Software Engineering



Submission Date: 28 Jnuary 2025.

UOC-BSCS-F2023-021

Hassan Fawad Haider

Semester: 3rd – CS Department (Fall 2023)

DEPARTMENT OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY UNIVERSITY OF CHAKWAL

**Software Requirements Specification (SRS)**

**Table of Contents**

1. **Introduction**
   * 1.1 Purpose
   * 1.2 Document Conventions
   * 1.3 Intended Audience and Reading Suggestions
   * 1.4 Project Scope
   * 1.5 References
2. **Overall Description**
   * 2.1 Product Perspective
   * 2.2 Product Features
   * 2.3 User Classes and Characteristics
   * 2.4 Operating Environment
   * 2.5 Design and Implementation Constraints
   * 2.6 User Documentation
   * 2.7 Assumptions and Dependencies
3. **System Features**
   * 3.1 System Feature 1
   * 3.2 System Feature 2
4. **External Interface Requirements**
   * 4.1 User Interfaces
   * 4.2 Hardware Interfaces
   * 4.3 Software Interfaces
   * 4.4 Communications Interfaces
5. **Other Nonfunctional Requirements**
   * 5.1 Performance Requirements
   * 5.2 Safety Requirements
   * 5.3 Security Requirements
   * 5.4 Software Quality Attributes
6. **Other Requirements**

Appendix A: Glossary  
Appendix B: Analysis Models  
Appendix C: Issues List

**1. Introduction**

**1.1 Purpose**

The purpose of this document is to define the software requirements for the Account Management System. It serves as a reference for stakeholders, developers, and testers throughout the software development lifecycle.

**1.2 Document Conventions**

* **Terminology**: Italics indicate new terms; bold indicates emphasis.
* **Units**: All monetary values are in USD.
* **Dates**: Follow the format YYYY-MM-DD.

**1.3 Intended Audience and Reading Suggestions**

* **Developers**: Focus on Sections 2 and 3 for implementation details.
* **Testers**: Review Section 5 for nonfunctional requirements and constraints.
* **Stakeholders**: Review Sections 1 and 2 for an overview of the system scope.

**1.4 Project Scope**

The Account Management System is designed to manage and differentiate between savings and transaction accounts. It provides functionalities for secure login, account management, and viewing account types.

**1.5 References**

* IEEE SRS Template
* C++ Documentation: [https://cplusplus.com](https://cplusplus.com/)
* UML Modeling Standards

**2. Overall Description**

**2.1 Product Perspective**

This system is a standalone application to handle various account types. It is designed to ensure modularity and scalability using object-oriented programming principles.

**2.2 Product Features**

* Secure account creation and management.
* Support for multiple account types (SavingsAccount, TransactionAccount).
* Polymorphic behavior using account type-specific methods.

**2.3 User Classes and Characteristics**

* **Administrators**: Can create, delete, and modify accounts.
* **Account Holders**: Can view and manage their respective accounts.

**2.4 Operating Environment**

* **Operating System**: Windows 10 or later / Linux-based OS.
* **Programming Language**: C++.
* **Hardware**: Minimum 4 GB RAM, 1 GHz processor.

**2.5 Design and Implementation Constraints**

* The application must adhere to OOP principles, utilizing inheritance and polymorphism.
* Limited to desktop environments, no web-based access.

**2.6 User Documentation**

User manuals and tutorials will be provided to ensure smooth adoption.

**2.7 Assumptions and Dependencies**

* Users are assumed to have basic computer literacy.
* The system depends on a local database for storing account detai

**3. System Features**

**3.1 System Feature 1: Account Management**

* **Description**: Enables secure account creation and deletion.
* **Functional Requirements**:
  + Input account details (AccountNo, Password, Balance).
  + Validate and store account information.

**3.2 System Feature 2: Account Type Differentiation**

* **Description**: Provides unique behaviors for savings and transaction accounts.
* **Functional Requirements**:
  + Implement polymorphic methods for account type-specific features.
  + Display account type using displayAccountType() method.

**4. External Interface Requirements**

**4.1 User Interfaces**

* Graphical User Interface with fields for input and buttons for operations.

**4.2 Hardware Interfaces**

* Requires a keyboard and monitor for interaction.

**4.3 Software Interfaces**

* Integrates with SQLite for local data storage.

**4.4 Communications Interfaces**

* No external communication required for the current implementation.

