Andrew Taylor

3/14/25

COMS-280-WWW01

# Banking System – Phase 2

## Link

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

## Class Hierarchy

While not entirely literal, the graphic below gives a basic idea of the general structure of how accounts are established & accessed. The Account abstract class has 4 derived classes; instead of building Certificate of Deposit & Money Market off of Savings & Checking respectively, I figured it would be more versatile to have all classes derived from the abstract class. Similarly, users are split into Customers & Employees. There is a distinction between the two in terms of use however: Accounts are stored together using polymorphism (List of Accounts) while the users are split into distinct lists.

Users are linked to Accounts via account IDs, a string present within the Account. Customers have Accounts associated with them via a list of IDs. Employees do not need ownership of accounts, so they instead just ignore ownership & look up the account directly.

### Account

Abstract class, defines the default constructor & a moderate amount of full virtual methods, alongside members. It has:

* Transactions: a LinkedList of all Transaction in order.
* ID: string identifier for the account.
* Balance, Available, InterestSoFar: US Dollar amounts for their namesakes. Compound Interest can be accrued at different payout rates.
* Double APY: annual percentage of interest.
* Int interestType, payoutRate: instructions for how to collect interest, and how to pay it out
* Compare: gets ID for use in comparisons.
* LastPayout,LastInterest: Timestamps to know when to increase/pay out interest
* Preview, transaction History: display functions for use in the presentation layer
* updateBalance, makes sure Balance & Available balance are filled out from transactions.
* Virtual functions: specialFunction, will handle any operations that won’t be triggered by users. deposit, deposits money into the account. sendTransfer, creates transfer amount & returns dollar amount. recieveTransfer, takes a dollar amount & processes that as a transfer. processTransaction, does as it says; by using this instead of operating on the transactions list as needed, this will allow future validation, like marking suspicious transactions.

#### Savings

Implements all virtual functions, will use interest, no new fields/members.

#### Checking

Implements all virtual functions, will not use interest, no new fields/members.

#### Certificate of Deposit

Implements all virtual functions, will use interest, has a new member to track CoD term.

#### Money Market

Implements all virtual functions, will use interest, no new fields/members.

### Transaction

Abstract class, holds a dollar value & information about the transaction (name, origin, suspicion, pending).

#### Purchase

Represents purchases, requires transaction name & origin.

#### Transfer

Represents transfers, needs account ID as origin.

#### Deposit

Deposit, has default origin of bank, can take other origins (for things like ATM, etc.)

#### Bank Function

Handles all other functions; interest payout, overdraft fee/protection, etc.

## Demonstration

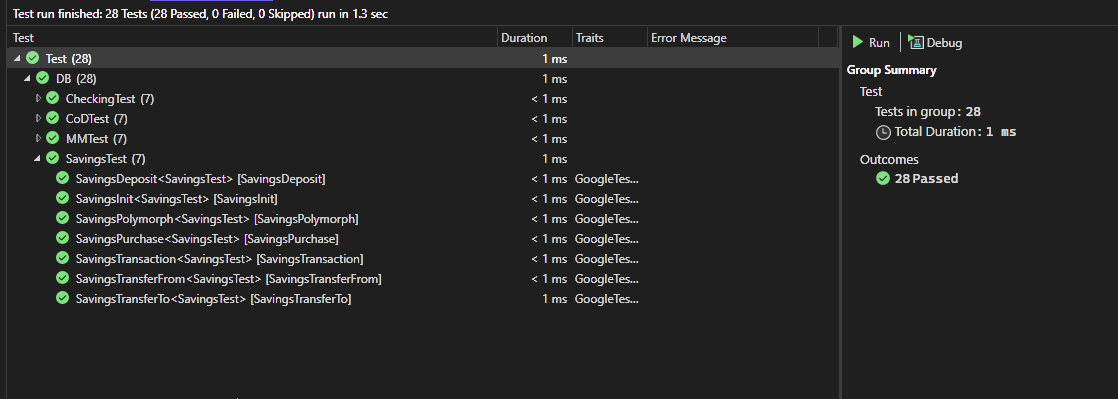
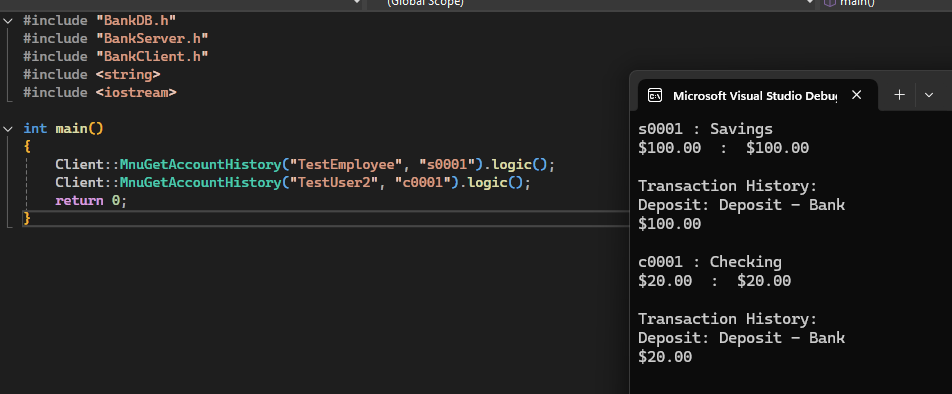


