Bank Account Management System

Object-Oriented Analysis and Design Documentation

Student: Đoàn Trọng Trung
Student ID: 24110140
Course: Object-Oriented Programming
Assignment: Week 6 - Inheritance and Operator Overloading

---

1. Object-Oriented Analysis (OOA) Model

1.1 Identify Objects (Nouns)

The main entities in the Bank Account Management System are:

- **Account: The base class representing a general bank account**
- **SavingsAccount: A specialized account with interest rates and withdrawal limits**
- **BusinessAccount: A specialized account for business operations with transaction fees**
- **Transaction: Represents banking transactions with amount, type, and date**
- **RecurringTransaction: A specialized transaction that occurs regularly**
- **Customer: Represents a bank customer who can own multiple accounts**
- **Operations: System management class that handles all banking operations**

1.2 Identify Attributes for Each Object

Account Class Attributes:

- **accountNumber (int): Unique identifier for the account**
- **balance (double): Current account balance**
- **ownerName (string): Name of the account owner**
- **transactionHistory (vector<Transaction>): List of all transactions**
- **nextAccountNumber (static int): Auto-incrementing account number generator**

SavingsAccount Class Attributes (inherits from Account):

- **interestRate (double): Annual interest rate**
- **withdrawalLimit (double): Maximum withdrawal amount per transaction**

- **withdrawalsThisMonth (int): Number of withdrawals made this month**
- **MAX_WITHDRAWALS (const int): Maximum allowed withdrawals per month (6)**

BusinessAccount Class Attributes (inherits from Account):

- **transactionFee (double): Fee charged per transaction**
- **creditLimit (double): Available credit limit beyond balance**

Transaction Class Attributes:

- **amount (double): Transaction amount**
- **type (string): Transaction type (Deposit, Withdrawal, etc.)**
- **date (string): Transaction date and time**

Customer Class Attributes:

- **name (string): Customer's full name**
- **customerID (int): Unique customer identifier**
- **accounts (vector<Account*>): List of accounts owned by customer**
- **nextCustomerID (static int): Auto-incrementing customer ID generator**

1.3 Identify Methods for Each Object (Verbs)

Account Class Methods:

- **deposit(double amount): Add money to the account**
- **withdraw(double amount): Remove money from the account**
- **transfer(Account& toAccount, double amount): Transfer money between accounts**
- **displayInfo(): Show account information and transaction history**
- **Overloaded operators: +=, ==, >, <<**

SavingsAccount Class Methods (inherits from Account):

- **withdraw(double amount) (overridden): Withdraw with restrictions**
- **applyInterest(): Calculate and apply monthly interest**
- **resetMonthlyWithdrawals(): Reset withdrawal counter**
- **displayInfo() (overridden): Show savings-specific information**

Transaction Class Methods:

- **displayTransaction(): Show transaction details**
- **Getter methods: getAmount(), getType(), getDate()**

Customer Class Methods:

- **addAccount(Account* account): Add an account to customer**
- **getTotalBalance(): Calculate total balance across all accounts**
- **displayCustomerInfo(): Show customer and account information**
- **findAccount(int accountNum): Find account by number**

1.4 Identify Inheritance Relationships

The system demonstrates clear inheritance hierarchies:

1. **Account Hierarchy:**
    - **Account (Base class)**
        - **SavingsAccount (Derived class)**
        - **BusinessAccount (Derived class)**
2. **Transaction Hierarchy:**
    - **Transaction (Base class)**
        - **RecurringTransaction (Derived class)**

Inheritance Benefits:

- **Code Reuse: Derived classes inherit common attributes and methods**
- **Polymorphism: Virtual functions allow different behaviors for same interface**
- **Extensibility: New account types can be easily added**
- **Method Overriding: Specialized behavior in derived classes (e.g., SavingsAccount withdrawal restrictions)**

---

2. Class Design Explanation

2.1 Inheritance Implementation

The inheritance design follows the "is-a" relationship principle:

- **A SavingsAccount is-a Account with additional features (interest rate, withdrawal limits)**
- **A BusinessAccount is-a Account with business-specific features (transaction fees, credit limits)**
- **A RecurringTransaction is-a Transaction with recurring properties**

Key Design Decisions:

- **Virtual Destructors: Ensure proper cleanup in polymorphic scenarios**
- **Virtual Methods: Enable runtime polymorphism for withdraw() and displayInfo()**
- **Protected Members: Allow derived classes to access base class data while maintaining encapsulation**

2.2 Operator Overloading Implementation

The system implements several overloaded operators to provide intuitive syntax:

1. **operator+= (Account class):**
2. **Account& operator+=(const Transaction& trans)**
    - **Allows natural syntax: account += transaction**
    - **Automatically applies deposits based on transaction type**
3. **operator== (Account class):**
4. **bool operator==(const Account& other) const**
    - **Compares accounts by balance equality**
    - **Usage: if (account1 == account2)**
5. **operator> (Account class):**
6. **bool operator>(const Account& other) const**
    - **Compares accounts by balance**
    - **Usage: if (savings > checking)**
7. **operator<< (Friend function):**
8. **friend ostream& operator<<(ostream& os, const Account& account)**

