Andrew Taylor

2/28/25

COMS-280-WWW01

# Banking System – Phase 1

## Link

<https://github.com/andrewrtaylor1/BankProj>

## System Design

The Banking System is split into 3 parts: presentation layer, middleware, & a backend database. These are designed to mimic the basic structure of a client, server, & database and have been named as such. The client requests information from the server, and the server grabs the necessary data from the database. If this is banking information, it is only accessible after user authentication. This authentication applies to both bank customers & bank employees. The system module roles are as follows:

Client, responsible for presentation, user input, & beginning user authentication.

Server, responsible for grabbing data from the database according to Client requests. Will block requests without proper authentication. Handles password hashing & encrypts/decrypts sensitive data.

Database, holds the data in memory. This includes users, encryption keys, and bank accounts in linked lists. Sensitive strings were encrypted at the Server step. Has classes & functionality for needed banking functions, like currency handling, account types, KYC, anti-fraud/suspicious transactions handling, interest, etc.

## Implementation Plan

Database:

Currency: Base currency class. Will leverage operator overloading for easy addition/subtraction, multiplication/division, & comparison.

Transaction: utilizing embedded timestamps (from chrono) & currency, alongside information fields, this will represent a single transaction.

Account: Featuring a name, linked list of transactions, & stored account summaries, this is the account class. Abstract, will be split into Checking and Savings with appropriate functionality.

User: stores user information, including hashed password. Will be in a linked list in the database. Employees & Customers are derived classes. Customers will have KYC info.

Server: Uses Listeners to pull request strings from the client, & use events to validate the request & then fulfill it

Client:

Menu: Governing class for each menu (starting screen, user screen, accounts, etc.), contains most of the client logic.

## Challenges & Solutions

One of the challenges for this project is developing with security in mind, while keeping within the scope of our learning. I will do this with hashing for passwords & single key encryption for other needs. The “server” & database will also fail any “malformed” requests.

Storing a variable amount of data in memory, and having it accessible, is another challenge. This will be handled via linked lists. At least one set, transactions, will technically be a list within another list; the transaction records will be under accounts.

To properly have a class to represent currency, it would be very convenient to be able to add the currency classes together quickly. Operator overloading comes in very handy there. Similarly, you could overload comparison operators to have quick comparisons. This solves at least one portion of the overdraft implementation, comparing against the total.

Simulating a database-server-client scheme within a single program is also a strange challenge. C++

## Reflection

Honestly, one of the hardest things for me on this project is simply planning this heavily in advance. When coding for C# and/or hobbyist projects, I usually already have the plan when I start coding or start adding more things as I need them. Obviously, this isn’t a good approach on an actual project, especially if I had collaborators. It’s a hard habit to shake, though.

I am confident I am taking the right direction with the project. I have addressed the requirements I could at the current point, even if all the implementation is definitely not all there. It’s to be secure, handle basic and advanced banking needs, designed to utilize linked lists to store accounts, transaction history, users, and encryption keys.

## Skeleton Code

BankDB.h

#pragma once

#include "List.h"

#include <chrono>

#include <string>

namespace DB

{

//currency base class

class Currency

{

public:

Currency(int i)

{

value = i;

}

virtual ~Currency(){}

virtual const std::string getName() = 0; //currency Name

virtual const std::string getSymbol() = 0;//currency symbol, like $

virtual const std::string formattedValue() = 0; //formatted currency, like $0.00

protected:

int value; //value is stored as int to innately handle

};

//individual transactions

class Transaction

{

public:

Transaction(Currency\* c)

{

value = c;

}

virtual ~Transaction(){}

const std::chrono::system\_clock::time\_point timestamp = std::chrono::system\_clock::now(); //time, to resolve conflicts + sorting

Currency\* value; //the actual value of the transaction

std::string name = "Transaction"; //default transaction name is Transaction; should be changed

};

//Bank Accounts (savings, checking) base class

class Account

{

public:

Account(Transaction t){}

virtual ~Account(){}

LinkedList transactions;

virtual void SpecialFunctions() = 0; //will run special functions in derived classes

};

//base class, takes a string

class User

{

public:

User(std::string s, std::string pass)

{

name = s;

password = pass;

}

virtual ~User() {}

std::string name;

std::string password;

