Banking System Documentation

Tran Le Dung (MSSV: 24110084)

September 16, 2025

1 Object-Oriented Analysis (OOA)

The banking system manages customer accounts, transactions, and savings accounts with interest. The key entities are:

- **Customer**: Represents a bank customer who owns multiple accounts.
- **Account**: Base class for bank accounts, storing account number, owner, balance, and transaction history.
- **SavingsAccount**: Derived from Account, adding interest rate and withdrawal fees.
- **Transaction**: Captures financial activities (deposit, withdrawal, transfer, fee) with amount, type, and date.
- **Time**: Manages date information for transactions.

The system supports operations like opening accounts, depositing/withdrawing funds, transferring money, applying interest, and displaying transaction history. Relationships include:

- A Customer has multiple Accounts (one-to-many).
- An Account contains a list of Transactions (one-to-many).
- SavingsAccount inherits from Account (is-a relationship).

2 Class Design

The system uses a hierarchical class structure with inheritance and operator overloading to enhance functionality and usability.

2.1 Inheritance

- **Account Class**: Encapsulates common account attributes (ID, account number, balance, owner name, transaction history) and methods (deposit, withdraw, transfer). It uses a virtual destructor and virtual methods (`displayInfo`, `withdraw`) to support polymorphism.

- **SavingsAccount Class**: Inherits from Account, adding an interest rate attribute and overriding `withdraw` to include a $5 fee. The `applyInterest` method calculates and adds interest as a deposit transaction.

Inheritance enables code reuse (e.g., `deposit` is reused in SavingsAccount) and supports polymorphic behavior (e.g., calling `displayInfo` on Account or SavingsAccount objects).

2.2 Operator Overloading

Operator overloading provides intuitive interaction with Account objects:

- +=: Applies a transaction (deposit or withdrawal) to an account, updating balance and history.
- ==: Compares two accounts based on balance.
- <, >: Compares account balances for sorting or ranking.
- «, »: Stream operators for outputting account details and inputting account data.

These operators simplify code, making operations like `*acc1 += t1` more readable than explicit method calls like `deposit` or `withdraw`.

3 Code Walkthrough

Key components of the C++ code are highlighted below, focusing on operator overloading and critical functionality:

- **Transaction Class**: Stores transaction details (ID, amount, type, date). The constructor auto-generates unique IDs using a static counter (`cnt`).
- **Account Class**: The += operator checks the transaction type and calls `deposit` or `withdraw`. The « operator outputs account details in a formatted string, while » prompts user input for account creation. The `transferTo` method updates balances and transaction histories for both accounts involved.
- **SavingsAccount Class**: Overrides `withdraw` to deduct a $5 fee, adding both the withdrawal and fee as transactions.

Example of += operator implementation:

```
Account& operator+=(const Transaction& other) {
    if (other.getType() == 1) deposit(other);
    else if (other.getType() == 2) withdraw(other);
    else cout << "Unsupported transaction type for +=.\n";
    return *this;
}
```

This allows intuitive transaction application, e.g., `*acc1 += t1`, which deposits or withdraws based on the transaction type. The `transferTo` method ensures

atomic updates to both accounts' balances and histories, maintaining consistency.

4 Test Results

The main function demonstrates the system's functionality. Below is a sample output from the program:

```
Customer ID: C01 | Name: Alice | Total Balance: 15000
Accounts:
Account ID: A1 | Number: A1001 | Owner: Alice | Balance: 5000
Savings Account ID: A2 | Number: S2001 | Owner: Alice | Balance:
   10000
   Interest Rate: 5%

[After transactions]
Transfer 1500 from A1001 to S2001 successful!
Customer ID: C01 | Name: Alice | Total Balance: 15000
Accounts:
Account ID: A1 | Number: A1001 | Owner: Alice | Balance: 4500
Savings Account ID: A2 | Number: S2001 | Owner: Alice | Balance:
   10500
   Interest Rate: 5%

Transaction history for account A1001:
Transaction ID: T1 | Amount: 2000 | Type: Deposit | Date: 15/09/2025
Transaction ID: T2 | Amount: 1000 | Type: Withdrawal | Date:
   16/09/2025
Transaction ID: T3 | Amount: 1500 | Type: Transfer Out | Date:
   17/09/2025

Transaction history for account S2001:
Transaction ID: T4 | Amount: 1500 | Type: Transfer In | Date:
   17/09/2025

===== Operator Overloading Tests =====
Account 1: Account ID: A1 | Number: A1001 | Owner: Alice | Balance:
   4500
Account 2: Account ID: A2 | Number: S2001 | Owner: Alice | Balance:
   10500

Enter new account details using >> operator:
Enter account number: A1002
Enter owner name: Bob
Enter initial balance: 3000
You entered: Account ID: A3 | Number: A1002 | Owner: Bob | Balance:
   3000

Testing operator+= (deposit 500 into acc1)
Account ID: A1 | Number: A1001 | Owner: Alice | Balance: 5000
```

```
35
36 Testing operator==
37 acc1 and acc2 do NOT have equal balance.
38
39 Testing operator< and operator>
40 acc1 has LESS balance than acc2.
```

This output demonstrates:

- Successful account creation and initial balances.
- Correct execution of deposit, withdrawal, and transfer operations.
- Operator overloading functionality:
- «: Displays account details clearly.
- »: Captures user input for a new account.
- +=: Applies a deposit transaction, increasing acc1's balance to 5000.
- ==, <, >: Accurately compare account balances, showing acc1 (5000) is less than acc2 (10500).

5 LLM Usage

I used an LLM (Grok, created by xAI) to brainstorm ideas for operator overloading in the banking system. The prompt was: "Suggest ways to overload operators in a bank account class to make it more intuitive." The LLM suggested overloading += for applying transactions, == for balance comparison, and «, » for stream operations. I implemented these ideas myself, ensuring they aligned with the system's design. The LLM was used solely for ideation, and I wrote all code independently.