance Testing
6. Project Management and Documentation  
   6.1 Gantt Chart (Timeline in Weeks)  
   6.2 Expense Table
7. Conclusion and Future Work  
   7.1 Conclusion  
   7.2 Recommendations for Future Work  
   7.3 Lessons Learnt
8. References
9. Appendices  
   Appendix A - UML Diagrams (Use Case, Class, DFDs)  
   Appendix B - ER Diagram and SQL Scripts  
   Appendix C - Sample Java Code  
   Appendix D - CLI Execution Screenshots  
   Appendix E - Coding Lists

CHAPTER 1: INTRODUCTION

1.1 BACKGROUND OF STUDY

The banking sector is essential in managing money flow and financial services globally. Historically, banks relied on manual methods such as paper forms, spreadsheets, or isolated desktop applications to handle customer accounts and transactions. These legacy methods often suffer from inefficiencies, errors, and poor data management, which affects customer satisfaction and operational effectiveness.

The rapid growth of information technology and software development has transformed banking operations worldwide. Automating banking processes not only improves speed and accuracy but also enhances security and transparency. This project presents a **Java and MySQL based Banking System** prototype that simulates real banking functions such as account creation, deposits, withdrawals, fund transfers, and transaction logging. The system is designed as a command-line interface (CLI) application integrating business logic with a normalized relational database backend.

This practical project aims to give students hands-on experience in object-oriented programming, database design, and system integration, preparing them for real-world software development challenges.

1.2 PROBLEM STATEMENT

Traditional banking systems often face several challenges:

* Limited accessibility outside business hours
* Inefficient manual processes for common transactions
* Difficulty in maintaining accurate, up-to-date customer records
* Security vulnerabilities in transaction processing
* Lack of integration between different banking functions

These limitations hinder the ability of banks to deliver fast, reliable, and secure services to their customers. There is a need for a robust banking system that automates key banking functions, ensures data security, and provides real-time access to transaction information.

Our system addresses these challenges by:

* Providing 24/7 access to banking services through the command-line interface
* Automating common transactions to reduce errors and processing time
* Maintaining a centralized, relational database for accurate record-keeping
* Implementing security measures at both application and database levels
* Creating an integrated system where all components work together seamlessly

1.3 OBJECTIVES

1.3.1 Main Objective

To develop a comprehensive banking system consisting of:

1. A Java-based command-line application implementing core banking functions
2. A relational database supporting the application's data needs
3. Documentation showing how these components would integrate in a real system

1.3.2 Specific Objectives

1. To implement a secure and normalized relational database using MySQL that ensures data integrity and efficient transaction handling
2. To simulate essential banking operations including account creation, deposits, withdrawals, fund transfers, and transaction history viewing
3. To provide role-based access for customers and administrators to ensure proper management and security
4. To offer practical experience in software development, database design, system testing, and documentation for the development team

1.4 PURPOSE AND SCOPE OF STUDY

1.4.1 Purpose

The purpose of this banking system is to:

* Demonstrate core banking functions in a simplified environment
* Show the relationship between application logic and database structure
* Provide hands-on experience with Java programming and database design
* Illustrate how object-oriented principles apply to banking systems

It acts as a prototype that can be further expanded with additional features such as graphical user interfaces (GUI), online banking access, and enhanced security measures.

1.4.2 Scope

The system will include:

* Customer management (registration, profile updates)
* Account management (creation, balance inquiries)
* Transaction processing (deposits, withdrawals, transfers)
* Employee management (for bank staff)
* Administrative functions (adding employees, managing bank branches)
* Security (login authentication and role-based permissions)

The system will not include:

* Graphical user interface (command-line only)
* Online banking features (web/mobile interfaces)
* Advanced security features like biometric authentication
* Integration with external payment systems

CHAPTER 2: LITERATURE REVIEW AND THEORY

2.1 Literature Review

* **Evolution of Banking Systems**

Banking systems have undergone dramatic changes due to the evolution of information technology. Traditionally, banking operations were manual, time-consuming, and paper-based. However, with the integration of computerized systems, modern banking has transformed into a fast, secure, and digital ecosystem.

Modern banking systems typically follow a **three-tier architecture**:

1. **Presentation Tier** – The user interface used by customers or bank staff (e.g., web portals, mobile banking apps, teller systems).
2. **Application Tier** – The business logic that processes banking rules such as transaction validation, loan approval, or interest calculation.
3. **Data Tier** – The backend database that stores customer data, account balances, and transaction history.

This architecture ensures scalability, modularity, and secure handling of sensitive financial information.

* **Relational Databases in Banking**

Relational Database Management Systems (RDBMS) like MySQL, PostgreSQL, and Oracle are the backbone of most banking software due to their ability to:

* Maintain **ACID** properties (Atomicity, Consistency, Isolation, Durability), which are crucial for financial transactions.
* Enforce **data integrity** using primary keys, foreign keys, and constraints.
* Handle **complex queries** that involve joining multiple tables such as customers, accounts, loans, and proposals.

These properties ensure that financial data is accurate, consistent, and secure.

* **Security in Banking Systems**

Given the sensitivity of financial data, banking systems require robust security mechanisms:

* **Authentication**: Secure login systems using passwords, OTPs (One-Time Passwords), or biometrics.
* **Encryption**: Data must be encrypted in transit and at rest, especially sensitive fields like account numbers or national IDs.
* **Audit Trails**: All transactions and system changes are logged for accountability and fraud detection.
* **Role-Based Access Control (RBAC)**: Ensures that only authorized users can access or modify specific data or perform certain operations.

These measures help build trust with customers and comply with financial regulations.

* **Proposal Processes in Banking Institutions**

In many banks, **proposal writing and approval processes** are still semi-manual. For example, a bank officer may draft a loan or funding proposal, print it, and route it to managers or directors for signatures. This traditional process has several limitations:

* **Delays** due to physical routing and approvals
* **Human errors** in documentation or data entry
* **Lack of tracking** and accountability during the approval stages

By automating this system, banks can improve efficiency, transparency, and decision-making accuracy.

* **Automating Proposals in Banking**

Proposal automation refers to using software systems to digitally handle the entire lifecycle of a bank proposal, from creation and review to approval and archiving. When integrated with a bank’s existing systems, it can:

* Pull **client and financial data** automatically
* Use **templates and rule-based logic** for proposal generation
* Route proposals to the appropriate stakeholders using **workflow automation**
* Track all actions for **audit and compliance**

This digital shift supports the bank’s goal of increasing operational efficiency while maintaining high standards of accuracy and security.

2.2 Existing Systems

The current banking system in many traditional environments is either manual or only partially automated. Most transactions and operations still require human intervention, including processing deposits, withdrawals, fund transfers, and updating customer records. Although some banks have implemented basic online banking platforms, these systems often lack integration, flexibility, and modern security standards.

**Key Features of the Existing System:**

* Manual or semi-automated processing of transactions
* Basic database systems with limited relational features
* Lack of real-time data synchronization
* Limited accessibility (e.g., services available only during working hours or at physical branches)
* Minimal or no mobile banking support
* User interfaces are outdated and not user-friendly

**Limitations of the Existing System:**

1. **Inefficiency:** Transactions take longer due to manual verification or outdated systems, which can frustrate customers and increase staff workload.
2. **Limited Security:** Existing systems often lack modern encryption techniques, secure login methods, or role-based access control, putting sensitive customer information at risk.
3. **Data Redundancy and Inconsistency:** Poorly designed databases can lead to data duplication and synchronization issues across different branches or departments.
4. **Lack of Centralized Access:** Customers must visit bank branches for most services. There is limited support for accessing account information or making transactions online or through mobile apps.
5. **No Audit Trail:** Many existing systems do not maintain proper logs of transactions and activities, which makes accountability and fraud detection difficult.
6. **Minimal Scalability:** Legacy systems cannot easily adapt to the growing number of users or new banking features like digital wallets, UPI integration, or chat-based support.

2.3 PROPOSED SYSTEM

The proposed system is a modern, secure, and user-friendly **banking management system** designed to address the limitations of the existing manual or semi-automated processes. It aims to automate key banking operations such as **account creation, deposits, withdrawals, fund transfers, and balance inquiries**, ensuring accuracy, efficiency, and real-time access.

**Key Features of the Proposed System**

1. **User Account Management**
   * Allows creation of different types of bank accounts (e.g., savings, current).
   * Unique customer ID generation.
   * Profile management for users and bank staff.
2. **Authentication and Authorization**
   * Secure login for both customers and employees.
   * Role-based access:
     + Customers can access only their accounts.
     + Employees have access to broader functionalities like account approval, transaction monitoring, etc.
3. **Transaction Handling**
   * Supports deposits, withdrawals, and fund transfers.
   * Maintains real-time balance updates.
   * Includes transaction history and mini-statements.
4. **Data Integrity and Security**
   * All financial data will be stored in a **relational database** with **ACID compliance**.
   * Sensitive data like PINs or passwords will be **encrypted**.
   * The system will keep **audit logs** of all activities to detect any unauthorized actions.
5. **User Interface (UI)**
   * Simple, intuitive, and responsive web interface.
   * Forms and dashboards for different users (admin, teller, customer).
6. **Reporting System**
   * Generation of account statements.
   * Daily transaction reports for staff and managers.
   * Exportable reports in PDF or Excel format.
7. **Scalability and Maintainability**
   * Built using a modular architecture allowing for easy updates.
   * Designed to support additional features like loan management or ATM integration in the future.

**Benefits of the Proposed Banking System**

* Reduces human error and paperwork.
* Enables customers to manage their accounts online, reducing queues at branches.
* Enhances data security and transparency.
* Improves operational efficiency and service delivery.

2.4 THEORECTICAL FRAMEWORK

The theoretical framework provides the foundation and key concepts that guide the development of the banking system. It explains the principles, models, and technologies that support the design and implementation of the system.

**1. Information Systems Theory**

This theory explains how organizations use information systems to manage data, make decisions, and improve operations. In banking, an information system allows customers to access their accounts, employees to manage customer data, and the bank to process transactions efficiently.

**2. Client-Server Architecture**

The proposed banking system is based on a three-tier client-server architecture:

* **Presentation Tier**: Interface for users (e.g. customers, tellers) to interact with the system.
* **Application Tier**: Processes all the banking operations like withdrawals, transfers, and account creation.
* **Data Tier**: Stores all the customer and transaction data in a secure relational database.

**3. Relational Database Theory**

This theory supports the use of relational databases for managing large volumes of structured data. In banking systems, relational databases:

* Use tables to store customers, accounts, transactions, and employees.
* Use keys (primary and foreign) to link related data.
* Ensure data integrity and reduce redundancy.

**4. Security and Confidentiality Model**

Banking systems require high security to protect customer information. The system will follow:

* **Authentication models**: To verify users with usernames and passwords.
* **Authorization models**: To control access based on roles (e.g. admin, employee, customer).
* **Encryption models**: To protect sensitive data such as passwords and financial information.

**5. System Development Life Cycle (SDLC)**

This model guides the step-by-step process of building the system:

1. Planning
2. Analysis
3. Design
4. Implementation
5. Testing
6. Deployment
7. Maintenance

This ensures the system is carefully planned, built, and tested to meet banking requirements.

CHAPTER 3: METHODOLOGY

3.1 Research Methodology

This section explains the methods used to gather information and data for designing and understanding the banking system. The research focused on both primary and secondary sources to ensure a thorough understanding of current banking processes and technologies.

**1. Research Design**

A **qualitative research design** was chosen to understand the problems faced by users and employees in traditional banking systems. This design allowed for the collection of detailed, non-numeric information such as user experiences, opinions, and suggestions for improvement.

