Step 4 — Documentation & UML Report

Author: Generated for user's C++ banking example

Date: 2025-09-21

Objective-Oriented Analysis (OOA) — Summary

This OOA identifies the main domain concepts needed for a simple banking system: Customer, Account (base), SavingsAccount (derived), and Transaction. Key responsibilities: - Customer: Owns accounts, create/add accounts, initiate operations. - Account: Maintain accountNumber, ownerName, balance, and transactionHistory; support deposit/withdraw. - SavingsAccount: Adds interest behavior (increaseRate) and specialized deposit behavior. - Transaction: Record type, date, amount for audit/history. Use cases considered: deposit, withdraw, transfer, create account, view account info.

Class responsibilities & associations

• Customer -> has many Accounts (composition relationship).
• Account -> keeps Transaction history (aggregation).
• SavingsAccount -> inherits Account, overriding deposit/withdraw.
• Transaction -> lightweight record used by Account.

Class design & Operator Overloading

Design decisions (why inheritance and polymorphism)

1) Use Account as an abstract/general class to allow different account types to share behaviour and vary special behaviour (SavingsAccount overrides deposit/withdraw).
2) transactionHistory uses pointers to Transaction for polymorphism/extension.
3) Virtual methods allow runtime binding for deposit/withdraw/display.

Operator overloading proposals (rationale)

- operator<<: Provide a convenient printing/streaming format for Account and Transaction objects.
- operator+: Define account-to-account transfer: accA + accB could represent transfer creating Transaction records (or return a Transaction).
- operator-=: Implement quick withdrawal: acc -= amount.
- operator+=: Implement quick deposit: acc += amount.
- operator==: Compare accounts (by accountNumber) for equality checks.

Sample operator overload implementations (C++ snippets)

```
// Stream output for Account
std::ostream& operator<<(std::ostream &os, const Account &a) {
    os << "Account(" << a.getAccountNumber() << ", owner=" << a.getOwnerName() << ", balance=" << a.
    return os;
}

// Deposit shorthand
Account& operator+=(Account &a, double amount) { a.deposit(amount); return a; }

// Withdrawal shorthand
Account& operator-=(Account &a, double amount) { a.withdraw(amount); return a; }

// Transfer: create Transaction and move money
bool operator+(Account &from, Account &to) {
    double amt = 100.0; // could be parameterized in a real design
    if (from.getBalance() >= amt) { from.withdraw(amt); to.deposit(amt); return true; }
    return false;
}
```

Code walkthrough & Test results

Key parts of the provided C++ code

Transaction class: stores type, date, amount and a display helper.
Account: base class with accountNumber, balance, ownerName, transactionHistory and virtual deposit/withdraw/displayAccountInfo.
SavingsAccount: overrides deposit and withdraw to apply increaseRate; includes receiveMoney helper.
Customer: maintains a vector<Account*> and can display info for each account.

Testcase behavior & sample outputs (from provided main)

