Simple Banking Application

PROJECT REVIEW

Submitted by

RITIK RAJ

Roll No: 2301301363

In Partial Fulfillment of the Requirements for
the Degree of

BACHOLAR OF TECHNOLOGY
4th SEM

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

QUANTUM UNIVERSITY

ROORKEE- 247167

Session: 2024-25

INDEX

Simple Banking Application

Abstract

This report presents the design, implementation, and analysis of a Simple Bank System developed as a Java application. The system simulates basic banking operations including account creation, deposits, withdrawals, and balance inquiries within a console-based interface. Employing object-oriented programming principles, the application utilizes a single-class architecture with an inner class for account representation and a HashMap for in-memory data storage. The implementation prioritizes simplicity and clarity while maintaining essential functionality and data integrity through comprehensive input validation and error handling. This project demonstrates fundamental programming concepts applicable to financial software systems and provides a foundation for more complex banking applications. The report discusses design decisions, implementation challenges, testing approaches, and potential future enhancements. The Simple Bank System serves as both an educational tool for understanding Java programming concepts and a practical demonstration of software design for financial transaction processing.

Introduction

Banking systems are fundamental components of modern financial infrastructure, providing essential services for managing financial assets and transactions. This report details the design and implementation of a Simple Bank System developed as a Java application. The system aims to simulate basic banking operations in a simplified yet functional manner, making it suitable for educational purposes while demonstrating core programming concepts.

The primary motivation behind this project was to create a practical application that showcases fundamental programming principles while addressing real-world requirements of financial transaction processing. By implementing a banking system, the project emphasizes important concepts such as data management, user interaction, transaction processing, and system security at a basic level.

Banking systems have evolved significantly over the decades, from manual ledger-based systems to sophisticated digital platforms. This project represents a simplified model of modern banking software, focusing on core functionality rather than advanced features. The system provides a foundation that could be expanded with more complex features in future iterations.

Project Overview

Project Scope

The Simple Bank System is designed to provide a basic yet comprehensive banking experience within a single Java application. The system accommodates the following core banking operations:

1. Account creation and management
2. Deposit functionality
3. Withdrawal processing
4. Balance inquiries
5. Account listing and overview

The project deliberately limits its scope to these fundamental operations to maintain simplicity while still demonstrating essential programming concepts.

Objectives

The primary objectives of the Simple Bank System project are:

1. To create a functional bank account management system using Java
2. To implement basic banking operations (deposit, withdrawal, balance check)
3. To develop a user-friendly command-line interface
4. To ensure data integrity and basic validation
5. To demonstrate object-oriented programming principles
6. To establish a foundation for potential future enhancements

Target Users

The system is designed primarily for educational purposes and targets:

1. Computer science students learning Java programming
2. Instructors teaching programming fundamentals
3. Developers interested in financial system implementation basics

System Design

Architecture

The Simple Bank System employs a straightforward monolithic architecture contained within a single Java class file. This design decision prioritizes simplicity and clarity while still adhering to good programming practices. The system follows a structured approach with:

1. **Main Class**: Serves as the application entry point and controller
2. **Nested Account Class**: Encapsulates account data and operations
3. **Static Methods**: Handle system-wide operations and user interface
4. **In-Memory Data Store**: Utilizes a HashMap for account storage

The architecture employs a simple Model-View-Controller (MVC) pattern, where:

- The Account class represents the Model (data)
- The menu and display methods constitute the View
- The operation methods (createAccount, deposit, etc.) serve as the Controller

Data Model

The system's data model is centered around the Account class, which contains:

- **accountNumber**: String identifier for the account
- **holderName**: Name of the account holder
- **balance**: Current account balance

This simplified data model captures the essential attributes needed for basic banking operations while maintaining a clear structure.

User Interface Design

The user interface utilizes a console-based menu system that provides:

1. Clear options for all available operations
2. Structured input prompts
3. Immediate feedback on actions
4. Error handling for invalid inputs
5. Formatted display of account information

While simple, the interface is designed to be intuitive and user-friendly, guiding users through the banking operations with clear instructions and feedback.

Implementation Details

Development Environment

The Simple Bank System was developed using:

- Java Development Kit (JDK) 11+
- Standard Java libraries only (no external dependencies)
- A text editor or IDE (such as IntelliJ IDEA, Eclipse, or VS Code)

Core Components

The system consists of the following core components:

1. **SimpleBankSystem Class**: The main class containing the application logic and user interface.
2. **Account Inner Class**: A nested class that encapsulates account-related data and operations.
3. **Account Management Methods**: Functions for creating, accessing, and managing accounts.
4. **Transaction Methods**: Functions handling deposits and withdrawals.
5. **User Interface Methods**: Functions for displaying menus and processing user input.
6. **Utility Methods**: Helper functions for input validation and account number generation.