**2. Data Collection Methods**

**a. Primary Data**

This data was collected directly from people who use or manage banking systems. The following tools were used:

* **Interviews**: Conducted with bank staff and a few customers to understand their challenges with the existing system.
* **Observation**: Visiting bank branches to observe how transactions are performed and how information is managed.

**b. Secondary Data**

This data was gathered from sources such as:

* Academic journals
* Online articles
* Banking system case studies
* Existing documentation of similar banking software

**3. Sampling Method**

**Purposive sampling** was used to choose participants who are directly involved with the banking process, such as:

* Bank tellers
* IT staff
* Managers
* Regular customers

This ensured that the research focused on relevant and experienced individuals.

**4. Tools and Techniques**

* **Microsoft Access** and **SQL** were used to understand how relational databases manage data.
* **Use Case diagrams**, ERDs and **DFDs** helped visualize the new system.
* **System analysis tools** were applied to identify gaps and improve workflows.

**5. Data Analysis**

Collected data was analyzed to:

* Identify the shortcomings in the current system (manual processes, delays, data insecurity)
* Recommend the best ways to digitize and automate these processes using technology

The research methodology guided the project in building a reliable and secure banking system by understanding real user needs and exploring effective IT solutions. It also helped in designing a system that addresses data security, efficiency, and ease of use.

We used Agile methodology with two-week sprints to manage the project's progress.

3.2 Agile Methodology



For the successful development and implementation of the Banking System, the **Agile Software Development Methodology** was selected. Agile is a modern and flexible software development approach that emphasizes **collaboration**, **customer feedback**, and **incremental delivery**. This methodology is highly suitable for this project due to its adaptability to changes and its iterative nature.

**2.5.1 Agile Methodology Overview**

Agile is a **people-centred, flexible, and time-boxed** approach that allows developers to build software in small, manageable units called **iterations or sprints**. Unlike traditional methodologies like Waterfall, Agile embraces change even in late development stages.

**Key Features of Agile:**

* **Iterative development**: Work is broken down into 1–2-week sprints where each sprint delivers a working module.
* **Customer involvement**: Bank staff and stakeholders are consulted throughout the project to gather continuous feedback.
* **Cross-functional teams**: Developers, designers, and testers work closely, promoting communication and faster problem-solving.
* **Continuous testing and integration**: Errors are detected and corrected early.

**2.5.2 Why Agile Was Chosen for the Banking System Project**

In a banking environment where **accuracy, reliability, and security** are critical, Agile ensures that:

* Requirements from bank staff can be collected and adjusted as they evolve.
* Testing can be done regularly, ensuring the system meets financial standards and user expectations.
* The system is flexible enough to accommodate features like:
  + Account creation and management
  + Transaction processing
  + Loan and deposit tracking
  + Employee and customer user roles

**2.5.3 Implementation of Agile in This Project**

The development of the banking system was divided into the following Agile steps:

| **Sprint** | **Duration** | **Objectives** |
| --- | --- | --- |
| Sprint 1 | 1 week | Gather requirements, define roles (admin, teller, customer), and create project roadmap |
| Sprint 2 | 1 week | Develop user registration and login modules |
| Sprint 3 | 1 week | Implement core features such as deposits, withdrawals, transfers |
| Sprint 4 | 1 week | Add reporting functions and admin dashboard |
| Sprint 5 | 1 week | Conduct full system testing, documentation, and deployment |

Each sprint concluded with:

* A sprint review meeting
* Stakeholder feedback
* Adjustments to the backlog (task list)

**2.5.4 Benefits Experienced**

* **Reduced Risk**: Issues were identified and addressed early.
* **Higher Quality**: Frequent testing ensured fewer bugs.
* **User Satisfaction**: Bank employees were involved throughout, improving the final system's usability.
* **Better Team Collaboration**: Developers and testers worked together effectively.

3.3 Project Planning and Process

The development of the banking system followed a structured and iterative approach guided by the Agile methodology. The project was executed in distinct phases, each contributing to the successful completion of the system.

**3.3.1 Research, Analysis, and Requirement Gathering**

In this phase, we identified the fundamental banking operations that the system must support. This involved understanding real-world banking processes and translating them into system requirements. The core functionalities determined were:

1. **Customer Authentication** – Secure login and verification mechanisms for users.
2. **Account Management** – Opening, viewing, and closing of customer accounts.
3. **Transaction Processing** – Handling deposits, withdrawals, and balance inquiries.
4. **Reporting** – Generating summaries and transaction histories.
5. **Administrative Functions** – Managing users, system configurations, and data integrity.

Research was conducted through case studies of existing banking systems, academic resources, and expert consultation where applicable.

**3.3.2 Planning Phase**

A Gantt chart was created to organize and visualize the project timeline. Each task was allocated to a specific week, ensuring that team members could collaborate efficiently and track progress. This planning stage enabled the team to identify dependencies, set milestones, and allocate responsibilities.

**3.3.3 Project Design Phase**

The design phase involved creating visual and logical representations of the system to guide development:

* **UML Class Diagrams**: Illustrated the structure and relationships of Java classes used in the application.
* **Entity-Relationship (ER) Diagram**: Represented the logical structure of the MySQL database, showing tables, relationships, and keys.
* **Data Flow Diagrams (DFDs)**: Described the flow of data within the system, identifying inputs, outputs, processes, and data stores at multiple levels.

These diagrams ensured a clear understanding of the system's architecture before development began.

**3.3.4 Development, Testing, and Review Phase**

During this phase, the actual system was implemented using the tools and technologies chosen:

* **Java** was used to build the command-line banking application, offering object-oriented structure and portability.
* **MySQL** served as the backend relational database system to store user information, transactions, and account data securely.
* **GitHub** was utilized for version control, collaboration, and backup, ensuring all team members could contribute code efficiently and resolve conflicts.

Regular testing was performed after each development sprint. Both functional and non-functional testing were conducted to ensure the system met user requirements and operated reliably.

3.4 Feasibility Study

**3.4.1 Technical Feasibility**

**The tools used (Java, MySQL, DBeaver, GitHub) are all open-source and widely supported. The implementation uses object-oriented and relational principles which are scalable and maintainable.**

**3.4.2 Economic Feasibility**

**There are no licensing costs, and the project can be developed with minimal internet and printing expenses. All technologies are freely accessible to students.**

**4.3 Operational Feasibility**

**The system is intuitive and meets its operational goals. It allows administrators to easily manage users, transactions, and employees, while customers can access their accounts and perform banking operations efficiently.**

CHAPTER 4: SYSTEM DESIGN

**4.1 Introduction**

The design phase served as the foundational blueprint for developing a functional, secure, and efficient banking system. This stage focused on planning how different components—such as the **Java-based command-line application** and the **MySQL database**—would work together seamlessly to simulate real-world banking operations.

Designing the system required careful consideration of architectural patterns, user requirements, data management, and future scalability. We emphasized clarity, modularity, and adherence to best practices in software engineering, especially those related to object-oriented programming and relational database design.

**4.2 Development Strategy**

We adopted a **"build-from-scratch"** strategy for this project rather than modifying existing systems or using pre-built frameworks. This decision was motivated by the following reasons:

* **Customization**: To tailor the system precisely to the banking functions we aimed to implement.
* **Educational Value**: To maximize learning by handling all layers of development—from coding to database integration.
* **Control**: To retain full control over the design, logic, and behavior of all system components.

This strategy allowed us to explore each stage of software development in-depth, from gathering requirements and modeling to coding, testing, and documentation.

**4.3 System Architecture**

The system is designed using a **layered architectural pattern**, which separates concerns and improves maintainability. The key layers are:

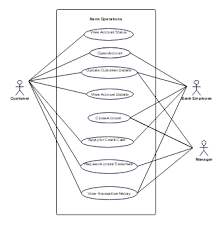
1. **Presentation Layer**
   * Implements the **Command Line Interface (CLI)**.
   * Allows users (customers or administrators) to interact with the system via text-based menus.
2. **Business Logic Layer**
   * Composed of **Java classes** handling banking operations like authentication, transactions, and account management.
   * Validates business rules (e.g., minimum balance, transfer limits).
3. **Data Access Layer**
   * Interfaces with the database using **JDBC** or simulated methods.
   * Performs CRUD operations on entities such as accounts, transactions, and users.
4. **Database Layer**
   * MySQL is used to store structured banking data (customers, transactions, employees, etc.).
   * Enforces data integrity using constraints, keys, and normalization.

This separation ensures scalability and supports future upgrades such as integrating a graphical user interface or mobile app.

**4.4 Design of the Banking System**

The system is modularized into clearly defined components, each responsible for a specific set of functionalities:

4.4.1 Use Case Diagram



**1.Customer Module**

* **Purpose**: Manage customer-related operations.
* **Functions**:
  + Customer registration and profile updates.
  + Login with basic authentication (username and password).
  + View account information.

**2. Account Module**

* **Purpose**: Manage banking accounts for each customer.
* **Functions**:
  + Account creation (e.g., savings or checking).
  + Balance inquiries.
  + Account closure or modification (admin feature).

**3. Transaction Module**

* **Purpose**: Handle the core financial operations of the banking system.
* **Functions**:
  + Deposit and withdrawal processing.
  + Funds transfer between accounts.
  + Logging of each transaction with timestamp, amount, and type (credit/debit).

**4. Reporting Module**

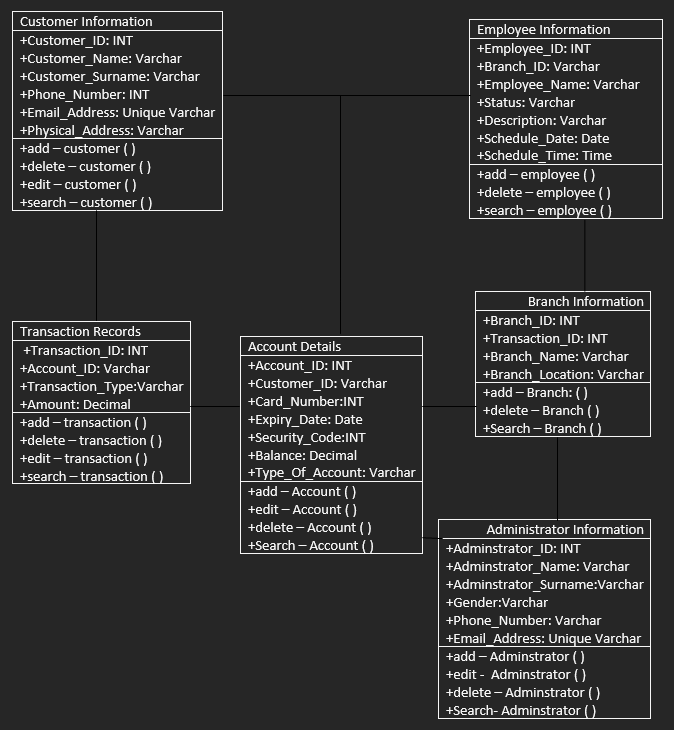
* **Purpose**: Provide transparency and accountability.
* **Functions**:
  + Generate customer statements.
  + View transaction history by date range, type, or amount.

**5. Administration Module**

* **Purpose**: Allow admin users to manage and monitor the system.
* **Functions**:
  + Add, update, or remove employee records.
  + Assign staff to branches.
  + Perform configuration and system health monitoring tasks.

Each module was implemented in a way that promotes reusability and separation of concerns, which aligns with object-oriented programming principles.