**5. Other Nonfunctional Requirements**

**5.1 Performance Requirements**

* System should process account operations within 0.5 seconds.

**5.2 Safety Requirements**

* Ensure secure handling of sensitive data (e.g., passwords).

**5.3 Security Requirements**

* Passwords must be stored securely using hashing techniques.

**5.4 Software Quality Attributes**

* **Reliability**: 99.9% uptime.
* **Maintainability**: Code should follow OOP design patterns for easy modification.

**6. Other Requirements**

Future versions may include support for web and mobile platforms.

**Appendices**

**Appendix A: Glossary**

* **AccountNo**: A unique identifier for an account.
* **Balance**: The monetary value associated with an account.
* **Polymorphism**: The ability of different classes to respond to the same method call in different ways.

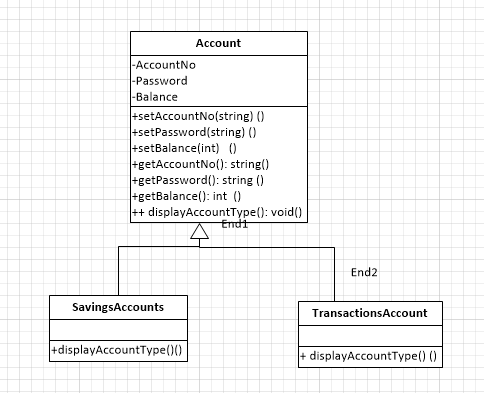
**Appendix B: Analysis Models**

* UML diagrams for class structure and interactions.

**Appendix C: Issues List**

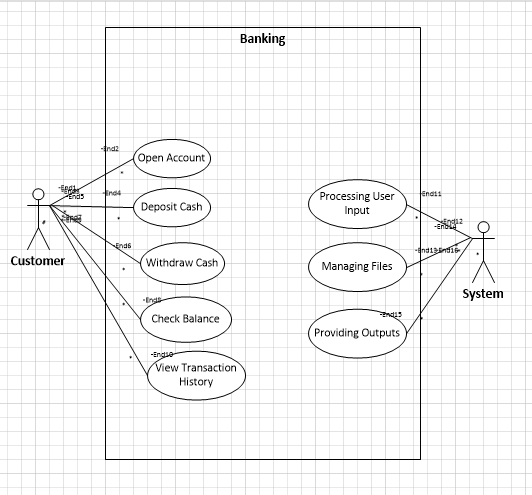
* **Issue 1**: Potential delays in database queries.
* **Issue 2**: Adding more account types requires additional testing.

Class Diagram:



1. Base Class: Account
   * Purpose:
     + Acts as a general template for all account types.
     + Contains attributes (AccountNo, Password, Balance) that are common to all accounts.
     + Provides methods to set and get these attributes.
     + Includes a virtual method displayAccountType() that can be overridden by derived classes.
   * Attributes:
     + AccountNo: Represents the unique identifier for an account.
     + Password: Stores the password for secure access.
     + Balance: Maintains the monetary balance in the account.
   * Methods:
     + Setters and getters for all attributes.
     + A virtual method displayAccountType() to define the type of account.
2. Derived Class: SavingsAccount
   * Purpose:
     + Represents a specific type of account (savings).
     + Inherits attributes and methods from the Account class.
     + Overrides the displayAccountType() method to specify that it is a savings account.
   * Key Characteristics:
     + Shares all attributes and functionality with the Account class.
     + No additional attributes or methods are added in this class.
3. Derived Class: TransactionAccount
   * Purpose:
     + Represents another specific type of account (transaction).
     + Inherits attributes and methods from the Account class.
     + Overrides the displayAccountType() method to specify that it is a transaction account.
   * Key Characteristics:
     + Similar to SavingsAccount, it shares the base functionality of the Account class without adding new attributes or methods.
4. Relationships
   * Inheritance:
     + Both SavingsAccount and TransactionAccount inherit from the Account class.
     + This is depicted with a generalization relationship (a solid line with a hollow triangle pointing to the base class).
   * Shared Behavior:
     + Since SavingsAccount and TransactionAccount are derived from Account, they share the common attributes and behaviors but can define their own specialized behaviors via method overriding.
5. Design Principles
   * Reusability:
     + Common functionality is encapsulated in the Account base class, avoiding duplication.
   * Polymorphism:
     + The displayAccountType() method allows different account types to define their specific behavior, even when accessed through a pointer or reference to the base class.
   * Extensibility:
     + The design allows for easy addition of new account types by creating new classes that inherit from Account.