Technical Choices

Several key technical decisions shaped the implementation:

1. **Single-File Design**: The entire system is contained in one file to simplify distribution, compilation, and execution.
2. **Inner Class for Accounts**: Using an inner class for accounts maintains encapsulation while keeping related code together.
3. **HashMap for Storage**: A HashMap was chosen for account storage due to its efficient key-based retrieval, which is ideal for account lookup by account number.
4. **Scanner for Input**: Java's Scanner class provides simple yet effective user input handling.
5. **Static Methods and Variables**: Static methods and variables enable straightforward access to system functionality without complex instantiation.

6. **Input Validation**: Robust input validation ensures system stability and data integrity.

Key Features

Account Management

The system implements comprehensive account management features:

1. **Account Creation**: Users can create new accounts with a unique account number, holder name, and initial balance.
2. **Unique Account Numbers**: The system automatically generates sequential account numbers with an "ACC" prefix (e.g., ACC1001, ACC1002).
3. **Account Information Display**: Users can view detailed account information including the account number, holder name, and current balance.
4. **Account Listing**: The system can display all existing accounts, facilitating oversight of the entire account portfolio.

Transaction Processing

Transaction processing capabilities include:

1. **Deposits**: Users can add funds to accounts with immediate balance updates.
2. **Withdrawals**: Users can remove funds from accounts, subject to sufficient balance verification.
3. **Balance Checks**: Users can check the current balance of any account in the system.

Input Validation and Error Handling

The system implements several validation and error handling mechanisms:

1. **Numeric Input Validation**: Ensures that amounts and menu selections are valid numbers.
2. **Account Existence Validation**: Verifies that referenced accounts exist before attempting operations.
3. **Transaction Amount Validation**: Ensures that transaction amounts are positive values.
4. **Insufficient Funds Checking**: Prevents withdrawals that would result in negative balances.
5. **Error Messages**: Provides clear, specific error messages to guide users when issues occur.

Code Structure Analysis

Class Structure

The SimpleBankSystem class serves as both the main application class and the container for the nested Account class. This structure provides several advantages:

1. **Encapsulation**: The Account class encapsulates all account-related data and behaviors.
2. **Namespace Management**: The nested class structure prevents namespace pollution.
3. **Logical Grouping**: Related functionality is kept together in a logical structure.

The code follows object-oriented design principles including:

- **Encapsulation**: Account data is accessible only through appropriate methods.
- **Abstraction**: Complex operations are abstracted behind simple method calls.
- **Single Responsibility**: Methods and classes have clearly defined responsibilities.

Method Organization

The methods within the SimpleBankSystem class are organized into logical groups:

1. **Main and Control Methods**: main(), displayMenu()
2. **Account Operation Methods**: createAccount(), deposit(), withdraw(), checkBalance(), listAllAccounts()
3. **Utility Methods**: findAccount(), generateAccountNumber()
4. **Input Handling Methods**: getIntInput(), getDoubleInput(), getStringInput()

This organization enhances code readability and maintainability by grouping related functionality.

Data Management

The system employs an in-memory data storage approach using a HashMap:

```
private static Map<String, Account> accounts = new HashMap<>();
```

This implementation provides:

1. **Efficient Retrieval**: O(1) time complexity for account lookup by account number
2. **Dynamic Sizing**: Automatic expansion as new accounts are added
3. **Key-Value Association**: Natural mapping of account numbers to account objects

While this approach is suitable for the project's scope, a production system would likely use persistent storage (database) instead of in-memory storage.

Testing and Validation

Testing Approach

The Simple Bank System can be tested through several approaches:

1. **Manual Testing**: Executing all system functions through the console interface
2. **Unit Testing**: Creating JUnit tests for individual methods
3. **Integration Testing**: Testing sequences of operations to verify correct behavior

Test Scenarios

Key test scenarios for the system include:

1. **Account Creation**:
    - Creating an account with valid inputs
    - Attempting to create an account with negative initial balance
    - Verifying unique account number generation
2. **Deposit Operations**:
    - Depositing valid amounts
    - Attempting to deposit negative or zero amounts
    - Depositing to non-existent accounts
3. **Withdrawal Operations**:
    - Withdrawing valid amounts
    - Attempting withdrawals exceeding the balance
    - Attempting to withdraw negative or zero amounts
    - Withdrawing from non-existent accounts
4. **Balance and Account Inquiries**:
    - Checking balances of existing accounts
    - Attempting to check non-existent accounts
    - Listing accounts with various numbers of accounts in the system
5. **Input Validation**:
    - Providing non-numeric input for numeric fields
    - Testing boundary conditions for inputs

Validation Results

Manual testing of the system reveals that it correctly handles:

1. Basic account operations
2. Input validation
3. Error conditions
4. Edge cases like zero or negative amounts

The system successfully maintains data integrity throughout operations and provides appropriate feedback to users.

