OBJECT-ORIENTED ANALYSIS (OOA) – BANK ACCOUNT MANAGEMENT SYSTEM

Step 1: Identify Objects (Nouns)

Từ yêu cầu thực tế và kịch bản, ta xác định được các đối tượng chính:

- **Customer:** khách hàng của ngân hàng.
- **Account**: tài khoản cơ bản.
- **SavingsAccount**: tài khoản tiết kiệm (kế thừa từ Account).
- **Transaction**: giao dịch (nạp, rút, chuyển khoản, tính lãi).

Step 2: Identify Attributes (Descriptive Nouns)

Xác định các thuộc tính quan trọng cho từng đối tượng:

- **Customer**
  - id: mã khách hàng (2 chữ số).
  - name: tên khách hàng.
  - accounts: danh sách tài khoản (vector<Account*>).
- **Account**
  - accountNumber: số tài khoản (duy nhất).
  - balance: số dư hiện tại.
  - ownerName: tên chủ tài khoản.
  - transactionHistory: lịch sử giao dịch (vector).
- **SavingsAccount (extends Account)**
  - interestRate: lãi suất hàng tháng (%).
- **Transaction**
  - date: ngày giao dịch.
  - amount: số tiền.
  - type: loại giao dịch (Deposit, Withdrawal, Transfer In, Transfer Out, Interest).
  - description: ghi chú (nguồn tiền, lý do…).

Step 3: Identify Methods (Verbs / Behaviors)

Xác định hành động mà mỗi đối tượng có thể thực hiện:

- **Customer**
  - addAccount(Account*): thêm tài khoản mới cho khách hàng.
  - displayCustomerInfo(): in thông tin khách hàng và các tài khoản.
- **Account**
  - deposit(double, string): nạp tiền.
  - withdraw(double, string): rút tiền.
  - transfer(Account&, double, string): chuyển tiền sang tài khoản khác.
  - addTransaction(Transaction): thêm một giao dịch vào lịch sử.
  - displayAccountInfo(): in chi tiết tài khoản và lịch sử.
- **SavingsAccount (override)**
  - withdraw(double, string): rút tiền nhưng có quy định riêng (ví dụ: hạn chế số lần hoặc phí).

  - applyInterest(string): cộng lãi suất vào số dư và thêm vào lịch sử giao dịch.
- **Transaction**
  - displayTransaction(): in chi tiết giao dịch.

Step 4: Identify Inheritance Relationships

- **Account (base class)**
  - Thuộc tính chung: accountNumber, balance, ownerName, transactionHistory.
  - Hành vi chung: deposit, withdraw, transfer, display.
- **SavingsAccount (derived class)**
  - Kế thừa từ **Account**.
  - Thêm interestRate.
  - Ghi đè (override) phương thức withdraw.
  - Thêm phương thức applyInterest.

Code Walkthrough

The code begins with the **Transaction** class, which stores details such as amount, type, date, and description.

The **Account** class manages account number, owner, balance, and transaction history. It provides methods for deposit(), withdraw(), and transferTo(). Operator overloading is implemented with += (to add a transaction) and == (to compare accounts).

The **SavingsAccount** class inherits from Account and overrides the withdraw() method to enforce withdrawal limits and apply fees. It also includes an applyInterest() method to add interest to the balance.

The **Customer** class stores customer ID, name, and a list of owned accounts. It provides a method printPortfolio() to display account details.

In the main() function, several customers and accounts (Phuc, Loc, Tho) are created. Operations such as deposit, withdrawal, transfer, and interest application are demonstrated, and final portfolios are printed to the console as test output.