};

class SpecialFunction

{

public:

SpecialFunction() {}

~SpecialFunction() {}

};

class Database

{

public:

Database() {}

~Database() {}

LinkedList users;

LinkedList accounts;

LinkedList encryptKeys;

};

//savings account

class Saving : public Account

{

public:

Saving(Transaction t) : Account(t) {}

~Saving() {}

void SpecialFunctions();

};

//Checking account

class Checking : public Account

{

public:

Checking(Transaction t) : Account(t) {}

~Checking() {}

void SpecialFunctions();

};

//purchases (like debit)

class Purchase : public Transaction

{

Purchase(Currency\* c) : Transaction(c) {}

~Purchase() {}

};

//transfer between accounts transaction

class Transfer : public Transaction

{

Transfer(Currency\* c) : Transaction(c) {}

~Transfer() {}

};

//transaction for deposits

class Deposit : public Transaction

{

Deposit(Currency\* c) : Transaction(c) {}

~Deposit() {}

};

//transaction for overdraft/interest

class BankFunction : public Transaction

{

BankFunction(Currency\* c) : Transaction(c) {}

~BankFunction() {}

};

//customer; has accounts associated via IDs in Linked List (primary checking as first id)

class Customer : public User

{

public:

Customer(std::string s, std::string pass, int i) : User(s,pass) {}

~Customer() {

}

LinkedList accountIDs;

};

class Employee : public User

{

public:

Employee(std::string s, std::string pass) : User(s,pass) {}

~Employee() {}

};

class Overdraft : public SpecialFunction

{

public:

Overdraft() {}

~Overdraft() {}

};

class Interest : public SpecialFunction

{

public:

Interest() {}

~Interest() {}

};

class Suspicion : public SpecialFunction

{

public:

Suspicion() {}

~Suspicion() {}

};

class USDollar : public Currency

{

USDollar(int i) :Currency(i) {}

~USDollar() {}

//returns the name of the currency. Could be used to ID currency types

const std::string getName()

{

return "US Dollar";

}

//gets the symbol; might not need this but could be useful later

const std::string getSymbol()

{

return "$";

}

//returns the value with the proper decimal position & dollar sign, as a string

const std::string formattedValue()

{

std::string s = std::to\_string(value);

if (s.front() == '-')

{

s.erase(0, 1);

if (s.length() < 3) s = "00" + s;

s = "$" + s;

s.insert(s.length() - 2, ".");

s = "-" + s;

}

else

{

if (s.length() < 3) s = "00" + s;

s = "$" + s;

s.insert(s.length() - 2, ".");

}

return s;

}

//allows addition; subtraction not needed, just use negative ints instead

USDollar operator+(const USDollar&)

{

}

//multiplication; will be used to get interest (IE: Dollar\*0.01)

USDollar operator\*(const USDollar&)

{

}

//comparison operators

USDollar operator==(const USDollar&)

{

}

USDollar operator>(const USDollar&)

{

}

USDollar operator<(const USDollar&)

{

}

};

}

BankServer.h

#pragma once

namespace Serv

{

//server converts strings into events, which communicate with the database

class Event

{

public:

Event() {}

~Event() {}

};

//contains listener, uses that to make Events

class Server

{

public:

Server(){}

~Server(){}

};

//recieves strings

class Listener

{

public:

Listener(){}

~Listener(){}

};

class SrvLogon : public Event

{

public:

SrvLogon() {}

~SrvLogon() {}

};

class SrvChangePassword : public Event

{

public:

SrvChangePassword() {}

~SrvChangePassword() {}

};

class SrvAccountTransfer : public Event

{

public:

SrvAccountTransfer() {}

~SrvAccountTransfer() {}

};

class SrvAccountChange : public Event

{

public:

SrvAccountChange() {}

~SrvAccountChange() {}

};

class SrvAccountCreate : public Event

{

public:

SrvAccountCreate() {}

~SrvAccountCreate() {}

};

class SrvAccountHistory : public Event

{

public:

SrvAccountHistory() {}

~SrvAccountHistory() {}

};

class SrvUserCreate : public Event

{

public:

SrvUserCreate() {}

~SrvUserCreate() {}

};