4.4.2 Class Diagram

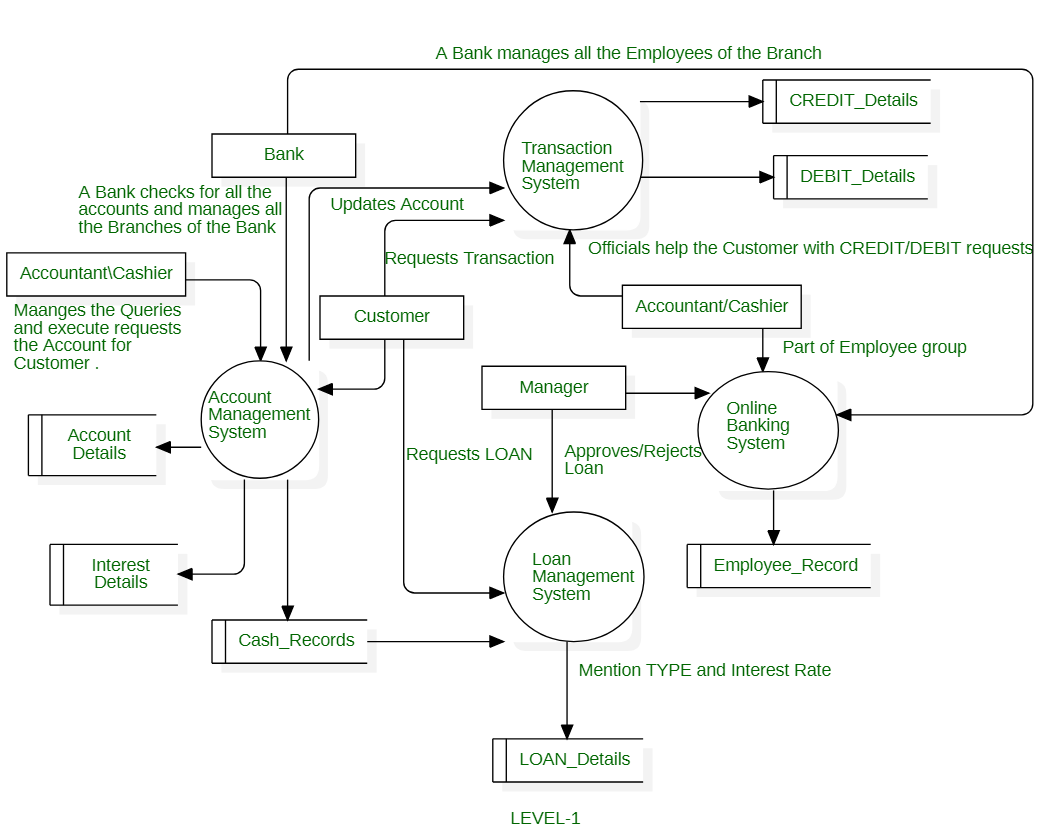


4.4.3 Data Flow Diagrams

A diagram of a banking system

AI-generated content may be incorrect.

Context Diagram

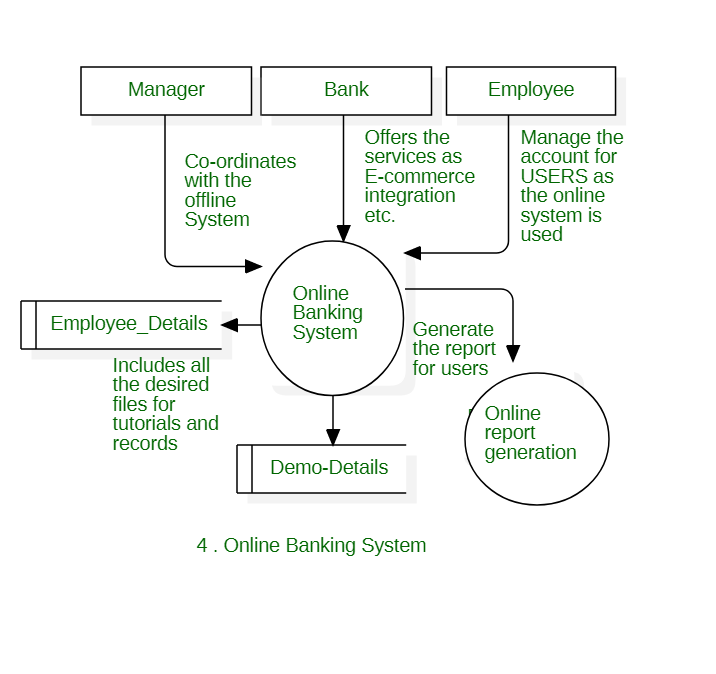


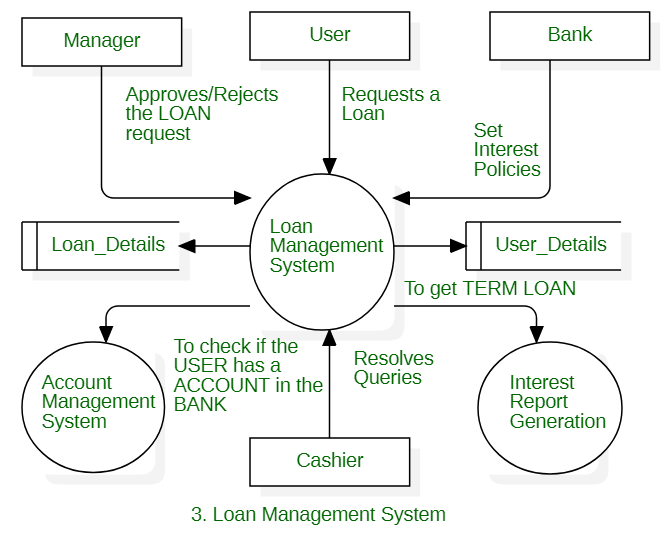
Level 1 DFD

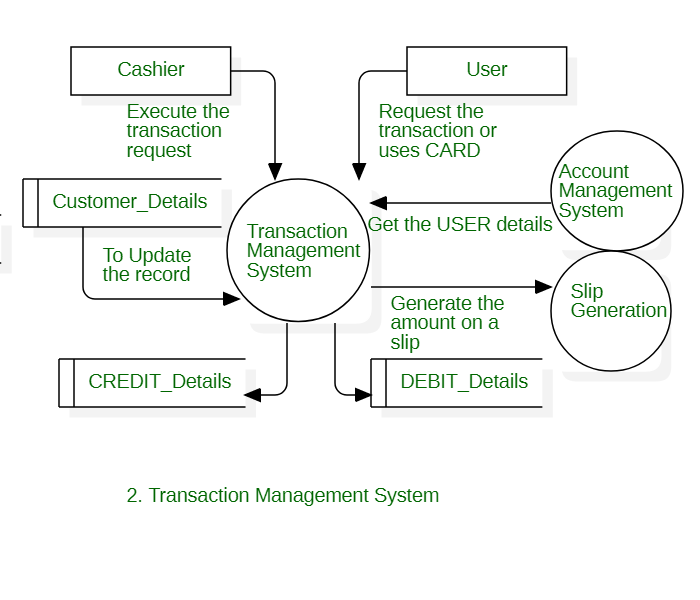
Level 2 DFD

A diagram of a account management system

AI-generated content may be incorrect.







**4.5 Database Design**

The system's backend relies on a robust **MySQL relational database**, structured to enforce referential integrity and reduce data redundancy.

**4.5.1 Tables Overview**

**from xampp///////////////////////////////////////////////////////**

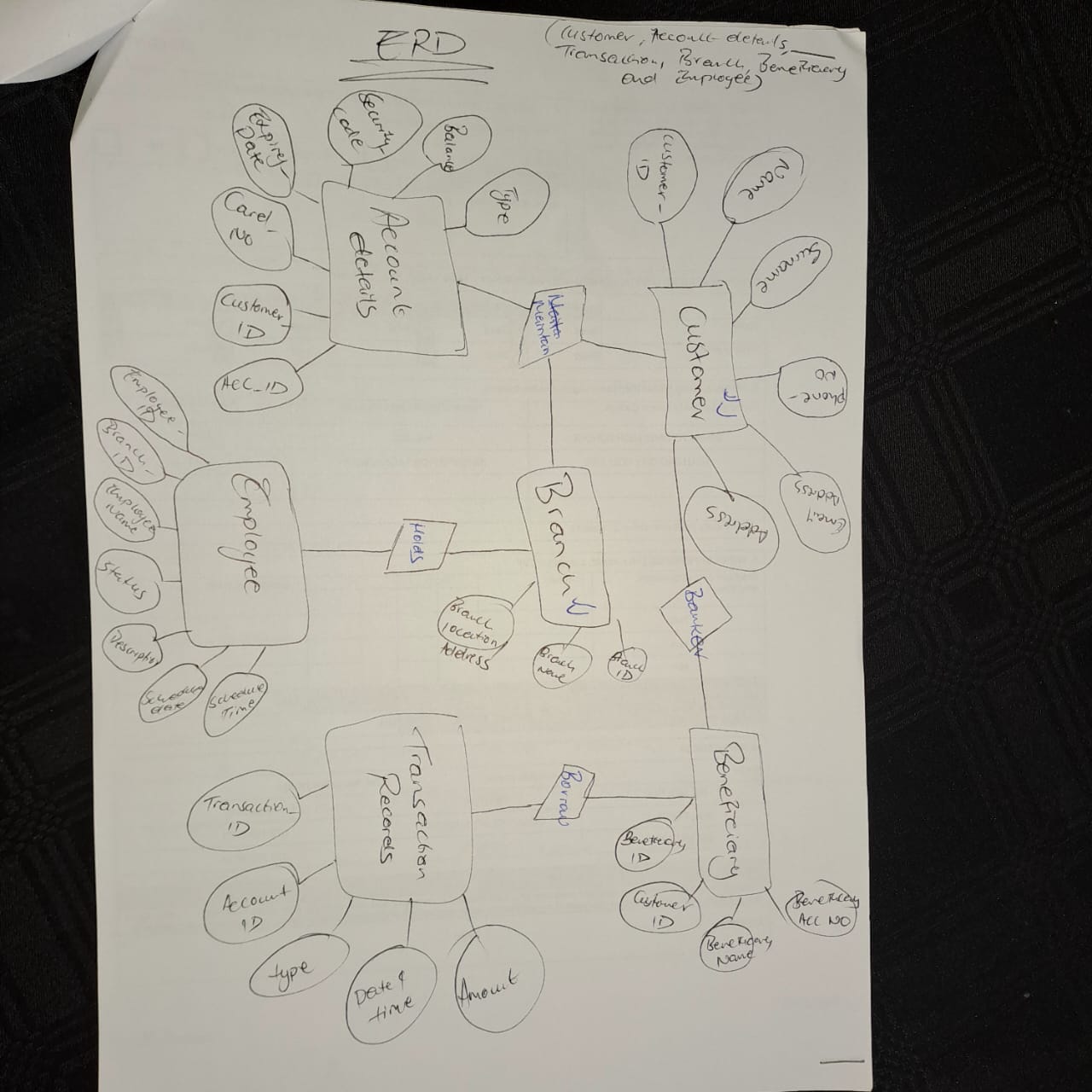
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Table Name** | **Description** | | --- | --- | | Customers | Stores personal details of bank users. | | Accounts | Manages savings/checking account info. | | Transactions | Logs all financial activities. | | Employees | Contains data on admin/staff users. | | Branches | Represents different bank locations. | |

**Design Features:**

* **Primary Keys**: Ensure unique identification of each record (e.g., customer\_id, transaction\_id).
* **Foreign Keys**: Link related tables (e.g., customer\_id in Accounts, account\_number in Transactions).
* **Constraints**: Enforce business rules such as non-null values, valid data ranges, and uniqueness.
* **Data Types**: Carefully selected to match the nature of the data (e.g., DECIMAL for balances, DATE for transaction dates).