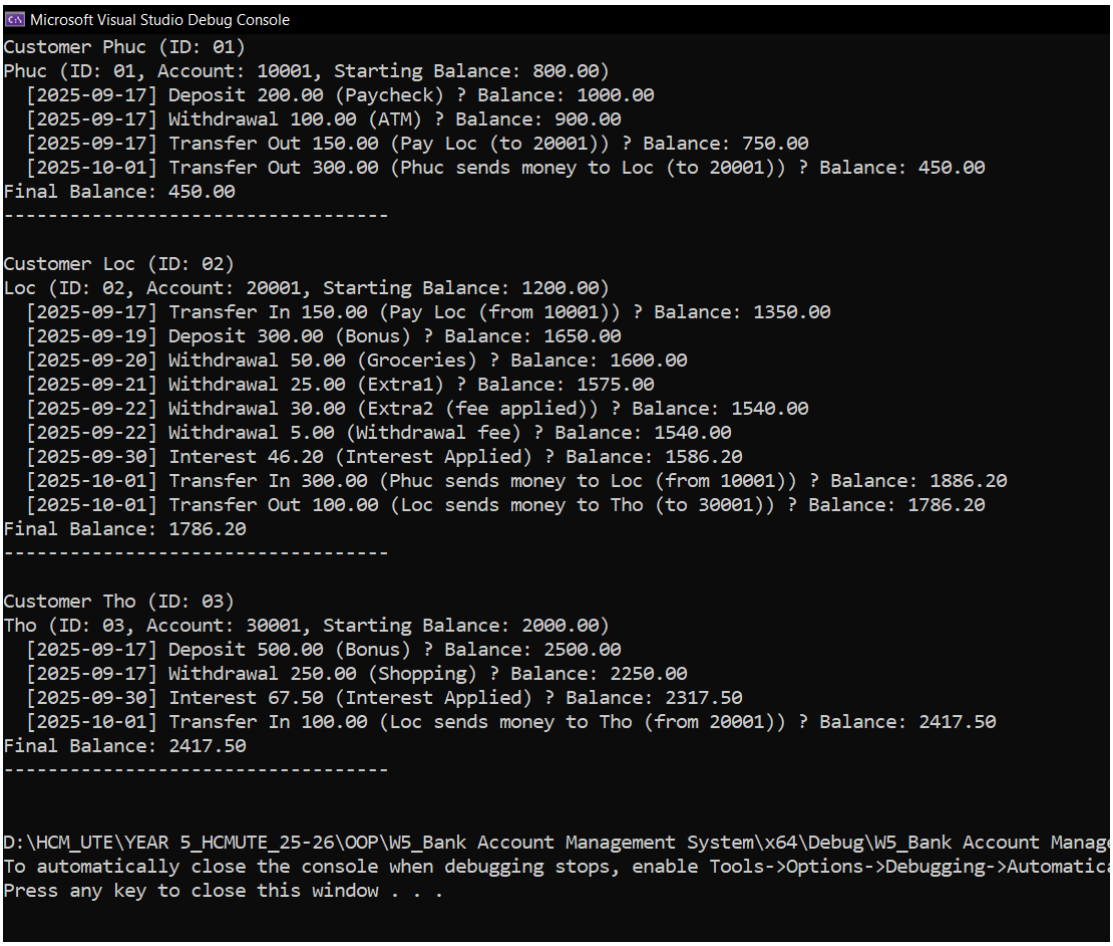
CLASS DESIGN AND OOP EXPLANATION

The system is designed around four main classes: Account, SavingsAccount, Customer, and Transaction. The Account class represents a generic bank account with attributes such as account number, balance, owner name, and transaction history. The SavingsAccount class extends Account through inheritance and adds specialized features like interest rate, withdrawal limits, and fee handling. The Customer class represents a bank customer who owns one or more accounts, while the Transaction class stores details of each operation (amount, date, description).

Object-Oriented Programming principles are applied throughout the design. **Encapsulation** is ensured by making attributes private and exposing operations through public methods (e.g., deposit(), withdraw()). **Inheritance** is demonstrated by SavingsAccount : public Account, allowing the derived class to override the withdraw() method to include specific rules like fees and limits. **Polymorphism** occurs when different account types respond differently to the same method (e.g., withdraw() in regular vs. savings accounts). **Operator overloading** is used to make the system more

intuitive: the += operator is overloaded to add a transaction to an account, automatically updating the balance and history, and the == operator is overloaded to compare two accounts by their balances.

This design not only mirrors real-world banking operations but also demonstrates the use of core OOP concepts in a clear and practical way.