}

BankClient.h

#pragma once

#include <string>

namespace Client

{

//base Menu class, will display and handle logic for each action

class Menu

{

public:

Menu(){}

virtual ~Menu(){}

virtual void logic() = 0;

};

//specific menu classes

class MnuStart : public Menu

{

public:

MnuStart() {}

~MnuStart() {

}

void logic() override;

};

class MnuCustomerLogin : public Menu

{

public:

MnuCustomerLogin() {}

~MnuCustomerLogin() {

}

void logic() override;

std::string login;

};

class MnuEmployeeLogin : public Menu

{

public:

MnuEmployeeLogin() {}

~MnuEmployeeLogin() {

}

void logic() override;

std::string login;

};

class MnuCustomerStart : public Menu

{

public:

MnuCustomerStart() {}

~MnuCustomerStart() {

}

void logic() override;

std::string user;

};

class MnuEmployeeStart : public Menu

{

public:

MnuEmployeeStart() {}

~MnuEmployeeStart() {

}

void logic() override;

std::string user;

};

class MnuEmployeeCreation : public Menu

{

public:

MnuEmployeeCreation() {}

~MnuEmployeeCreation() {

}

void logic() override;

};

class MnuCustomerCreation : public Menu

{

public:

MnuCustomerCreation() {}

~MnuCustomerCreation() {

}

void logic() override;

};

class MnuTransferBetweenAccounts : public Menu

{

public:

MnuTransferBetweenAccounts() {}

~MnuTransferBetweenAccounts() {

}

void logic() override;

std::string account1;

std::string account2;

};

class MnuDeposit : public Menu

{

public:

MnuDeposit() {}

~MnuDeposit() {

}

void logic() override;

std::string account;

};

class MnuGetAccountHistory : public Menu

{

public:

MnuGetAccountHistory() {}

~MnuGetAccountHistory() {

}

void logic() override;

std::string account;

};

//governing Menu class

class Client

{

public:

Client() {

currentMenu = new MnuStart;

}

~Client() {}

Menu\* currentMenu;

};

}

BankClient.ccp

#include "BankClient.h"

#include <iostream>

void Client::MnuStart::logic() //landing page

{

std::cout << "menu here";

}

void Client::MnuCustomerLogin::logic() //login for customers

{

std::cout << "menu here";

}

void Client::MnuEmployeeLogin::logic() //login for employees

{

std::cout << "menu here";

}

void Client::MnuCustomerStart::logic() //customer overview

{

std::cout << "menu here";

}

void Client::MnuEmployeeStart::logic() //employee overview

{

std::cout << "menu here";

}

void Client::MnuEmployeeCreation::logic() //page to make a new employee login

{

std::cout << "menu here";

}

void Client::MnuCustomerCreation::logic() //page to make a new customer login

{

std::cout << "menu here";

}

void Client::MnuTransferBetweenAccounts::logic() //move money between accounts

{

std::cout << "menu here";

}

void Client::MnuDeposit::logic() //employee despositing cash; no self service/ATM yet

{

std::cout << "menu here";

}

void Client::MnuGetAccountHistory::logic() //view the transaction history

{

std::cout << "menu here";

}

BankDP.cpp

#include "BankDB.h"

void DB::Saving::SpecialFunctions() //handles savings (interest)

{

}

void DB::Checking::SpecialFunctions() //handles checking (overdraft, suspicious charges)

{

}

Encrypt.h

#pragma once

namespace Encrypt

{

//Forward Declarations

//Password Hashing

class Hasher;

//Base

class Cryptography;

//Derived

class SingleKeyHandler;

//meant to hash passwords & compare hashes

class Hasher

{

};

//base Cryptography class; virtual decrypt & encrypt functions, in case different encryption is needed in the future

class Cryptography

{

};

//Single key encryption, encrypts in and out on one key

class SingleKeyHandler : public Cryptography

{

};

}

List.h

#pragma once

//Forward Declarations

//Base Types

class LinkedList;

class Node;

//Derived Types

class ExternalNode;

class InternalNode;

//Doubly linked list

class LinkedList

{

};

//base node definition

class Node

{

};

//head and tail nodes

class ExternalNode : public Node

{

};

//internal nodes, just contains data & pointers

class InternalNode : public Node

{

};