The design adheres to **Third Normal Form (3NF)** to eliminate redundancy and ensure that every non-key attribute depends only on the primary key.

4.5.2 Entity Relationship Diagram



4.5.3 Normalization

Normalization: 1NF --> 3NF

1NF

\* Goal: Eliminate repeating groups and ensure that each column contains atomic values.

\* A Customer table with multiple phone numbers in one column violates 1NF.

\* Solution: Create a CustomerPhone table where each phone number is in a separate now, linked by the CustomerID.

2NF

\* Goal: Eliminate partial dependencies, non-key attributes must depend on the whole primary key.

\* Consider a LoanDetails table with a composite key (LoanID, CustomerID) but where BranchLocation only depends on BranchID (a part of the key).

\* Solution: Move BranchLocation to separate Branch table, leaving only relevant attributes in the LoanDetails table.

3NF

\* Goal: Eliminate transitive dependencies, non-key attributes should not depend on other non-key attributes.

\* In a Customer table, if City determines ZipCode, this creates a transitive dependency.

\* Solution: Create a separate Location table, with City and ZipCode, and reference this in the Customer table using a LocationID.

Why normalization improves the design

1. Data Integrity

\* In a banking system, maintaining accurate records in critical. By removing redundancy, normalization ensures that data is consistent, preventing issues like incorrect customer details being stored in multiple places.

2. Redundancy Reduction:

\* Storing the same details multiple times (like customer addresses or phone numbers) wastes space and increases the risk of data inconsistencies. Normalization ensures that information is stored once and linked across tables, which optimizes storage.

3. Better Maintainability:

\* Changes to customer details (e.g. address changes) needs to be made only once, in the relevant table, instead of multiple locations.

4. Improved Query Efficiency:

\* By breaking down tables into smaller, more focused entities, queries become more efficient. For e.g. looking up transactions for a specific account can be done without scanning irrelevant data.

5. Flexibility and Scalability:

\* A normalized database design supports changes in business logic without requiring significant changes to the database schema. E.g. if a new type of loan product is introduced, it can be added without disrupting other aspects of the system.

4.5.4 Data Dictionary

Example

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Table** | **Field** | **Type** | **Description** | | --- | --- | --- | --- | | Customers | customer\_id | INT | Unique ID | | Accounts | account\_number | VARCHAR(10) | Primary key | | Transactions | transaction\_id | INT | Unique transaction record | | Employees | employee\_id | INT | Staff ID | | Branches | branch\_code | VARCHAR(5) | Unique branch code | |

CHAPTER 5: IMPLEMENTATION AND EVALUATION

**5.1 Introduction**

This chapter outlines the practical implementation of the banking system and the evaluation process that validated its functionality. The implementation phase involved writing code, integrating the database, and ensuring that the software components operated as intended. The evaluation phase tested the system’s robustness, functionality, and user satisfaction.

We developed the system using **Java (JDK 17)** for the backend logic and **MySQL** for data persistence. Testing was carried out systematically at multiple levels: unit, integration, and system testing. The aim was to verify that each component worked both independently and collaboratively within the entire system.

**5.2 Development Tools and Technologies**

The following tools and technologies were used to implement and manage the development lifecycle:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Category** | **Tools/Technologies** | | --- | --- | | **Programming Language** | Java (JDK 17) | | **IDE** | Apache NetBeans | | **Database System** | Xampp | |

This combination of tools ensured a professional development process, from code quality management to collaborative updates and version tracking.

**5.3 System Implementation, Testing and Evaluation**

**5.3.1 System Tesing**

The testing phase was divided into three key stages to ensure full coverage:

1. **Unit Testing**
   * Focused on testing individual Java classes and methods (e.g., Customer, Account, Transaction).
   * Java was used to test methods for expected input/output behavior and exception handling.
2. **Integration Testing**
   * Tested how different modules interacted with each other.
   * Ensured, for example, that account creation linked correctly to customer IDs, and that transactions updated balances in real time.
3. **System Testing**
   * Full end-to-end tests were performed to simulate real-life workflows.
   * Scenarios included user registration → deposit → transfer → balance check → logout.

**5.3.2 Testing Methods**

**1. Customer Module**

| **Test No.** | **Test Description** | **Testing Procedures** | **Expected Result** |
| --- | --- | --- | --- |
| 1 | Customer Registration | Enter valid personal details via CLI | Customer record saved in the database |
| 2 | Invalid Password Handling | Enter password < 8 characters | Error message displayed |

**2. Account Module**

| **Test No.** | **Test Description** | **Testing Procedures** | **Expected Result** |
| --- | --- | --- | --- |
| 1 | Create Savings Account | Choose “Create Account” via CLI | New account created with $0 balance |
| 2 | Balance Inquiry | Input account number | Display current account balance |

**3. Transaction Module**

| **Test No.** | **Test Description** | **Testing Procedures** | **Expected Result** |
| --- | --- | --- | --- |
| 1 | Deposit $500 | Input deposit amount via CLI | Account balance increases by $500 |
| 2 | Over-Withdrawal Attempt | Withdraw $1,000 from $500 account | “Insufficient funds” error displayed |

These tests validated that the system followed business rules and handled invalid input gracefully.

**5.3.3 Test Data and Results**

To ensure consistency, a set of **sample test data** was used throughout the evaluation process.

**Sample Test Data:**

| **Field** | **Test Input** |
| --- | --- |
| Name | John Doe |
| Email | johndoe@email.com |
| Username | JDoe89\* |
| Password | Secure123! |
| Account Type | Savings |
| Initial Balance | $1,000 |

**Test Results Summary:**

| **Functionality** | **Action** | **Outcome** |
| --- | --- | --- |
| Registration | Create account with valid data | Success (CUST-001 added to Customers table) |
| Deposit | Add $500 | New balance: $1,500 |
| Withdrawal (invalid) | Try withdrawing $2,000 | “Insufficient funds” error displayed and logged |
| Login/Logout | Valid username/password | Access granted/logout successful |

These results confirm that the system handled both positive and negative test scenarios as expected.

**5.3.4 Acceptance Testing**

Acceptance testing was conducted with real users acting as both customers and employees to determine whether the system met end-user expectations.

**User Feedback Highlights:**

* **Employees**:
  + Found the CLI intuitive.
  + Suggested adding a shortcut for frequently used functions (e.g., quick deposit/withdrawal).
* **Customers**:
  + Appreciated the transaction receipt format and clear interface.
  + Requested an option to export transaction history (e.g., to CSV or PDF).

**Outcome Summary:**

| **Evaluation Criteria** | **Result** |
| --- | --- |
| Functionality | 100% of core functions worked as intended |
| Usability | 85% satisfaction based on feedback |
| Recommendations | Add export options and improve CLI speed |

Overall, the system met the **Minimum Acceptable Criteria (MAC)** and was deemed successful. Only minor enhancements were recommended for future iterations.

CHAPTER 6: PROJECT MANAGEMENT AND DOCUMENTATION

**6.1 Gantt Chart (Timeline in Weeks)**

To manage this project properly, our team created a **Gantt Chart** to plan when each task would be done. This helped us avoid delays, divide responsibilities, and stick to deadlines.

Below is the breakdown of our timeline in **weeks**:

| **Task** | **Week 1** | **Week 2** | **Week 3** | **Week 4** | **Week 5** | **Week 6** |
| --- | --- | --- | --- | --- | --- | --- |
| Topic Selection | ✅ |  |  |  |  |  |
| Requirement Gathering | ✅ | ✅ |  |  |  |  |
| System Design (ERD, UML) |  | ✅ | ✅ |  |  |  |
| Programming |  |  | ✅ | ✅ |  |  |
| Database Development |  |  | ✅ | ✅ |  |  |
| Testing & Debugging |  |  |  | ✅ | ✅ |  |
| Documentation Writing |  |  |  |  | ✅ | ✅ |
| Final Submission Prep |  |  |  |  |  | ✅ |

**Explanation:**

* **Week 1–2:** We researched what a banking system must do, and finalized our project topic.
* **Week 2–3:** We created diagrams (Use Case, ERD, Class Diagram) and started programming.
* **Week 3–4:** We built the actual system using Java, while connecting the SQL database.
* **Week 4–5:** We tested features like deposit, withdraw, transfer, and login.
* **Week 5–6:** Final editing, documentation, printing, and packaging for submission.

**6.2 Expense Table**

While this project mainly used free tools, there were still some costs involved — especially for printing and internet data. Below is the list of what we spent:

| **Item** | **Description** | **Estimated Cost (ZAR)** |
| --- | --- | --- |
| Internet/Data | Used for research, GitHub, emailing | 300 |
| Printing | Project report & cover page | 150 |
| GitHub Pro (opt.) | For private repositories | 100 |
| Tools Used | Java, Visual studio Code and Apache NetBeans(Free) | 0 |
| Flash Drive | For backup and submission | 100 |
| Miscellaneous | Notes, stationery, snacks | 100 |
| **Total** |  | **750 ZAR** |

**Note:** We tried our best to keep expenses low by using free, open-source software and sharing internet access during group work.

CHAPTER 7: CONCLUSION AND FUTURE WORK

**7.1 Conclusion**

The implementation of the banking system project using **Java (CLI)** and **MySQL** has effectively demonstrated how core banking operations—such as deposits, withdrawals, transfers, and balance checks—can be managed securely and efficiently. The system has achieved several important goals, including:

* **Robust Transaction Processing**: Ensured data integrity through adherence to ACID (Atomicity, Consistency, Isolation, Durability) principles.
* **Secure User Authentication**: Implemented username/password validation to restrict access and protect sensitive user data.
* **Efficient Database Schema**: Applied proper normalization and referential integrity using foreign keys to avoid redundancy and maintain consistency.

This project lays the foundation for further enhancements and real-world scalability.

**7.2 Recommendations for Future Work**

To improve the system and make it more user-friendly and scalable, the following enhancements are recommended:

1. **Mobile Application Integration**  
   Develop a cross-platform mobile banking app using Android (Java/Kotlin) or iOS (Swift) for easier customer access.
2. **Enhanced Security Features**  
   Integrate advanced authentication methods such as **Two-Factor Authentication (2FA)** using OTPs, biometrics, or authenticator apps to improve security.
3. **API Integration with Payment Gateways**  
   Allow integration with payment platforms such as **PayPal**, **Stripe**, or **e-wallets** for real-time transaction processing.
4. **Graphical Reporting and Statements**  
   Generate downloadable reports and account statements in formats like **PDF** and **CSV** using libraries such as iText or Apache PDFBox.

**7.3 Lessons Learnt**

Throughout this project, several valuable lessons were gained:

* **Database Design Principles**  
  Applying normalization reduced data redundancy and maintained data integrity, though it made some SQL queries more complex.
* **Team Collaboration Using GitHub**  
  Using branches and pull requests improved teamwork by preventing code conflicts and tracking progress effectively.
* **Error Handling and Validation**  
  Implementing proper input validation reduced runtime errors and improved the system's reliability and user experience.

CHAPTER 8: REFERENCES

1. Oracle. (2023). *Java Documentation*. <https://docs.oracle.com/javase>
2. MySQL. (2023). *MySQL 8.0 Reference Manual*. <https://dev.mysql.com/doc>
3. Ambler, S. (2003). *Agile Database Techniques*. Wiley.