Use case diagram:



Actors in the Diagram

1. User (Primary Actor):
   * Represents the person who interacts with the system to perform various operations, such as creating an account, depositing cash, withdrawing money, and checking account information.
   * The user directly initiates all the use cases in the system.
2. System (Optional Secondary Actor):
   * Optionally depicted as an internal component or actor responsible for backend operations like updating files, logging transactions, and retrieving data.

System Boundary

* The Bank Account Management System is represented as a rectangular boundary enclosing all the use cases.
* It signifies that all the use cases are functionalities provided by the system.

Use Cases and Descriptions

1. Open Account:
   * The user provides account details (Account No, Password) to create a new account.
   * The system records these details and initializes the balance to zero.
2. Deposit Cash:
   * The user inputs the account number and the amount to deposit.
   * The system updates the account balance and logs the transaction in a file for future reference.
3. Withdraw Cash:
   * The user provides the account number, password, and the amount to withdraw.
   * The system verifies the account details and ensures sufficient balance before deducting the amount and updating the account.
4. Check Balance:
   * The user inputs the account number to view the current balance.
   * The system retrieves the balance from the stored account data and displays it to the user.
5. View Transaction History:
   * The user requests a list of past transactions associated with their account.
   * The system searches the transaction history file and retrieves all relevant transactions for display.

Fully Dressed Usecase:

Use Case: Deposit Cash

1. Use Case Name:

Deposit Cash

1. Actors:
   * Primary Actor: User (The person initiating the deposit).
   * Secondary Actors:
     + Bank System (Manages the backend processing of the deposit).
     + Account File (Stores account information, like account numbers and balances).
     + Transaction History File (Stores transaction logs, including deposits).
2. Description:

The User initiates a request to deposit cash into their account. The system verifies the account, updates the balance, logs the transaction, and displays the updated balance to the user.

1. Preconditions:
   * The User has an existing account in the bank system.
   * The Bank System is online and functional.
   * The Account File (Account.txt) and Transaction History File (TransactionHistory.txt) exist and are accessible.
2. Postconditions:
   * The deposit amount is added to the user’s account balance.
   * A transaction record is created and stored in the Transaction History File (TransactionHistory.txt).
   * The User receives the updated balance after the deposit.
3. Flow of Events:

Basic Flow (Main Success Scenario):