Figure 1: Transactions & Accounts are in a linked list of the abstract type. Savings & checking, and specific transaction types, both successfully execute their overridden virtual functions.

Figure 2: All account types have 7 unit tests currently. They are as follows: init, normal initialization. Polymorph, testing accessing functions/members polymorphically (getType, returns the type as a string.) Transaction, testing the generic transaction handling class. Deposit, testing the deposit transaction type & function. Purchase, testing the purchase function & transaction type. Transfer from, testing sending out a function. TransferTo, testing sending & receiving a function.

## Challenges & Solutions

The greatest challenge for accessing things polymorphically is not “slicing” any classes down to their base class. I solved this problem by extensively using shared\_ptr, a smart (shared) pointer implementation in C++ standard (specifically in the memory library). Important polymorphic functionality is accessed via a linked list using these smart pointers or rarely put into normal pointer form temporarily.

## Reflection

One of the most important parts of this is, essentially, me laying in the bed I’ve made for myself. I had to work ahead of our current goals due to my data design, but it has reaped major rewards. This will make the next phases much easier to implement. This is also my first time using unit tests, and I actually like those systems way more than I expected. The integration of Google Test unit tests into Visual Studio is particularly nice.

Now that I am using pointers extensively, I am surprised at how easy polymorphism is to use. Pointers are a bit annoying, but having this dynamic behavior is very worth it. I could have definitely also taken further use of it by making further abstract classes for Account, for example, but I feel like keeping the implementation simple has value as well.

## Relevant Code

DB.h

//Forward declarations

class Database;

/// <summary>

/// Currency base class

/// </summary>

class Currency

{

public:

Currency(int i = 0) //default/int constructor; this isn't human facing so we just need to pass the raw value

{

value = i;

}

virtual ~Currency(){} //destructor

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

//allows string comparison function

int compare(std::string s)

{

return formattedValue().compare(s); //compares formatted value string

}

protected:

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

};

/// <summary>

/// Currency used by the bank, United States Dollars

/// </summary>

class USDollar : public Currency

{

public:

USDollar(int i = 1) : Currency(i) {} //default/int constructor; this isn't human facing so we just need to pass the raw value

USDollar(double d) : Currency((int)(d \* 100)) {} //double constructor; human-facing systems & interest calcs will make doubles

~USDollar() {} //destructor

//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); //converts value to a string

//is it negative

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

{

s.erase(0, 1); //get rid of negative

if (s.length() < 3) s = "00" + s; //adds 2 0's to keep formatting right

s = "$" + s; //add dollar sign

s.insert(s.length() - 2, "."); //add decimal point

s = "-" + s; //add negative back

}

else

{

if (s.length() < 3) s = "00" + s; //adds 2 0's to keep formatting right

s = "$" + s; //add dollar sign

s.insert(s.length() - 2, "."); //add decimal point

}

return s; //return string

}

//operators & certain other functions do need to be implemented in the derived class