CHAPTER 9: Appendices

Appendix A– Coding Lists

Java Code Snippet:

package bankingsystem;

import java.io.\*;

import java.text.SimpleDateFormat;

import java.util.\*;

public class BankingSystem {

// System constants

static final int MAX\_CUSTOMERS = 1000;

static final int MAX\_EMPLOYEES = 100;

static final int MAX\_TRANSACTIONS = 10000;

static final double SAVINGS\_INTEREST\_RATE = 0.02; // 2% annual

// File paths

static final String CUSTOMER\_FILE = "customers.txt";

static final String EMPLOYEE\_FILE = "employees.txt";

static final String TRANSACTION\_FILE = "transactions.txt";

static final String LOCKED\_ACCOUNTS\_FILE = "locked\_accounts.txt";

// Customer data arrays

static String[] customerName = new String[MAX\_CUSTOMERS];

static String[] customerSurname = new String[MAX\_CUSTOMERS];

static String[] usernames = new String[MAX\_CUSTOMERS];

static String[] emails = new String[MAX\_CUSTOMERS];

static String[] passwords = new String[MAX\_CUSTOMERS];

static String[] accountTypes = new String[MAX\_CUSTOMERS];

static String[] accountNumbers = new String[MAX\_CUSTOMERS];

static String[] cardNumbers = new String[MAX\_CUSTOMERS];

static String[] expiryDates = new String[MAX\_CUSTOMERS];

static String[] securityCodes = new String[MAX\_CUSTOMERS];

static double[] balances = new double[MAX\_CUSTOMERS];

static String[] branches = new String[MAX\_CUSTOMERS];

static String[] phoneNumbers = new String[MAX\_CUSTOMERS];

static int customerCount = 0;

// Employee data arrays

static String[] employeeUsernames = new String[MAX\_EMPLOYEES];

static String[] employeePasswords = new String[MAX\_EMPLOYEES];

static String[] employeeBranches = new String[MAX\_EMPLOYEES];

static String[] employeeRoles = new String[MAX\_EMPLOYEES];

static String[] employeeStatuses = new String[MAX\_EMPLOYEES];

static String[] employeeDescriptions = new String[MAX\_EMPLOYEES];

static int employeeCount = 0;

// Transaction data

static String[][][] transactions = new String[MAX\_CUSTOMERS][MAX\_TRANSACTIONS][4];

static int[] transactionCounts = new int[MAX\_CUSTOMERS];

// Security tracking

static Map<String, Integer> failedAttempts = new HashMap<>();

static Set<String> lockedAccounts = new HashSet<>();

// Admin credentials

static final String ADMIN\_USERNAME = "admin";

static final String ADMIN\_PASSWORD = "admin123";

static Scanner scanner = new Scanner(System.in);

/\*-------------------------------------------------

FILE HANDLING METHODS

-------------------------------------------------\*/

// Load all customer data

static void loadCustomers() {

Set<String> loadedUsernames = new HashSet<>();

try (BufferedReader reader = new BufferedReader(new FileReader(CUSTOMER\_FILE))) {

String line;

customerCount = 0;

while ((line = reader.readLine()) != null && customerCount < MAX\_CUSTOMERS) {

String[] parts = line.split(",");

if (parts.length >= 12) {

String username = parts[2].trim();

if (!loadedUsernames.contains(username)) {

customerName[customerCount] = parts[0].trim();

customerSurname[customerCount] = parts[1].trim();

usernames[customerCount] = username;

emails[customerCount] = parts[3].trim();

passwords[customerCount] = parts[4].trim();

accountTypes[customerCount] = parts[5].trim();

accountNumbers[customerCount] = parts[6].trim();

cardNumbers[customerCount] = parts[7].trim();

expiryDates[customerCount] = parts[8].trim();

securityCodes[customerCount] = parts[9].trim();

balances[customerCount] = Double.parseDouble(parts[10].trim());

branches[customerCount] = parts[11].trim();

if (parts.length > 12) {

phoneNumbers[customerCount] = parts[12].trim();

}

loadedUsernames.add(username);

customerCount++;

}

}

}

} catch (IOException | NumberFormatException e) {

System.out.println("No existing customer data found or data corrupted.");

}

}

// Save new customers (appends only)

static void saveCustomers() {

Set<String> existingUsernames = loadExistingUsernames(CUSTOMER\_FILE, 2);

try (FileWriter writer = new FileWriter(CUSTOMER\_FILE, true)) {

for (int i = 0; i < customerCount; i++) {

if (!existingUsernames.contains(usernames[i])) {

writer.write(String.join(",",

customerName[i], customerSurname[i], usernames[i],

emails[i], passwords[i], accountTypes[i], accountNumbers[i],

cardNumbers[i], expiryDates[i], securityCodes[i],

String.valueOf(balances[i]), branches[i], phoneNumbers[i]

) + "\n");

}

}

} catch (IOException e) {

System.out.println("Error saving customer data: " + e.getMessage());

}

}

// Load all employee data

static void loadEmployees() {

Set<String> loadedUsernames = new HashSet<>();

try (BufferedReader reader = new BufferedReader(new FileReader(EMPLOYEE\_FILE))) {

String line;

employeeCount = 0;

while ((line = reader.readLine()) != null && employeeCount < MAX\_EMPLOYEES) {

String[] parts = line.split(",");

if (parts.length >= 4) {

String username = parts[0].trim();

if (!loadedUsernames.contains(username)) {

employeeUsernames[employeeCount] = username;

employeePasswords[employeeCount] = parts[1].trim();

employeeBranches[employeeCount] = parts[2].trim();

employeeRoles[employeeCount] = parts[3].trim();

if (parts.length > 4) employeeStatuses[employeeCount] = parts[4].trim();

if (parts.length > 5) employeeDescriptions[employeeCount] = parts[5].trim();

loadedUsernames.add(username);

employeeCount++;

}

}

}

} catch (IOException e) {

System.out.println("No existing employee data found. Creating default admin...");

createDefaultAdmin();

}

}

// Save new employees (appends only)

static void saveEmployees() {

Set<String> existingUsernames = loadExistingUsernames(EMPLOYEE\_FILE, 0);

try (FileWriter writer = new FileWriter(EMPLOYEE\_FILE, true)) {

for (int i = 0; i < employeeCount; i++) {

if (!existingUsernames.contains(employeeUsernames[i])) {

writer.write(String.join(",",

employeeUsernames[i], employeePasswords[i],

employeeBranches[i], employeeRoles[i],

employeeStatuses[i] != null ? employeeStatuses[i] : "",

employeeDescriptions[i] != null ? employeeDescriptions[i] : ""

) + "\n");

}

}

} catch (IOException e) {

System.out.println("Error saving employee data: " + e.getMessage());

}

}

// Load transaction history

static void loadTransactions() {

try (BufferedReader reader = new BufferedReader(new FileReader(TRANSACTION\_FILE))) {

String line;

while ((line = reader.readLine()) != null) {

String[] parts = line.split("\\|");

if (parts.length >= 4) {

String accNumber = parts[0].trim();

// Find customer index by account number

for (int i = 0; i < customerCount; i++) {

if (accountNumbers[i].equals(accNumber) && transactionCounts[i] < MAX\_TRANSACTIONS) {

transactions[i][transactionCounts[i]] = new String[]{

parts[1].trim(), // date

parts[2].trim(), // type

parts[3].trim(), // amount

parts.length > 4 ? parts[4].trim() : "" // related account

};

transactionCounts[i]++;

break;

}

}

}

}

} catch (IOException e) {

System.out.println("No existing transaction data found.");

}

}

// Record a new transaction (appends to file)

static void recordTransaction(String accNumber, String type, double amount, String relatedAccount) {

String timestamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date());

String record = String.join("|",

accNumber, timestamp, type,

String.format("%.2f", amount),

relatedAccount != null ? relatedAccount : ""

);

// Save to file

try (FileWriter writer = new FileWriter(TRANSACTION\_FILE, true)) {

writer.write(record + "\n");

} catch (IOException e) {

System.out.println("Error saving transaction: " + e.getMessage());

}

// Update in-memory array

for (int i = 0; i < customerCount; i++) {

if (accountNumbers[i].equals(accNumber)) {

if (transactionCounts[i] < MAX\_TRANSACTIONS) {

transactions[i][transactionCounts[i]] = new String[]{

timestamp, type,

String.format("%.2f", amount),

relatedAccount != null ? relatedAccount : ""

};

transactionCounts[i]++;

}

break;

}

}

}

// Load locked accounts

static void loadLockedAccounts() {

try (BufferedReader reader = new BufferedReader(new FileReader(LOCKED\_ACCOUNTS\_FILE))) {

String line;

while ((line = reader.readLine()) != null) {

lockedAccounts.add(line.trim());

}

} catch (IOException e) {

System.out.println("No locked accounts found.");

}

}

// Save locked accounts

static void saveLockedAccounts() {

try (FileWriter writer = new FileWriter(LOCKED\_ACCOUNTS\_FILE)) {

for (String username : lockedAccounts) {

writer.write(username + "\n");

}

} catch (IOException e) {

System.out.println("Error saving locked accounts: " + e.getMessage());

}

}

// Helper method to load existing usernames from file

private static Set<String> loadExistingUsernames(String filename, int usernameFieldIndex) {

Set<String> usernames = new HashSet<>();

try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {

String line;

while ((line = reader.readLine()) != null) {

String[] parts = line.split(",");

if (parts.length > usernameFieldIndex) {

usernames.add(parts[usernameFieldIndex].trim());

}

}

} catch (IOException e) {

// File may not exist yet

}

return usernames;

}

// Create default admin account

private static void createDefaultAdmin() {

employeeUsernames[0] = "admin";

employeePasswords[0] = "admin123";

employeeBranches[0] = "Headquarters";

employeeRoles[0] = "Administrator";

employeeStatuses[0] = "Active";

employeeDescriptions[0] = "Default system administrator";

employeeCount = 1;

saveEmployees();

}

/\*-------------------------------------------------

CUSTOMER METHODS

-------------------------------------------------\*/

// Customer sign up

static void customerSignUp() {

loadCustomers();

if (customerCount >= MAX\_CUSTOMERS) {

System.out.println("System cannot accept more users.");

return;

}

System.out.println("\n===== CUSTOMER SIGN UP =====");

// Get user details with validation

String name = getInput("Enter your first name: ", "Name cannot be empty");

String surname = getInput("Enter your surname: ", "Surname cannot be empty");

String username;

while (true) {

username = getInput("Choose a username: ", "Username cannot be empty");

if (isUsernameTaken(username)) {

System.out.println("Username already exists. Please choose another.");

} else {

break;

}

}

String email;

while (true) {

email = getInput("Enter your email: ", "Email cannot be empty");

if (!email.matches("^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}$")) {

System.out.println("Invalid email format. Please try again.");

} else {

break;

}

}

String phone;

while (true) {

phone = getInput("Enter your phone number (10 digits): ", "Phone cannot be empty");

if (!phone.matches("\\d{10}")) {

System.out.println("Invalid phone number. Must be 10 digits.");

} else {

break;

}

}

String accountType;

while (true) {

accountType = getInput("Choose account type (Savings/Cheque): ", "Type cannot be empty");

if (accountType.equalsIgnoreCase("Savings") || accountType.equalsIgnoreCase("Cheque")) {

break;

}

System.out.println("Invalid account type. Please choose Savings or Cheque.");

}

String password;

while (true) {

password = getInput("Create a password (min 8 chars, 1 special character): ",

"Password cannot be empty");

if (!isValidPassword(password)) {

System.out.println("Password must be at least 8 characters with 1 special character.");

} else {

break;

}

}

String branch = getInput("Enter your branch: ", "Branch cannot be empty");

// Generate account details

String accNumber = "ACC" + (10000 + customerCount);

String cardNumber = "4111" + String.format("%012d", (long)(Math.random() \* 1000000000000L));

String expiry = String.format("%02d/%02d", (int)(Math.random() \* 12) + 1, (int)(Math.random() \* 10) + 25);