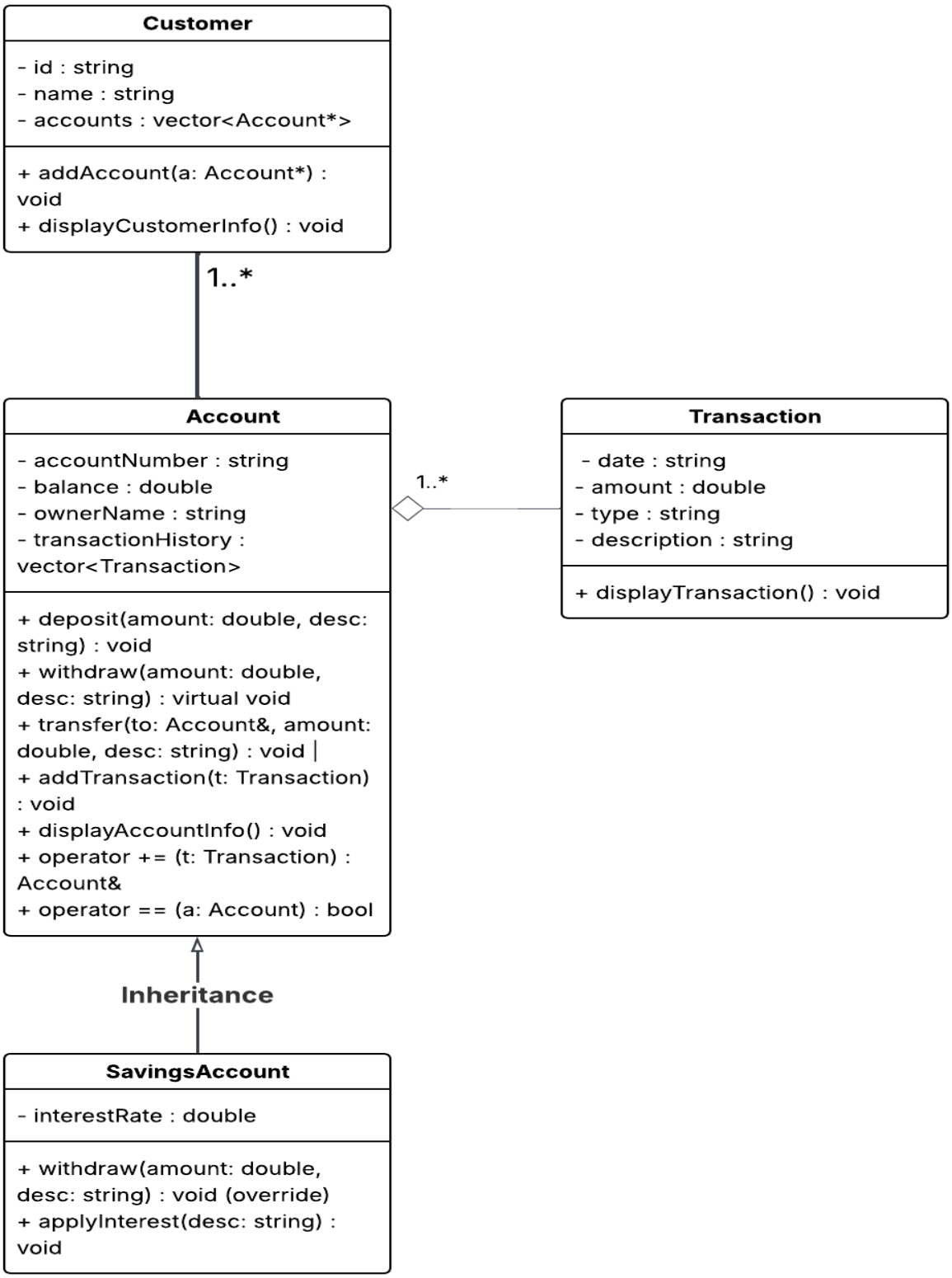
TESTING OUTPUT:

Testing Results

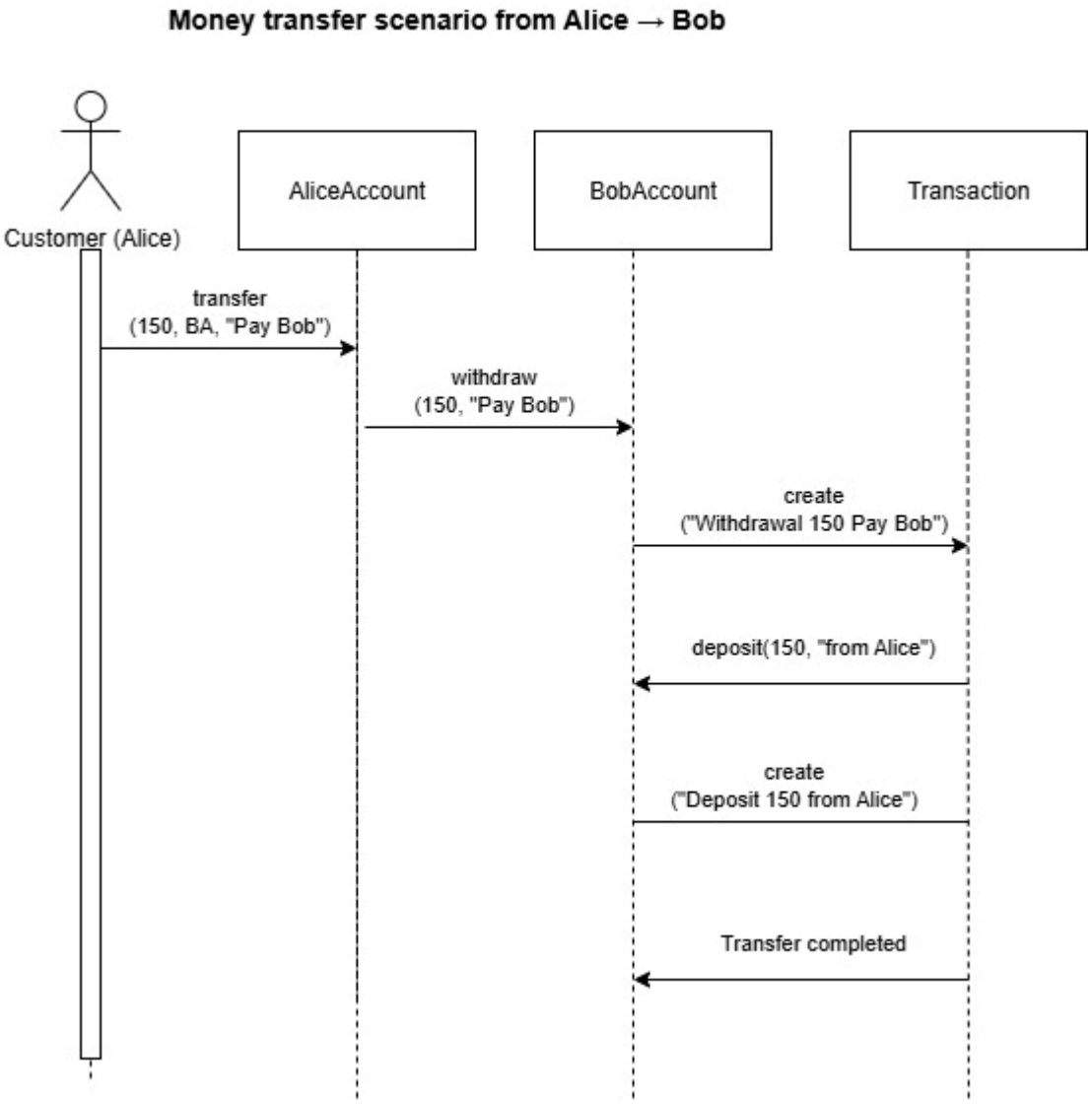
The program was compiled and executed successfully. The output shows:

- Deposits and withdrawals update balances correctly.
- Transfers between users (e.g., Phuc → Loc, Loc → Tho) are recorded in both sender and receiver transaction histories.
- SavingsAccount applies interest at the end of the month, and withdrawal fees are charged after exceeding the monthly limit.
- Final balances are consistent with all transactions.
- Operator overloading (+= for adding transactions, == for comparing accounts) works as expected.

UML Class Diagram Relationships:



UML sequence Diagram :



LLM Usage

During the development of this assignment, I used ChatGPT as a supporting tool for brainstorming and clarification. Specifically, I asked ChatGPT to suggest ideas for operator overloading in the Account class (e.g., overloading += for transactions and == for balance comparison) and to provide explanations of how inheritance and method overriding could be applied to a savings account. I also used it to refine the structure of my test cases and to draft summaries of the console output for documentation. The code implementation and final design decisions were done by myself; ChatGPT was only used as an assistant to generate ideas and improve clarity.