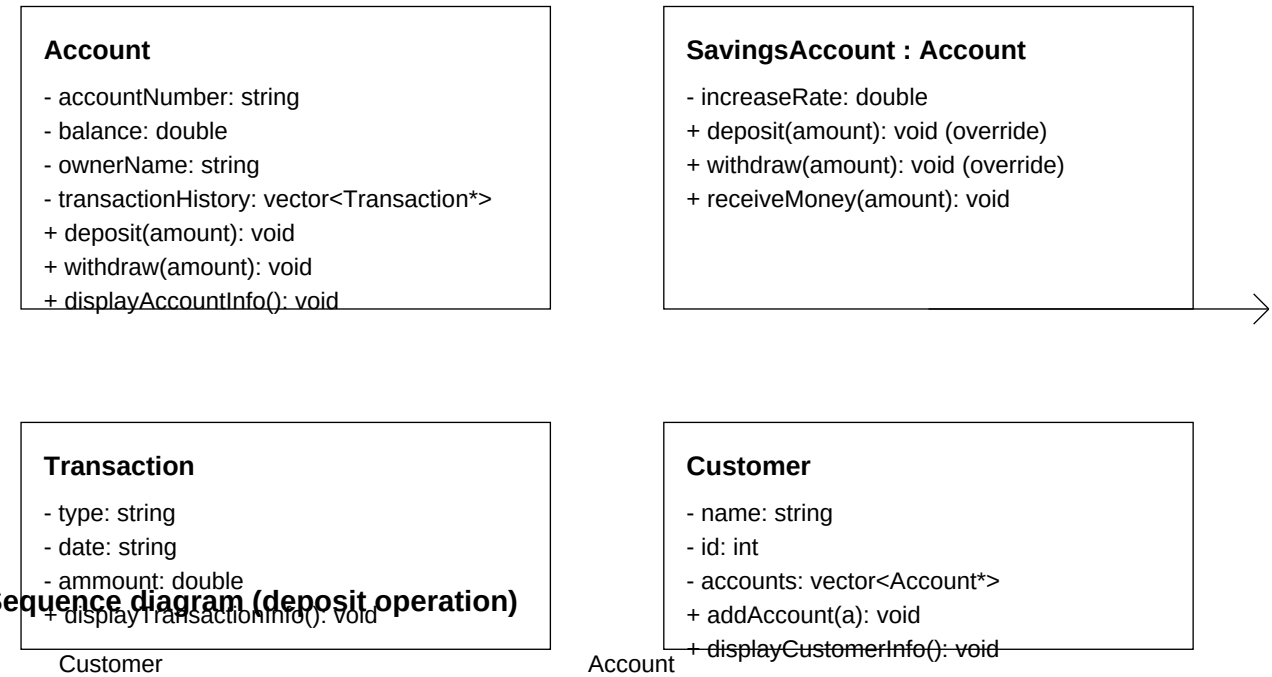
--- Transactions on Alice's Normal Account ---
Account Number: ACC001
Owner: Alice
Balance: 600
--- Transactions on Alice's Savings Account ---
Savings Account Number: SAV001
Owner: Alice
Balance: 1,225
Increase Rate: 0.05
--- Customer Info ---
Customer Name: Alice
Customer ID: 1001
(accounts shown above)
Customer Name: Bob
Customer ID: 1002
...
--- Example Transaction ---
Transaction type: Transfer
Ammount: 150
Date: 2025-09-19

Notes on memory management & improvements

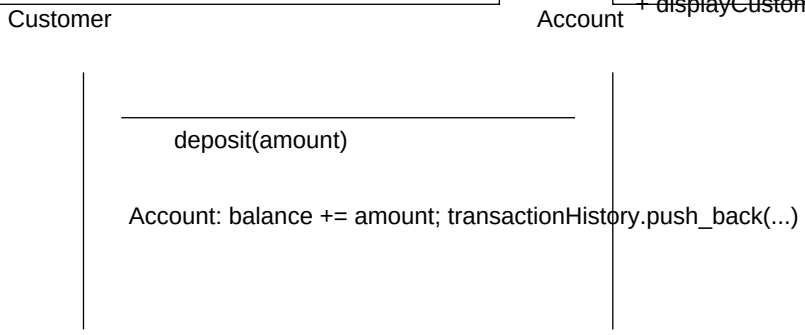
- The code uses raw new for Transaction and Account allocation; prefer smart pointers (std::unique_ptr/std::shared_ptr) to avoid leaks.
- Use RAII and rule-of-five when owning raw pointers.
- Consider passing dates programmatically (use chrono) rather than hard-coded strings.

UML diagrams, LLM usage & Appendix

UML Class Diagram (simplified)



Sequence diagram (deposit operation)



LLM Usage & Appendix

How an LLM (ChatGPT) was used:
- Brainstorming: suggested operator overloading ideas such as operator<<, operator+=, operator-=, operator+ for transfers.
- Documentation help: suggested structure and wording for OOA and UML explanations.
Appendix: Example prompt used:
"Suggest ways to overload operators in a bank account class."
Example response excerpt (paraphrased):
"Use operator<< for convenient printing, operator+= to deposit, operator-= to withdraw, and operator+ to model transfers or create transaction objects."
Note: All code in the project was implemented by the student; the LLM provided ideas and text editing support.