String cvv = String.format("%03d", (int)(Math.random() \* 1000));

// Store customer data

customerName[customerCount] = name;

customerSurname[customerCount] = surname;

usernames[customerCount] = username;

emails[customerCount] = email;

phoneNumbers[customerCount] = phone;

passwords[customerCount] = password;

accountTypes[customerCount] = accountType;

accountNumbers[customerCount] = accNumber;

cardNumbers[customerCount] = cardNumber;

expiryDates[customerCount] = expiry;

securityCodes[customerCount] = cvv;

balances[customerCount] = 0.0;

branches[customerCount] = branch;

customerCount++;

// Save to file

saveCustomers();

customerMenu(customerCount - 1);

}

// Customer login

static void customerLogin() {

loadCustomers();

loadLockedAccounts();

System.out.println("\n===== CUSTOMER LOGIN =====");

String username = getInput("Enter username: ", "Username required");

if (lockedAccounts.contains(username)) {

System.out.println("Account locked. Please contact support.");

return;

}

String password = getInput("Enter password: ", "Password required");

int customerIndex = -1;

for (int i = 0; i < customerCount; i++) {

if (usernames[i].equals(username) && passwords[i].equals(password)) {

customerIndex = i;

break;

}

}

if (customerIndex == -1) {

System.out.println("Invalid credentials.");

// Track failed attempts

int attempts = failedAttempts.getOrDefault(username, 0) + 1;

failedAttempts.put(username, attempts);

if (attempts >= 3) {

lockedAccounts.add(username);

saveLockedAccounts();

System.out.println("Account locked due to multiple failed attempts.");

} else {

System.out.println((3 - attempts) + " attempts remaining.");

}

return;

}

// Successful login

failedAttempts.remove(username);

System.out.println("Login successful!");

customerMenu(customerIndex);

}

// Customer menu

static void customerMenu(int customerIndex) {

while (true) {

System.out.println("\n===== CUSTOMER DASHBOARD =====");

System.out.println("Welcome, " + customerName[customerIndex] + " " + customerSurname[customerIndex] + "!");

System.out.println("Account: " + accountNumbers[customerIndex] +

" (" + accountTypes[customerIndex] + ")");

System.out.println("Balance: R" + String.format("%.2f", balances[customerIndex]));

System.out.println("Branch: " + branches[customerIndex]);

System.out.println("\n1. View Account Details");

System.out.println("2. Deposit Money");

System.out.println("3. Withdraw Money");

System.out.println("4. Transfer Money");

System.out.println("5. View Transaction History");

System.out.println("6. Logout");

int choice = getIntInput("Choose an option: ", 1, 6);

switch (choice) {

case 1:

showAccountDetails(customerIndex);

break;

case 2:

depositMoney(customerIndex);

break;

case 3:

withdrawMoney(customerIndex);

break;

case 4:

transferMoney(customerIndex);

break;

case 5:

viewTransactionHistory(customerIndex);

break;

case 6:

System.out.println("Logging out...");

return;

}

}

}

// Show account details

static void showAccountDetails(int customerIndex) {

System.out.println("\n===== ACCOUNT DETAILS =====");

System.out.println("Name: " + customerName[customerIndex] + " " + customerSurname[customerIndex]);

System.out.println("Username: " + usernames[customerIndex]);

System.out.println("Email: " + emails[customerIndex]);

System.out.println("Phone: " + phoneNumbers[customerIndex]);

System.out.println("Account Type: " + accountTypes[customerIndex]);

System.out.println("Account Number: " + accountNumbers[customerIndex]);

System.out.println("Card Number: " + cardNumbers[customerIndex]);

System.out.println("Expiry Date: " + expiryDates[customerIndex]);

System.out.println("Security Code: " + securityCodes[customerIndex]);

System.out.println("Balance: R" + String.format("%.2f", balances[customerIndex]));

System.out.println("Branch: " + branches[customerIndex]);

}

// Deposit money

static void depositMoney(int customerIndex) {

double amount = getDoubleInput("Enter amount to deposit: R", 0.01, 1000000);

balances[customerIndex] += amount;

recordTransaction(accountNumbers[customerIndex], "DEPOSIT", amount, null);

saveCustomers();

System.out.println("Successfully deposited R" + String.format("%.2f", amount));

System.out.println("New balance: R" + String.format("%.2f", balances[customerIndex]));

}

// Withdraw money

static void withdrawMoney(int customerIndex) {

double amount = getDoubleInput("Enter amount to withdraw: R", 0.01, balances[customerIndex]);

balances[customerIndex] -= amount;

recordTransaction(accountNumbers[customerIndex], "WITHDRAWAL", amount, null);

saveCustomers();

System.out.println("Successfully withdrew R" + String.format("%.2f", amount));

System.out.println("New balance: R" + String.format("%.2f", balances[customerIndex]));

}

// Transfer money

static void transferMoney(int senderIndex) {

String recipientAcc = getInput("Enter recipient account number: ", "Account number required");

// Find recipient

int recipientIndex = -1;

for (int i = 0; i < customerCount; i++) {

if (accountNumbers[i].equals(recipientAcc)) {

recipientIndex = i;

break;

}

}

if (recipientIndex == -1) {

System.out.println("Account not found.");

return;

}

if (recipientIndex == senderIndex) {

System.out.println("Cannot transfer to yourself.");

return;

}

double amount = getDoubleInput("Enter amount to transfer: R", 0.01, balances[senderIndex]);

// Perform transfer

balances[senderIndex] -= amount;

balances[recipientIndex] += amount;

// Record transactions

recordTransaction(accountNumbers[senderIndex], "TRANSFER\_OUT", amount, recipientAcc);

recordTransaction(accountNumbers[recipientIndex], "TRANSFER\_IN", amount, accountNumbers[senderIndex]);

saveCustomers();

System.out.println("Successfully transferred R" + String.format("%.2f", amount) +

" to account " + recipientAcc);

System.out.println("New balance: R" + String.format("%.2f", balances[senderIndex]));

}

// View transaction history

static void viewTransactionHistory(int customerIndex) {

System.out.println("\n===== TRANSACTION HISTORY =====");

System.out.printf("%-20s %-15s %-10s %-15s\n", "Date/Time", "Type", "Amount", "Related Account");

System.out.println("--------------------------------------------------");

for (int i = 0; i < transactionCounts[customerIndex]; i++) {

String[] t = transactions[customerIndex][i];

System.out.printf("%-20s %-15s R%-10s %-15s\n",

t[0], t[1], t[2], t[3].isEmpty() ? "N/A" : t[3]);

}

if (transactionCounts[customerIndex] == 0) {

System.out.println("No transactions found.");

}

}

/\*-------------------------------------------------

EMPLOYEE METHODS

-------------------------------------------------\*/

// Employee login

static void employeeLogin() {

loadEmployees();

loadLockedAccounts();

System.out.println("\n===== EMPLOYEE LOGIN =====");

String username = getInput("Enter username: ", "Username required");

if (lockedAccounts.contains(username)) {

System.out.println("Account locked. Please contact administrator.");

return;

}

String password = getInput("Enter password: ", "Password required");

String branch = getInput("Enter your branch: ", "Branch required");

int employeeIndex = -1;

for (int i = 0; i < employeeCount; i++) {

if (employeeUsernames[i].equals(username) &&

employeePasswords[i].equals(password) &&

employeeBranches[i].equals(branch)) {

employeeIndex = i;

break;

}

}

if (employeeIndex == -1) {

System.out.println("Invalid credentials or branch mismatch.");

// Track failed attempts

int attempts = failedAttempts.getOrDefault(username, 0) + 1;

failedAttempts.put(username, attempts);

if (attempts >= 3) {

lockedAccounts.add(username);

saveLockedAccounts();

System.out.println("Account locked due to multiple failed attempts.");

} else {

System.out.println((3 - attempts) + " attempts remaining.");

}

return;

}

// Successful login

failedAttempts.remove(username);

System.out.println("Login successful!");

employeeMenu(employeeIndex);

}

// Employee menu

static void employeeMenu(int employeeIndex) {

while (true) {

System.out.println("\n===== EMPLOYEE DASHBOARD =====");

System.out.println("Welcome, " + employeeUsernames[employeeIndex] +

" (" + employeeRoles[employeeIndex] + ")");

System.out.println("Branch: " + employeeBranches[employeeIndex]);

System.out.println("\n1. Add New Customer");

System.out.println("2. View Customer Details");

System.out.println("3. View All Customers");

System.out.println("4. Reset Customer Password");

System.out.println("5. Apply Monthly Interest");

System.out.println("6. Logout");

int choice = getIntInput("Choose an option: ", 1, 6);

switch (choice) {

case 1:

addCustomerByEmployee(employeeBranches[employeeIndex]);

break;

case 2:

searchCustomer();

break;

case 3:

viewAllCustomers();

break;

case 4:

resetCustomerPassword();

break;

case 5:

applyInterest();

break;

case 6:

System.out.println("Logging out...");

return;

}

}

}

// Add customer by employee

static void addCustomerByEmployee(String branch) {

loadCustomers();

if (customerCount >= MAX\_CUSTOMERS) {

System.out.println("System cannot accept more customers.");

return;

}

System.out.println("\n===== ADD NEW CUSTOMER =====");

// Get customer details

String name = getInput("Enter customer's first name: ", "Name required");

String surname = getInput("Enter customer's surname: ", "Surname required");

String username;

while (true) {

username = getInput("Choose a username: ", "Username required");

if (isUsernameTaken(username)) {

System.out.println("Username already exists.");

} else {

break;

}

}

String email;

while (true) {

email = getInput("Enter email: ", "Email required");

if (!email.matches("^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}$")) {

System.out.println("Invalid email format.");

} else {

break;

}

}

String phone;

while (true) {

phone = getInput("Enter phone number (10 digits): ", "Phone required");

if (!phone.matches("\\d{10}")) {

System.out.println("Must be 10 digits.");

} else {

break;

}

}

String accountType;

while (true) {

accountType = getInput("Account type (Savings/Cheque): ", "Type required");

if (accountType.equalsIgnoreCase("Savings") || accountType.equalsIgnoreCase("Cheque")) {

break;

}

System.out.println("Invalid type. Choose Savings or Cheque.");

}

String password;

while (true) {

password = getInput("Set password (min 8 chars, 1 special): ", "Password required");

if (!isValidPassword(password)) {

System.out.println("Password requirements not met.");

} else {

break;

}

}

// Generate account details

String accNumber = "ACC" + (10000 + customerCount);

String cardNumber = "4111" + String.format("%012d", (long)(Math.random() \* 1000000000000L));

String expiry = String.format("%02d/%02d", (int)(Math.random() \* 12) + 1, (int)(Math.random() \* 10) + 25);

String cvv = String.format("%03d", (int)(Math.random() \* 1000));

// Store customer data

customerName[customerCount] = name;

customerSurname[customerCount] = surname;

usernames[customerCount] = username;

emails[customerCount] = email;

phoneNumbers[customerCount] = phone;

passwords[customerCount] = password;

accountTypes[customerCount] = accountType;

accountNumbers[customerCount] = accNumber;

cardNumbers[customerCount] = cardNumber;

expiryDates[customerCount] = expiry;

securityCodes[customerCount] = cvv;

balances[customerCount] = 0.0;

branches[customerCount] = branch;

customerCount++;

// Save to file

saveCustomers();

}

// View all customers

static void viewAllCustomers() {

loadCustomers();

System.out.println("\n===== ALL CUSTOMERS =====");

System.out.printf("%-5s %-20s %-15s %-10s %-15s %-10s\n",

"No.", "Name", "Account", "Type", "Balance", "Branch");