- Enables stream output: cout << account
- Provides formatted account summary

2.3 Polymorphism Implementation

The system demonstrates both compile-time and runtime polymorphism:

Runtime Polymorphism:

- **Virtual withdraw() method behaves differently for SavingsAccount (with limits) vs. regular Account**
- **Virtual displayInfo() shows account-specific information**
- **Operations class uses polymorphic behavior when applying interest to savings accounts**

Compile-time Polymorphism:

- **Method overloading in constructors**
- **Operator overloading for intuitive syntax**

---

3. Code Walkthrough - Key Components

3.1 Overloaded Operators in Action

Stream Output Operator (<<):

```
friend ostream& operator<<(ostream& os, const Account& account) {
   os << "Account " << account.accountNumber << " (" << account.ownerName
      << "): $" << fixed << setprecision(2) << account.balance;
   return os;
}
```

This operator provides a clean, formatted output for any account object, making debugging and user interaction more intuitive.

Addition Assignment Operator (+=):

```
Account& operator+=(const Transaction& trans) {
   if (trans.getType() == "Deposit") {
```

```
        deposit(trans.getAmount());
    }
    return *this;
}
```

This allows transactions to be applied to accounts using natural mathematical syntax.

3.2 Virtual Function Override Example

SavingsAccount Withdrawal Override:

```
bool withdraw(double amount) override {
    if (withdrawalsThisMonth >= MAX_WITHDRAWALS) {
        cout << "Error: Exceeded monthly withdrawal limit!" << endl;
        return false;
    }
    if (amount > withdrawalLimit) {
        cout << "Error: Withdrawal amount exceeds limit!" << endl;
        return false;
    }

    if (Account::withdraw(amount)) { // Call base class method
        withdrawalsThisMonth++;
        return true;
    }
    return false;
}
```

This demonstrates method overriding where the derived class adds additional validation while still using the base class implementation.

3.3 System Architecture - Operations Class

The Operations class serves as a central management system, demonstrating the Facade Pattern:

- **Provides simplified interface for complex banking operations**
- **Manages all customers and accounts**
- **Handles system-wide operations like monthly interest application**
- **Implements search and statistical functions**

---

4. Test Results and Demonstrations

4.1 Sample Program Output

Account Creation:

Customer created: Alice Johnson (ID: 1)

Regular account created for Alice Johnson (Account #1)

Savings account created for Alice Johnson (Account #2)

Operator Overloading Demonstration:

Adding transaction using += operator:

Deposited $100.00 to account 1

Comparing account balances using == operator:

Alice's checking and Bob's checking have different balances

Alice's savings account has more money than her checking account

Polymorphic Behavior:

Applied monthly interest: $10.42 to savings account 2

Interest applied to 2 savings accounts.

4.2 Error Handling Demonstration

The system properly handles various error conditions:

- **Invalid withdrawal amounts (negative values)**
- **Insufficient funds**
- **Exceeding savings account withdrawal limits**
- **Non-existent account operations**

4.3 System Statistics

The final test run demonstrates the system's comprehensive functionality:

=== BANKING SYSTEM SUMMARY ===

Total Customers: 3

Total Accounts: 5

Regular Accounts: 2

Savings Accounts: 2

Total System Balance: $11,025.87

---

5. LLM Usage Documentation

5.1 How LLM Was Used

I utilized Claude (Anthropic's LLM) to assist with this assignment in the following ways:

Initial Planning:

- **Prompt: "I need to design a bank account management system in C++ demonstrating inheritance and operator overloading. What classes should I include?"**
- **Response: Suggested SavingsAccount as derived class, Customer management, and operators like <<, +=, ==, >.**

Code Structure:

- **Prompt: "How should I implement virtual functions in my Account base class?"**
- **Response: Guidance on making withdraw() and displayInfo() virtual for polymorphism.**

5.2 Original Implementation

All code was written independently. The LLM helped with C++ syntax understanding and design patterns, but all implementations, test cases, and system architecture were created by myself.

---

6. Conclusion

This Bank Account Management System successfully demonstrates key object-oriented programming concepts:

- **Inheritance: Clear hierarchical relationships with proper method overriding and virtual functions**
- **Polymorphism: Runtime behavior varies based on object type, enabling flexible code**
- **Operator Overloading: Intuitive syntax for common operations, improving code readability**
- **Encapsulation: Proper data hiding with controlled access through public interfaces**

The implementation showcases how OOP principles create maintainable, extensible software systems. The use of virtual functions, inheritance hierarchies, and operator overloading demonstrates mastery of advanced C++ concepts while solving practical business problems.