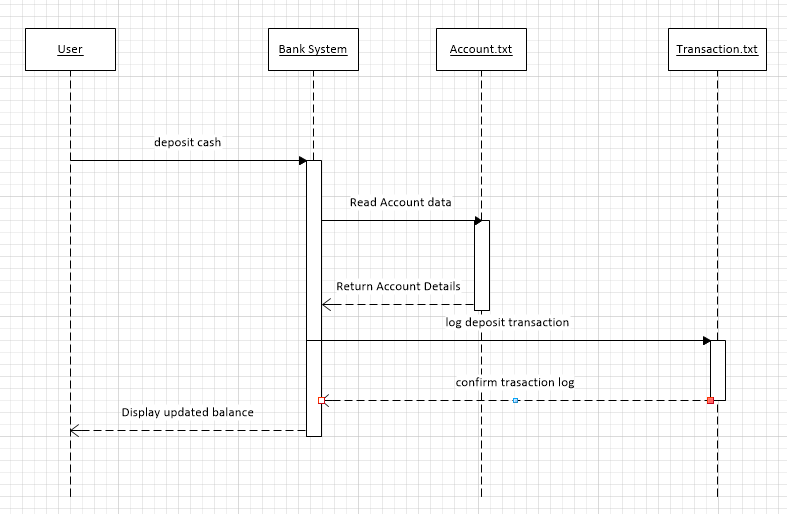
1. User Initiates Deposit:
   * The User opens the banking application or interface.
   * The User selects the "Deposit Cash" option and enters their account number and the deposit amount.
   * The User submits the deposit request.
2. System Verifies Account:
   * The Bank System receives the request and verifies if the provided account number exists by querying the Account File (Account.txt).
   * If the account is found, the Bank System retrieves the current balance associated with that account.
3. System Updates Balance:
   * The Bank System adds the deposit amount to the current balance of the account.
   * The updated balance is stored in the Account File (Account.txt).
4. System Logs the Transaction:
   * The Bank System creates a new Transaction object with details like the transaction type ("Deposit"), deposit amount, and timestamp.
   * The Bank System logs the transaction by writing it to the Transaction History File (TransactionHistory.txt).
5. User Receives Updated Balance:
   * The Bank System displays the updated balance to the User.
   * The User receives the confirmation of the deposit and the new balance. Alternative Flow (Account Not Found):
6. User Submits Deposit Request:
   * The User initiates the "Deposit Cash" action by entering the account number and deposit amount.
7. System Cannot Find Account:
   * The Bank System searches for the account number in the Account File (Account.txt).
   * If the account number is not found, the Bank System displays an error message indicating that the account does not exist.
8. User Corrects or Cancels:
   * The User can correct the account number or cancel the transaction. Alternative Flow (Invalid Deposit Amount):
9. User Enters Invalid Deposit Amount:
   * The User submits a deposit with a negative or zero value.
10. System Detects Invalid Amount:
    * The Bank System checks the deposit amount and detects that it is invalid (negative or zero).
    * The Bank System displays an error message, indicating that the deposit amount is invalid.
11. User Corrects Amount:
    * The User corrects the amount and resubmits the request.
12. Special Requirements:
    * The system must ensure that the user is authenticated before initiating the deposit action.
    * The Account File and Transaction History File must be maintained securely, with proper access controls to prevent unauthorized modifications.
13. Extension Points:
    * The system can be extended to support additional deposit methods (e.g., mobile payment, check deposit) or multiple currencies in the future.
    * The system could include an audit log for all transaction operations (e.g., account updates, transaction logs) for tracking and reporting.
14. Frequency of Use:
    * This use case is expected to be used frequently, as it involves a core banking operation— cash deposits—by customers.
15. Assumptions:
    * The User has basic knowledge of how to perform banking operations.
    * The Bank System has been set up to handle deposit actions and to interact with the account and transaction files correctly.

Example in Action:

Scenario: A user wants to deposit $500 into their checking account.

1. The User logs into the banking system, enters their account number, and requests a deposit of $500.
2. The Bank System verifies that the account exists in the Account File (Account.txt).
3. The current balance for the account is retrieved (e.g., $1,000).
4. The Bank System updates the balance to $1,500 and logs the deposit in the Transaction History File (TransactionHistory.txt).
5. The User receives a confirmation of the deposit and the updated balance of $1,500.

Sequence Diagram:



Identify Key Objects (Lifelines):

* + User: The individual interacting with the system. Represents the initiator of actions.
  + Bank System: The main system responsible for processing requests, verifying data, and updating records.
  + Account File (Account.txt): A file used by the system to store account details like account numbers and balances.
  + Transaction History File (TransactionHistory.txt): A file where transaction records are stored, including deposits.

These lifelines will be placed horizontally along the top of the sequence diagram.

1. Define the Interactions (Messages):

Next, we define the sequence of messages that occur between these objects:

* + User to System:
    - The user initiates the "Deposit Cash" action by entering the account number and deposit amount.
    - The system receives these inputs as a message.
  + System to Account File (Account.txt):
    - The system queries the account information by reading the Account.txt file.
  + System Processing:
    - The system verifies whether the account exists in the file.
    - If the account is found, the system updates the account balance by adding the deposit amount.
  + System to Transaction History File (TransactionHistory.txt):
    - The system logs the deposit action in TransactionHistory.txt, storing the transaction details (like account number, deposit amount, and date).
  + System to User:
    - The system sends the updated balance back to the user, displaying the new amount in their account.

1. Order of Events for the Deposit Cash Scenario:

Let’s break down the order of events:

1. User initiates action:
   * The User sends a message to the Bank System, requesting to deposit cash, along with the account number and deposit amount.
2. System verifies account:
   * The Bank System reads the Account.txt file to check if the account exists.
   * If found, the system retrieves the current balance for that account.
3. Balance update:
   * The Bank System calculates the new balance by adding the deposit amount to the current balance.
4. Logging the transaction:
   * The Bank System writes the transaction details (account number, deposit amount, timestamp) to the TransactionHistory.txt file.
5. User receives updated balance:
   * Finally, the Bank System sends a message back to the User, displaying the updated balance after the deposit.