System.out.println("------------------------------------------------------------");

for (int i = 0; i < customerCount; i++) {

System.out.printf("%-5d %-20s %-15s %-10s R%-15.2f %-10s\n",

i + 1,

customerName[i] + " " + customerSurname[i],

accountNumbers[i],

accountTypes[i],

balances[i],

branches[i]);

}

if (customerCount == 0) {

System.out.println("No customers found.");

}

}

// Search for customer

static void searchCustomer() {

loadCustomers();

String searchTerm = getInput("Enter account number or username: ", "Search term required");

for (int i = 0; i < customerCount; i++) {

if (accountNumbers[i].equals(searchTerm) || usernames[i].equals(searchTerm)) {

showAccountDetails(i);

return;

}

}

System.out.println("Customer not found.");

}

// Reset customer password

static void resetCustomerPassword() {

loadCustomers();

String username = getInput("Enter customer username: ", "Username required");

int customerIndex = -1;

for (int i = 0; i < customerCount; i++) {

if (usernames[i].equals(username)) {

customerIndex = i;

break;

}

}

if (customerIndex == -1) {

System.out.println("Customer not found.");

return;

}

String newPassword;

while (true) {

newPassword = getInput("Enter new password (min 8 chars, 1 special): ", "Password required");

if (!isValidPassword(newPassword)) {

System.out.println("Password requirements not met.");

} else {

break;

}

}

passwords[customerIndex] = newPassword;

saveCustomers();

System.out.println("Password reset successfully for " + username);

}

// Apply monthly interest

static void applyInterest() {

loadCustomers();

System.out.println("\nApplying monthly interest to savings accounts...");

int count = 0;

for (int i = 0; i < customerCount; i++) {

if (accountTypes[i].equalsIgnoreCase("Savings") && balances[i] > 0) {

double interest = balances[i] \* (SAVINGS\_INTEREST\_RATE / 12);

balances[i] += interest;

recordTransaction(accountNumbers[i], "INTEREST", interest, null);

count++;

}

}

saveCustomers();

System.out.println("Applied interest to " + count + " savings accounts.");

}

/\*-------------------------------------------------

ADMIN METHODS

-------------------------------------------------\*/

// Admin login

static void adminLogin() {

System.out.println("\n===== ADMIN LOGIN =====");

String username = getInput("Username: ", "Username required");

String password = getInput("Password: ", "Password required");

if (username.equals(ADMIN\_USERNAME) && password.equals(ADMIN\_PASSWORD)) {

adminMenu();

} else {

System.out.println("Invalid admin credentials.");

}

}

// Admin menu

static void adminMenu() {

while (true) {

System.out.println("\n===== ADMIN DASHBOARD =====");

System.out.println("1. Add Employee");

System.out.println("2. View All Employees");

System.out.println("3. Search Employee");

System.out.println("4. Reset Employee Password");

System.out.println("5. Unlock Accounts");

System.out.println("6. View All Customers");

System.out.println("7. Apply Monthly Interest");

System.out.println("8. Logout");

int choice = getIntInput("Choose an option: ", 1, 8);

switch (choice) {

case 1:

addEmployee();

break;

case 2:

viewAllEmployees();

break;

case 3:

searchEmployee();

break;

case 4:

resetEmployeePassword();

break;

case 5:

unlockAccounts();

break;

case 6:

viewAllCustomers();

break;

case 7:

applyInterest();

break;

case 8:

System.out.println("Logging out...");

return;

}

}

}

// Add employee

static void addEmployee() {

loadEmployees();

if (employeeCount >= MAX\_EMPLOYEES) {

System.out.println("Maximum employees reached.");

return;

}

System.out.println("\n===== ADD NEW EMPLOYEE =====");

String username;

while (true) {

username = getInput("Enter username: ", "Username required");

if (isEmployeeUsernameTaken(username)) {

System.out.println("Username already exists.");

} else {

break;

}

}

String password;

while (true) {

password = getInput("Enter password (min 8 chars, 1 special): ", "Password required");

if (!isValidPassword(password)) {

System.out.println("Password requirements not met.");

} else {

break;

}

}

String branch = getInput("Enter branch: ", "Branch required");

String role = getInput("Enter role: ", "Role required");

String status = getInput("Enter status (Active/Inactive): ", "Status required");

String description = getInput("Enter description: ", "Description required");

// Store employee data

employeeUsernames[employeeCount] = username;

employeePasswords[employeeCount] = password;

employeeBranches[employeeCount] = branch;

employeeRoles[employeeCount] = role;

employeeStatuses[employeeCount] = status;

employeeDescriptions[employeeCount] = description;

employeeCount++;

// Save to file

saveEmployees();

System.out.println("Employee added successfully!");

}

// View all employees

static void viewAllEmployees() {

loadEmployees();

System.out.println("\n===== ALL EMPLOYEES =====");

System.out.printf("%-5s %-15s %-15s %-15s %-10s\n",

"No.", "Username", "Branch", "Role", "Status");

System.out.println("------------------------------------------------");

for (int i = 0; i < employeeCount; i++) {

System.out.printf("%-5d %-15s %-15s %-15s %-10s\n",

i + 1,

employeeUsernames[i],

employeeBranches[i],

employeeRoles[i],

employeeStatuses[i]);

}

if (employeeCount == 0) {

System.out.println("No employees found.");

}

}

// Search employee

static void searchEmployee() {

loadEmployees();

String searchTerm = getInput("Enter username: ", "Search term required");

for (int i = 0; i < employeeCount; i++) {

if (employeeUsernames[i].equals(searchTerm)) {

System.out.println("\n===== EMPLOYEE DETAILS =====");

System.out.println("Username: " + employeeUsernames[i]);

System.out.println("Branch: " + employeeBranches[i]);

System.out.println("Role: " + employeeRoles[i]);

System.out.println("Status: " + employeeStatuses[i]);

System.out.println("Description: " + employeeDescriptions[i]);

return;

}

}

System.out.println("Employee not found.");

}

// Reset employee password

static void resetEmployeePassword() {

loadEmployees();

String username = getInput("Enter employee username: ", "Username required");

int employeeIndex = -1;

for (int i = 0; i < employeeCount; i++) {

if (employeeUsernames[i].equals(username)) {

employeeIndex = i;

break;

}

}

if (employeeIndex == -1) {

System.out.println("Employee not found.");

return;

}

String newPassword;

while (true) {

newPassword = getInput("Enter new password (min 8 chars, 1 special): ", "Password required");

if (!isValidPassword(newPassword)) {

System.out.println("Password requirements not met.");

} else {

break;

}

}

employeePasswords[employeeIndex] = newPassword;

saveEmployees();

System.out.println("Password reset successfully for " + username);

}

// Unlock accounts

static void unlockAccounts() {

loadLockedAccounts();

if (lockedAccounts.isEmpty()) {

System.out.println("No accounts are currently locked.");

return;

}

System.out.println("\n===== LOCKED ACCOUNTS =====");

int i = 1;

for (String username : lockedAccounts) {

System.out.println(i++ + ". " + username);

}

String input = getInput("\nEnter username to unlock or 'all': ", "Input required");

if (input.equalsIgnoreCase("all")) {

lockedAccounts.clear();

System.out.println("All accounts unlocked.");

} else if (lockedAccounts.contains(input)) {

lockedAccounts.remove(input);

System.out.println("Account " + input + " unlocked.");

} else {

System.out.println("Account not found in locked list.");

}

saveLockedAccounts();

}

/\*-------------------------------------------------

UTILITY METHODS

-------------------------------------------------\*/

// Check if username is taken

static boolean isUsernameTaken(String username) {

for (int i = 0; i < customerCount; i++) {

if (usernames[i].equalsIgnoreCase(username)) {

return true;

}

}

return false;

}

// Check if employee username is taken

static boolean isEmployeeUsernameTaken(String username) {

for (int i = 0; i < employeeCount; i++) {

if (employeeUsernames[i].equalsIgnoreCase(username)) {

return true;

}

}

return false;

}

// Validate password

static boolean isValidPassword(String password) {

return password.length() >= 8 &&

password.matches(".\*[!@#$%^&\*].\*");

}

// Get validated string input

static String getInput(String prompt, String errorMsg) {

while (true) {

System.out.print(prompt);

String input = scanner.nextLine().trim();

if (!input.isEmpty()) {

return input;

}

System.out.println(errorMsg);

}

}

// Get validated integer input

static int getIntInput(String prompt, int min, int max) {

while (true) {

System.out.print(prompt);

try {

int input = Integer.parseInt(scanner.nextLine());

if (input >= min && input <= max) {

return input;

}

System.out.println("Please enter a number between " + min + " and " + max);

} catch (NumberFormatException e) {

System.out.println("Invalid input. Please enter a number.");

}

}

}

// Get validated double input

static double getDoubleInput(String prompt, double min, double max) {

while (true) {

System.out.print(prompt);

try {

double input = Double.parseDouble(scanner.nextLine());

if (input >= min && input <= max) {

return input;

}

System.out.printf("Please enter an amount between R%.2f and R%.2f\n", min, max);

} catch (NumberFormatException e) {

System.out.println("Invalid input. Please enter a number.");

}

}

}

/\*-------------------------------------------------

MAIN MENU SYSTEM

-------------------------------------------------\*/

// Main menu

static void mainMenu() {

while (true) {

System.out.println("\n===== DBT BANKING SYSTEM =====");

System.out.println("1. Customer Sign Up");

System.out.println("2. Customer Login");

System.out.println("3. Employee Login");

System.out.println("4. Admin Login");

System.out.println("5. Exit");

int choice = getIntInput("Choose an option: ", 1, 5);

switch (choice) {

case 1:

customerSignUp();

break;

case 2:

customerLogin();

break;

case 3:

employeeLogin();

break;

case 4:

adminLogin();

break;

case 5:

System.out.println("Thank you for using our banking system. Goodbye!");

System.exit(0);

}

}

}

// Initialize system

public static void main(String[] args) {

// Load all data at startup

loadCustomers();

loadEmployees();

loadTransactions();

loadLockedAccounts();

// Start the system

mainMenu();

}

}

SQL Code Snippet:

---Create New Database

CREATE DATABASE Banking\_System;

---Use the database

USE Banking\_System;

---Create table customers

CREATE TABLE Customers(

Customer\_ID INT PRIMARY KEY,

Name VARCHAR(100) NOT NULL,

Surname VARCHAR(100) NOT NULL,

Email VARCHAR(100) UNIQUE,

Phone VARCHAR(20) NOT NULL,

Address VARCHAR(150) NOT NULL

);

---Populate the table with appropriate entities and attributes

INSERT INTO customers(Customer\_ID,Name,Surname,Email,Phone,Address)

VALUEs('C1','James','Anderson','James.anderson@gmail.com','0712345678','12 Elm street,yeovile'),

('C2','Samuel','Mkhize','SamuelMkhi@gmail.com','0723456789','23 Moi Avenue'),

('C3','Michael','Smith','MichaelM@gmail.com','073334456','56 Tom Mboya Street'),

('C4','Jason','Smith','JasonJason@gmail.com','0858734908','34 Troye street'),

('C5','David','Nkomo','DavidDiva@gmail.com','0987896543','534 braamfontein complex'),

('C6','Lawrence','Gopane','lawlaw@gmail.com','0878976543','13 grant avenue'),

('C7','Nicholas','Thabethe','NicholasThabethe@gmail.com','0678905628','237 medrine avenue'),

('C8','Chris','Mapetla','ChrisM@gmail.com','07890679876','154 street,soweto'),

('C9','Daniel','Jones','DanielDJones@gmail.com','0987896578','57 park road,midrand'),