//Gets Percentage; as the stored int is real value \* 100 already, we do have to cast to int

USDollar GetPercentage(double d)

{

return USDollar(int(value \* (d / 100)));

}

//positive operator

USDollar operator+() const

{

return USDollar(value);

}

//negative operator

USDollar operator-() const

{

return USDollar(-value);

}

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

USDollar operator+(const USDollar& add) const

{

return USDollar(value + add.value);

}

//comparison operators

//equals operator overload

bool operator==(const USDollar& comp) const

{

return value == comp.value;

}

//not equal operator overload

bool operator!=(const USDollar& comp) const

{

return value != comp.value;

}

//greater than operator overload

bool operator>(const USDollar& comp) const

{

return value > comp.value;

}

//less than operator overload

bool operator<(const USDollar& comp) const

{

return value < comp.value;

}

//greater than or equal overload

bool operator>=(const USDollar& comp) const

{

return value >= comp.value;

}

//less than or equal overload

bool operator<=(const USDollar& comp) const

{

return value <= comp.value;

}

};

/// <summary>

/// individual transactions

/// </summary>

class Transaction

{

public:

Transaction(USDollar c) //constructor; don't want a default constructor cause we always want a dollar amount

{

Val = c;

};

virtual ~Transaction(){} //destructor

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

USDollar Val; //the actual value of the transaction

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

std::string Origin = "Bank"; //default Origin is Bank; will need to be changed

bool Pending = false; //transaction isn't finalized; initalized as false

bool Suspicious = false; //marks transaction as suspicious

virtual std::string TransactionType() = 0;

int compare(std::string s) //lets Compare work on this class; uses all strings available

{

return (Name + Origin + TransactionType() + Val.formattedValue()).compare(s);

}

};

/// <summary>

/// purchases (like debit)

/// </summary>

class Purchase : public Transaction

{

public:

Purchase(USDollar c, std::string n, std::string o) : Transaction(c) {

Name = n;

Origin = o;

}

~Purchase() {}

std::string TransactionType() { return "Purchase"; }

};

/// <summary>

/// transfer between accounts transaction

/// </summary>

class Transfer : public Transaction

{

public:

Transfer(USDollar c, std::string n) : Transaction(c) {

Name = n;

}

~Transfer() {}

std::string TransactionType() { return "Transfer"; }

};

/// <summary>

/// transaction for deposits

/// </summary>

class Deposit : public Transaction

{

public:

Deposit(USDollar c, std::string o = "Bank") : Transaction(c) {

Name = "Deposit";

Origin = o;

}

~Deposit() {}

std::string TransactionType() { return "Deposit"; }

};

/// <summary>

/// transaction for overdraft/interest/other

/// </summary>

class BankFunction : public Transaction

{

public:

BankFunction(USDollar c, std::string n = "Bank Function") : Transaction(c) {

Name = n;

}

~BankFunction() {}

std::string TransactionType() { return "Bank Function"; }

};

/// <summary>

/// Bank Accounts base class

/// </summary>

class Account

{

public:

Account(Transaction\* t, std::string id) {

Transactions = LinkedList<Transaction>(std::shared\_ptr<Transaction>(t)); //construct Transaction list

updateBalance(); //get the first balance

ID = id; //gets the name; we always want a unique name, 0000 would be an error/placeholder

}

virtual ~Account(){}

LinkedList<Transaction> Transactions; //transaction history!

std::string ID = "0000"; //identifier

USDollar balance; //total balance; updated when transactions gets changed

USDollar available; //total available; in theory, it is total balance - account minimum & certain charges

double APY = 0; //interest rate (can always be expressed as APY, it's just that simple doesn't compound each year)

int interestType = 0; //0: None, 1: Simple, 2: Compound Yearly, 3: Compound Monthly, 4: Compound Daily

int payoutRate = 0; //0: Yearly/None, 1: Every 6 months, 2: monthly, 3: daily

USDollar interestSoFar; //interest accrued so far. This is needed both for compounding & also compound that doesn't pay out at the compound rate