Challenges and Solutions

Design Challenges

1. **Challenge**: Balancing simplicity with functionality **Solution**: Focused on core banking operations while maintaining clean code structure
2. **Challenge**: Implementing effective data storage **Solution**: Used HashMap for efficient account retrieval while maintaining simplicity

Implementation Challenges

1. **Challenge**: Robust input validation **Solution**: Created dedicated methods for different input types with error handling
2. **Challenge**: Maintaining account data integrity **Solution**: Implemented validation checks before all operations that modify account data
3. **Challenge**: Generating unique account numbers **Solution**: Used a static counter with prefix to ensure uniqueness

Learning Outcomes

The development of this system provided valuable insights into:

1. Object-oriented design principles
2. Data structure selection for specific use cases
3. User interface design for command-line applications
4. Input validation techniques

5. Error handling strategies
6. Financial transaction processing logic

Future Enhancements

The Simple Bank System could be extended with various features in future iterations:

Functional Enhancements

1. **Account Types**: Add support for different account types (savings, checking, etc.)
2. **Interest Calculation**: Implement interest accrual for savings accounts
3. **Transaction History**: Maintain a log of all transactions for each account
4. **Fund Transfers**: Enable transfers between accounts
5. **Multiple Currencies**: Support for multiple currencies and exchange rates

Technical Enhancements

1. **Persistent Storage**: Implement database storage for account data
2. **Authentication System**: Add user authentication for security
3. **GUI Interface**: Develop a graphical user interface
4. **RESTful API**: Create an API for integration with other systems
5. **Multi-threading**: Improve performance with concurrent operations

Security Enhancements

1. **Encryption**: Implement data encryption for sensitive information
2. **Access Control**: Add role-based access control
3. **Transaction Verification**: Implement multi-factor authentication for transactions
4. **Audit Logging**: Create comprehensive audit trails of all system activities

Conclusion

The Simple Bank System project successfully demonstrates the implementation of a basic banking application using Java. Despite its simplicity, the system incorporates essential banking operations, data management, and user interaction components in a structured and maintainable codebase.

Through this project, fundamental programming concepts such as object-oriented design, data encapsulation, input validation, and error handling have been applied to create a functional application. The system serves as both an educational tool and a foundation for more complex banking software development.

The design choices prioritized simplicity and clarity while still adhering to good programming practices, resulting in a system that is easy to understand, maintain, and extend. The implementation successfully balances the competing requirements of functionality, simplicity, and code quality.

While the current system is limited in scope, it provides a solid foundation that could be extended with additional features and improved with more advanced technologies in future iterations. The modular design allows for such enhancements without requiring a complete restructuring of the existing code.

In summary, the Simple Bank System project achieves its objectives of creating a functional banking application while demonstrating core programming principles and establishing a foundation for future development.

References

Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley Professional.

Friesen, J. (2019). *Java by Comparison: Become a Java Craftsman in 70 Examples*. Pragmatic Bookshelf.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Horstmann, C. S. (2021). *Core Java, Volume I--Fundamentals* (12th ed.). Pearson.

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications.

Oracle. (2020). *Java Platform, Standard Edition Documentation*. Retrieved from https://docs.oracle.com/en/java/javase/

Sierra, K., & Bates, B. (2005). *Head First Java* (2nd ed.). O'Reilly Media.

Schildt, H. (2019). *Java: The Complete Reference* (11th ed.). McGraw-Hill Education.

Wagner, B., & Rushing, A. (2020). *Design Patterns in C# and .NET Core*. Springer.

Yener, M., & Theedom, A. (2018). *Professional Java EE Design Patterns*. Wrox.