('C10','George','Sampsom','GeorgeGman@gmail.com','0987890989','15 parklane'),

('C11','Justin','De klerk','JustinDe\_klerk@gmail.com','0768965430','28 arthur rd'),

('C12','patrick','mofokeng','patpatrick2gmail.com','0987654329','200 paterson street'),

('C13','marcus','mamba','marcusmamaba@gmail.com','0676897665','74 corner eloff and vvon brandis street'),

('C14','Primrose','Baloyi','primBaloyi@gmail.com','0876745890','90 student street, lenasia'),

('C15','lesedi','Mohale','leeleelesedi@gmail.com','0987843211','654 thokoza street'),

('C16','Thandeka','Mangena','Thandeka@gmail.com','0957755833','27 Joe slovo street'),

('C17','Lindiwe','Moyo','LindiweMoyo@gmail.com','0872367232','885 davis avenue'),

('C18','Sipho','Yeni','SiphoY@gmail.com','0679234722','908 Midrand flat'),

('C19','Melusi','Duma','MelusiDuma@gmail.com','0789237486','81 shipton road'),

('C20','Thembi','Majola','ThembiMajola@gmail.com','0734673873','404 room, Thembisa');

---Create table Accounts

CREATE TABLE Accounts(

Account\_ID INT PRIMARY KEY,

Customer\_ID INT,

Account\_Type VARCHAR(50) NOT NULL,

Card\_Number INT NOT NULL,

Expiry\_Date DATE NOT NULL,

Security\_Code INT NOT NULL,

Balance DECIMAL(10,2) NOT NULL,

FOREIGN KEY (Customer\_ID) REFERENCES customers(Customer\_ID)

);

---Populate the table with appropriate attributes and entities

INSERT INTO accounts(Account\_ID,Customer\_ID, Account\_Type, Card\_Number,Expiry\_Date,Security\_Code,Balance)

VALUES ('A01','C1','Savings','9234785672354542','78/36','8733',15000.00),

('A02','C2','Cheque','9473646464763425','08/43','2622',3500.00),

('A03','C3','Savings','7234623682988372','73/71','5123',150.00),

('A04','C4','Savings','7853467576487657','98/90','5765',2500.50),

('A05','C5','Savings','8975463455434557','09/87','4656',00.00),

('A06','C6','Cheque','8745678476578568','98/08','8880',289.00),

('A07','C7','Savings','9877465353675559','97/56','5767',27000.00),

('A08','C8','Savings','4325656465555555','54/64','5464',750.00),

('A09','C9','Savings','545665465545767','44/55','1323',350.00),

('A010','C10','Savings','2157712363287878','61/25','1621',50.00),

('A011','C11','Savings','6125816783126268','21/65','1879',450.25),

('A012','C12','Savings','2378627868238625','12/67','0188',4500),

('A013','C13','Savings','6176682637826223','09/02','7829',550.45),

('A014','C14','Cheque','7389787348943896','33/44','7813',6522.00),

('A015','C15','Savings','761255756712357867','67/66','7612',2300.90),

('A016','C16','Savings','7523757237823767','27/82','1771',9000.00),

('A017','C17','Savings','6712817648789317','76/67','7812',4700.00),

('A018','C18','Cheque','1781236832663243','23/32','3443',00.00),

('A019','C19','Cheque','2847629729789723','27/22','2378',6123),

('A020','C20','Cheque','8957768346234234','89/47','2782',00.00);

---Create table transaction records

CREATE TABLE Transaction\_Records(

Transaction\_ID INT PRIMARY KEY,

Account\_ID INT,

Transaction\_Type VARCHAR(20) NOT NULL,

Amount DECIMAL(10,2) NOT NULL,

Transaction\_Date DATE NOT NULL,

FOREIGN KEY (Account\_ID) REFERENCES Accounts(Account\_ID)

);

---Populate table with appropriate entities and attributes

INSERT INTO transaction\_records(Transaction\_ID, Account\_ID,Transaction\_Type,Amount,Transaction\_Date)

VALUES ('T001','A01','Transfer',500.00,'2025-06-01'),

('T002','A02','Deposit',500.00,'2025-06-02'),

('T003','A03','Withdraw',150.00,'2025-06-03'),

('T004','A04','Withdraw',500.00,'2025-06-04'),

('T005','A05','Deposit',1000.00,'2025-06-05'),

('T006','A06','Deposit',3500.00,'2025-06-06'),

('T007','A07','Withdraw',2000.00,'2025-06-07'),

('T008','A08','Transfer',700.00,'2025-06-08'),

('T009','A09','Deposit',350.00,'2025-06-09'),

('T010','A10','Deposit',1500.00,'2025-06-10'),

('T011','A11','Deposit',4500.00,'2025-06-11'),

('T012','A12','Withdraw',500.00,'2025-06-11'),

('T013','A13','Transfer',300.00,'2025-06-01'),

('T014','A14','Transfer',4500.00,'2025-06-02'),

('T015','A15','Withdraw',2000.00,'2025-06-03'),

('T016','A16','Transfer',3000.00,'2025-06-04'),

('T017','A17','Transfer',700.00,'2025-06-05'),

('T018','A18','Deposit',100.00,'2025-06-06'),

('T019','A19','Withdraw',600.00,'2025-06-07'),

('T020','A20','Deposit',200.00,'2025-06-08');

---Create table branch\_information

CREATE TABLE Branch\_Information(

Branch\_ID INT PRIMARY KEY,

Branch\_Name VARCHAR(100) NOT NULL,

Branch\_Location VARCHAR(100) NOT NULL,

Contact\_Number VARCHAR(20) NOT NULL

);

---Populate table with appropriate entities and attributes

INSERT INTO branch\_information(Branch\_ID,Branch\_Name, Branch\_Location,Contact\_Number)

VALUES ('B101','DBT Bank','Johannesburg Station east','0115509237'),

('B102','DBT Bank','Johannesburg Braamfontein','0115508793'),

('B103','DBT Bank','Johannesburg Pritchard','0115508743'),

('B104','DBT Bank','Johannesburg Bree','0115501237'),

('B105','DBT Bank','Johannesburg Jewel City','0115503467'),

('B106','DBT Bank','Johannesburg carlton centre','0115507622'),

('B107','DBT Bank','Johannesburg kerk','0115503723'),

('B108','DBT Bank','Johannesburg Station west','0115502623'),

('B109','DBT Bank','Johannesburg plain street','0115509238'),

('B110','DBT Bank','Johannesburg Hillbrow','0115502738');

---Create table employees

CREATE TABLE Employees(

Employee\_ID INT PRIMARY KEY,

Name VARCHAR(100) NOT NULL,

Position VARCHAR(50) NOT NULL,

Status VARCHAR(50) NOT NULL,

Description VARCHAR(100) NOT NULL,

Schedule\_Date DATE NOT NULL,

Branch\_ID INT,

FOREIGN KEY (branch\_id) REFERENCES branches(branch\_id)

);

---Populate table with appropriate entities and attributes

INSERT INTO employees(Employee\_ID,Branch\_ID,Name,Position,Status, Description,Schedule\_Date)

VALUES ('E1001','B101','Sibusiso','Bank teller','Active','handles customer transactions','8:00'),

('E1002','B102','Michael','Auditor','on leave','Conduct internal audits','10:30'),

('E1003','B103','Lindiwe','Personal banker','Active','personalized banking services','7:00'),

('E1004','B104','Mark','investment banker','Terminated','advice clients on financial transactions','7:00'),

('E1005','B105','Anna','Credit analyst','inactive','analyze loan applicants','12:00'),

('E1006','B106','Siyabonga','bank teller','Inactive','handles customer transactions','10:00'),

('E1007','B107','Themba','Auditor','active','conduct internal aduits','6:30'),

('E1008','B108','Thembi','personal banker','inactive','deals with personal loans','7:30'),

('E1009','B109','Melusi','investment banker','on leave','advice customers on transfer of money','7:00'),

('E1010','B110','Sthabile','credit analyst','active','analyze loan applications','8:00'),

('E1011','B105','Thuli','bank teller','on leave','handles customer transactions','8:00'),

('E1012','B104','mandla','auditor','inactive','conduct internal audits','7:30'),

('E1013','B109','nick','personal banker','active','deals personal loan','7:00'),

('E1014','B102','tebogo','investment banker','active','advice clients on safe transactions','8:00'),

('E1015','B106','Jackson','credit analyst','inactive','anaylze loan request','7:30'),

('E1016','B101','sifiso','bank teller','inactive','handles customer transactions','06:30'),

('E1017','B110','oratile','auditor','on leave','deals with internal audits','14:00'),

('E1018','B103','tshamio','personal banker','active','deals with personal loan','9:00'),

('E1019','B108','Sam','investment banker','inactive','advice customers on safe transactions','11:00'),

('E1020','B107','Paul','credit analyst','terminated','analyze laon request','10:00');

-------------SQL QUERIES--------------------

--Get all customers' full names and emails

SELECT Name, Surname, Email FROM Customers;

--Retrieve all accounts with a balance greater than R1000

SELECT \* FROM Accounts WHERE Balance > 1000;

--Get transactions of type 'Withdraw'

SELECT \* FROM Transaction\_Records WHERE Transaction\_Type = 'Withdraw';

--List all branches in Johannesburg Braamfontein

SELECT \* FROM Branch\_Information WHERE Branch\_Location = 'Johannesburg Braamfontein';

--Count how many customers there are

SELECT COUNT(\*) AS Total\_Customers FROM Customers;

--Calculate the total money in all accounts

SELECT SUM(Balance) AS Total\_Funds FROM Accounts;

--Get the average transaction amount

SELECT AVG(Amount) AS Average\_Transaction FROM Transaction\_Records;

--Find the minimum account balance

SELECT MIN(Balance) AS Min\_Balance FROM Accounts;

--Count how many employees are 'on leave'

SELECT COUNT(\*) AS OnLeave\_Employees FROM Employees WHERE Status = 'on leave';

--List customer names with their account types and balances

SELECT c.Name, c.Surname, a.Account\_Type, a.Balance

FROM Customers c

JOIN Accounts a ON c.Customer\_ID = a.Customer\_ID;

--Show transaction details along with account type

SELECT t.Transaction\_ID, t.Transaction\_Type, t.Amount, a.Account\_Type

FROM Transaction\_Records t

JOIN Accounts a ON t.Account\_ID = a.Account\_ID;

--Get employee names with their branch locations

SELECT e.Name, e.Position, b.Branch\_Location

FROM Employees e

JOIN Branch\_Information b ON e.Branch\_ID = b.Branch\_ID;

--List administration staff and their email addresses

SELECT Name, Surname, Email FROM Adminstration;

--Get each branch and the number of employees assigned there

SELECT b.Branch\_Name, COUNT(e.Employee\_ID) AS Num\_Employees

FROM Branch\_Information b

JOIN Employees e ON b.Branch\_ID = e.Branch\_ID

GROUP BY b.Branch\_Name;

--Update the balance of a specific account (e.g., A01) by adding R1000

UPDATE Accounts SET Balance = Balance + 1000 WHERE Account\_ID = 'A01';

--Change the status of employee E1004 to 'inactive'

UPDATE Employees SET Status = 'inactive' WHERE Employee\_ID = 'E1004';

--Delete customers with no email address (if any)

DELETE FROM Customers WHERE Email IS NULL;

--Fix typo in email of admin Mandla Moyo

UPDATE Adminstration

SET Email = 'MandlaMoyo@gmail.com'

WHERE Name = 'Mandla' AND Surname = 'Moyo';

--Add R500 interest to all savings accounts

UPDATE Accounts

SET Balance = Balance + 500

WHERE Account\_Type = 'Savings';