//time members; will just go unused when interest is disabled

//last time paid out; compared against for current payout. default is now(), whenever it is initialized.

std::chrono::system\_clock::time\_point LastPayout = std::chrono::system\_clock::now();

//last time interest came in; compared against for interest. default is now(), whenever it is initialized.

std::chrono::system\_clock::time\_point LastInterest = std::chrono::system\_clock::now();

void updateBalance() //updates balance & available

{

USDollar b(0); //balance

USDollar a(0); //available

//for each member of the transactions list

for (int i = 1; i <= Transactions.getCount(); i++)

{

//get the transaction as a pointer

Transaction\* t = Transactions.get(i).get();

//check if transaction isn't pending, add to available

if (!(t->Pending)) a = a + t->Val;

b = b + t->Val; //add to balance

//fill the values

balance = b;

available = a;

}

}

int compare(std::string s) const //lets Compare work on this class; gets the ID

{

return ID.compare(s);

}

//function for displaying the account at a glance

std::string preview()

{

std::string s = "";

s.append(ID + " : " + this->getType()+ "\n");

s.append(available.formattedValue() + " : " + balance.formattedValue() + "\n\n");

return s;

}

//displays transaction history

std::string transactionHistory()

{

std::string s = "Transaction History:\n";

for (int i = Transactions.getCount(); i > 0; i--)

{

//grabs transaction as pointer (shared ptr lets me make & destroy as many as I want)

std::shared\_ptr<Transaction> t = Transactions.get(i);

s.append(t->TransactionType() + ": " + t->Name + " - " + t->Origin + "\n"); //type, name, and origin

s.append(t->Val.formattedValue()+"\n\n"); //value display (money gained/lost)

}

return s;

}

virtual bool deposit(double d) = 0; //deposit dollar amount

virtual USDollar sendTransfer(double d) = 0; //creates the transfer dollar amount

virtual bool receiveTransfer(USDollar d, std::string id) = 0; //receive transfer amount

virtual bool purchase(double d, std::string name, std::string origin) = 0; //handles purchases

virtual int processTransaction(std::shared\_ptr<Transaction> t) = 0; //receives a new transaction

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

virtual std::string getType() //returns account type

{

return "Account";

}

};

/// <summary>

/// User base class, takes a name & password

/// </summary>

class User

{

public:

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

{

name = s;

password = pass;

}

virtual ~User() {}

std::string name;

std::string password;

int compare(std::string s) //comparison function

{

return name.compare(s);

}

virtual int transfer(std::shared\_ptr<Database> d, std::string acc1, std::string acc2, double v) = 0; //Transfer by accounts for user; int for return code

virtual int deposit(std::shared\_ptr<Database> d, std::string acc, double v) = 0; //Deposit into account; int for return code

};

/// <summary>

/// Savings account

/// </summary>

class Saving : public Account

{

public:

Saving(Transaction\* t, std::string id) : Account(t, id) {}

~Saving() {}

void specialFunctions() //handles special functions ()

{

}

bool deposit(double d) //deposits money

{

//pretty verbose line here, let's work backwards

//I make a new dollar amount with the double, pass that to the new Transaction, which gets put into a smart pointer, which is then processed (success is 0)

return processTransaction(std::shared\_ptr<Transaction>(new Deposit(USDollar(d))))==0;

}

USDollar sendTransfer(double d) //transfers money

{

//simple for right now, logic for special functionality related to transfers will be here

USDollar dollar = USDollar(d); //makes dollar amount

processTransaction(std::shared\_ptr<Transaction>(new Transfer(-dollar, ID))); //create the transfer

return dollar;

}

bool receiveTransfer(USDollar d, std::string id) //recieves transfered money

{

//transfer recieve, success is 0

return processTransaction(std::shared\_ptr<Transaction>(new Transfer(USDollar(d), id)))==0;