Appendix: Source Code

```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class SimpleBankSystem {
    // Inner class to represent a bank account
    private static class Account {
        private String accountNumber;
        private String holderName;
        private double balance;

        public Account(String accountNumber, String holderName, double
initialBalance) {
            this.accountNumber = accountNumber;
            this.holderName = holderName;
            this.balance = initialBalance;
        }

        public String getAccountNumber() {
            return accountNumber;
        }

        public String getHolderName() {
            return holderName;
        }

        public double getBalance() {
            return balance;
        }

        public void deposit(double amount) {
            if (amount <= 0) {
                System.out.println("Error: Deposit amount must be positive.");
                return;
            }
            balance += amount;
            System.out.println("Deposited: $" + amount);
            System.out.println("New Balance: $" + balance);
        }

        public void withdraw(double amount) {
            if (amount <= 0) {
                System.out.println("Error: Withdrawal amount must be positive.");
                return;
            }
            if (amount > balance) {
                System.out.println("Error: Insufficient funds.");
                return;
            }
```

```
        balance -= amount;
        System.out.println("Withdrawn: $" + amount);
        System.out.println("New Balance: $" + balance);
    }

    public void displayInfo() {
        System.out.println("\nAccount Information:");
        System.out.println("Account Number: " + accountNumber);
        System.out.println("Account Holder: " + holderName);
        System.out.println("Current Balance: $" + balance);
    }
}

// Main class variables
private static Map<String, Account> accounts = new HashMap<>();
private static Scanner scanner = new Scanner(System.in);
private static int accountCounter = 1000;

// Main method
public static void main(String[] args) {
    boolean running = true;

    System.out.println("Welcome to Simple Bank System!");

    while (running) {
        displayMenu();
        int choice = getIntInput("Enter your choice: ");

        switch (choice) {
            case 1:
                createAccount();
                break;
            case 2:
                deposit();
                break;
            case 3:
                withdraw();
                break;
            case 4:
                checkBalance();
                break;
            case 5:
                listAllAccounts();
                break;
            case 6:
                running = false;
                System.out.println("Thank you for using Simple Bank System!");
                break;
            default:
                System.out.println("Invalid choice. Please try again.");
        }
    }

    scanner.close();
}

// Display main menu
private static void displayMenu() {
    System.out.println("\n===== SIMPLE BANK SYSTEM =====");
    System.out.println("1. Create New Account");
    System.out.println("2. Deposit Money");
    System.out.println("3. Withdraw Money");
    System.out.println("4. Check Balance");
    System.out.println("5. List All Accounts");
```

```
        System.out.println("6. Exit");
        System.out.println("============================");
    }

    // Create a new account
    private static void createAccount() {
        System.out.println("\n--- Create New Account ---");
        String name = getStringInput("Enter account holder name: ");
        double initialDeposit = getDoubleInput("Enter initial deposit amount: $");

        if (initialDeposit < 0) {
            System.out.println("Error: Initial deposit cannot be negative.");
            return;
        }

        String accountNumber = generateAccountNumber();
        Account newAccount = new Account(accountNumber, name, initialDeposit);
        accounts.put(accountNumber, newAccount);

        System.out.println("\nAccount created successfully!");
        System.out.println("Your account number is: " + accountNumber);
        newAccount.displayInfo();
    }

    // Generate unique account number
    private static String generateAccountNumber() {
        return "ACC" + (++accountCounter);
    }

    // Deposit money into an account
    private static void deposit() {
        System.out.println("\n--- Deposit Money ---");
        Account account = findAccount();

        if (account == null) {
            return;
        }

        double amount = getDoubleInput("Enter deposit amount: $");
        account.deposit(amount);
    }

    // Withdraw money from an account
    private static void withdraw() {
        System.out.println("\n--- Withdraw Money ---");
        Account account = findAccount();

        if (account == null) {
            return;
        }

        double amount = getDoubleInput("Enter withdrawal amount: $");
        account.withdraw(amount);
    }

    // Check account balance
    private static void checkBalance() {
        System.out.println("\n--- Check Balance ---");
        Account account = findAccount();

        if (account == null) {
            return;
        }
```

```
        account.displayInfo();
    }

    // List all accounts
    private static void listAllAccounts() {
        System.out.println("\n--- All Accounts ---");

        if (accounts.isEmpty()) {
            System.out.println("No accounts found.");
            return;
        }

        for (Account account : accounts.values()) {
            account.displayInfo();
            System.out.println("--------------------");
        }
        System.out.println("Total accounts: " + accounts.size());
    }

    // Find an account by account number
    private static Account findAccount() {
        String accountNumber = getStringInput("Enter account number: ");
        Account account = accounts.get(accountNumber);

        if (account == null) {
            System.out.println("Error: Account not found.");
            return null;
        }

        return account;
    }

    // Helper method to get integer input
    private static int getIntInput(String prompt) {
        while (true) {
            System.out.print(prompt);
            try {
                return Integer.parseInt(scanner.nextLine());
            } catch (NumberFormatException e) {
                System.out.println("Invalid input. Please enter a number.");
            }
        }
    }

    // Helper method to get double input
    private static double getDoubleInput(String prompt) {
        while (true) {
            System.out.print(prompt);
            try {
                return Double.parseDouble(scanner.nextLine());
            } catch (NumberFormatException e) {
                System.out.println("Invalid input. Please enter a number.");
            }
        }
    }

    // Helper method to get string input
    private static String getStringInput(String prompt) {
        System.out.print(prompt);
        return scanner.nextLine();
    }
}
```