}

bool purchase(double d, std::string name, std::string origin) //handles purchase

{

//purchase success is 0

return processTransaction(std::shared\_ptr<Transaction>(new Purchase(USDollar(d), name, origin)))==0;

}

int processTransaction(std::shared\_ptr<Transaction> t) //underlying method for processing transactions (int return code for what happened to the transaction)

{

//very simple for right now

int i = 1; //failure code is 1

if(Transactions.put(t))

{

i = 0; //success code is 0

updateBalance(); //update the balance/available

}

return i; //return code

}

std::string getType() //returns account type

{

return "Savings";

}

};

/// <summary>

/// Checking account

/// </summary>

class Checking : public Account

{

public:

Checking(Transaction\* t, std::string id) : Account(t, id) {}

~Checking() {}

void specialFunctions() //handles special functions ()

{

}

bool deposit(double d) //deposits money

{

//pretty verbose line here, let's work backwards

//I make a new dollar amount with the double, pass that to the new Transaction, which gets put into a smart pointer, which is then processed (success is 0)

return processTransaction(std::shared\_ptr<Transaction>(new Deposit(USDollar(d)))) == 0;

}

USDollar sendTransfer(double d) //transfers money

{

//simple for right now, logic for special functionality related to transfers will be here

USDollar dollar = USDollar(d); //makes dollar amount

processTransaction(std::shared\_ptr<Transaction>(new Transfer(-dollar, ID))); //create the transfer

return dollar;

}

bool receiveTransfer(USDollar d, std::string id) //recieves transfered money

{

//transfer recieve, success is 0

return processTransaction(std::shared\_ptr<Transaction>(new Transfer(USDollar(d), id))) == 0;

}

bool purchase(double d, std::string name, std::string origin) //handles purchase

{

//purchase success is 0

return processTransaction(std::shared\_ptr<Transaction>(new Purchase(USDollar(d), name, origin))) == 0;

}

int processTransaction(std::shared\_ptr<Transaction> t) //underlying method for processing transactions (int return code for what happened to the transaction)

{

//very simple for right now

int i = 1; //failure code is 1

if (Transactions.put(t))

{

i = 0; //success code is 0

updateBalance(); //update the balance/available

}

return i; //return code

}

std::string getType() //returns account type

{

return "Checking";

}

};

/// <summary>

/// Certificates of Deposit account

/// </summary>

class CertOfDep : public Account

{

public:

CertOfDep(Transaction\* t, std::string id) : Account(t, id) {}

~CertOfDep() {}

std::chrono::system\_clock::time\_point EndOfTerm;

void specialFunctions() //handles special functions (Savings + ensuring term)

{

}

bool deposit(double d) //deposits money

{

//pretty verbose line here, let's work backwards

//I make a new dollar amount with the double, pass that to the new Transaction, which gets put into a smart pointer, which is then processed (success is 0)

return processTransaction(std::shared\_ptr<Transaction>(new Deposit(USDollar(d)))) == 0;

}

USDollar sendTransfer(double d) //transfers money

{

//simple for right now, logic for special functionality related to transfers will be here

USDollar dollar = USDollar(d); //makes dollar amount

processTransaction(std::shared\_ptr<Transaction>(new Transfer(-dollar, ID))); //create the transfer

return dollar;

}

bool receiveTransfer(USDollar d, std::string id) //recieves transfered money

{

//transfer recieve, success is 0

return processTransaction(std::shared\_ptr<Transaction>(new Transfer(USDollar(d), id))) == 0;

}

bool purchase(double d, std::string name, std::string origin) //handles purchase

{

//purchase success is 0

return processTransaction(std::shared\_ptr<Transaction>(new Purchase(USDollar(d), name, origin))) == 0;

}

int processTransaction(std::shared\_ptr<Transaction> t) //underlying method for processing transactions (int return code for what happened to the transaction)

{

//very simple for right now

int i = 1; //failure code is 1

if (Transactions.put(t))

{

i = 0; //success code is 0

updateBalance(); //update the balance/available

}

return i; //return code

}

std::string getType() //returns account type

{

return "Certificate of Deposit";

}

};

/// <summary>

/// Money Market account

/// </summary>

class MoneyMarket : public Account

{

public:

MoneyMarket(Transaction\* t, std::string id) : Account(t, id) {}

~MoneyMarket() {}

void specialFunctions() //handles special functions (has needs of both checkings & savings, and its own needs)

{

}

bool deposit(double d) //deposits money

{

//pretty verbose line here, let's work backwards

//I make a new dollar amount with the double, pass that to the new Transaction, which gets put into a smart pointer, which is then processed (success is 0)

return processTransaction(std::shared\_ptr<Transaction>(new Deposit(USDollar(d)))) == 0;

}

USDollar sendTransfer(double d) //transfers money

{

//simple for right now, logic for special functionality related to transfers will be here

USDollar dollar = USDollar(d); //makes dollar amount

processTransaction(std::shared\_ptr<Transaction>(new Transfer(-dollar, ID))); //create the transfer

return dollar;

}

bool receiveTransfer(USDollar d, std::string id) //recieves transfered money

{

//transfer recieve, success is 0

return processTransaction(std::shared\_ptr<Transaction>(new Transfer(USDollar(d), id))) == 0;

}

bool purchase(double d, std::string name, std::string origin) //handles purchase

{

//purchase success is 0

return processTransaction(std::shared\_ptr<Transaction>(new Purchase(USDollar(d), name, origin))) == 0;

}

int processTransaction(std::shared\_ptr<Transaction> t) //underlying method for processing transactions (int return code for what happened to the transaction)

{

//very simple for right now

int i = 1; //failure code is 1

if (Transactions.put(t))

{

i = 0; //success code is 0

updateBalance(); //update the balance/available

}

return i; //return code

}

std::string getType() //returns account type

{

return "Money Market";

}

};

/// <summary>

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

/// </summary>

class Customer : public User

{

public:

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

~Customer() {}

LinkedList<std::string> AccountIDs; //accounts owned/accessible by user

//Transfer between accounts; int for return code. Customers need to own/have access to account

int transfer(std::shared\_ptr<Database> d, std::string acc1, std::string acc2, double v);

//Customer deposit logic

int deposit(std::shared\_ptr<Database> d, std::string acc, double v);

};

/// <summary>

/// Employee User

/// </summary>

class Employee : public User

{

public:

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

~Employee() {}

//Transfer between accounts; int for return code. Employees don't care about account ownership

int transfer(std::shared\_ptr<Database> d, std::string acc1, std::string acc2, double v);

int deposit(std::shared\_ptr<Database> d, std::string acc, double v);

};

/// <summary>

/// Database class

/// </summary>

class Database

{

public:

Database() {

//placeholders

std::shared\_ptr<Customer> c(new Customer("TestUser", "pass"));

std::shared\_ptr<Customer> c2(new Customer("TestUser2", "pass2"));

std::shared\_ptr<Employee> e(new Employee("TestEmployee", "pass"));

std::shared\_ptr<Employee> e2(new Employee("TestEmployee", "pass2"));

std::shared\_ptr<Account> sav(new Saving(new Deposit(USDollar(10000)), "s0001"));

std::shared\_ptr<Account> chk(new Checking(new Deposit(USDollar(2000)), "c0001"));

std::shared\_ptr<std::string> s(new std::string("s0001"));

std::shared\_ptr<std::string> s2(new std::string("c0001"));

c->AccountIDs.put(s);

c2->AccountIDs.put(s2);

Customers = LinkedList<Customer>(c);

Employees = LinkedList<Employee>(e);

Accounts = LinkedList<Account>(sav);

EncryptionKeys = LinkedList<std::string>();

Accounts.put(chk);

Customers.put(c2);

Employees.put(e2);

//placeholder end

}

~Database() {}

LinkedList<Customer> Customers;

LinkedList<Employee> Employees;

LinkedList<Account> Accounts;

LinkedList<std::string> EncryptionKeys;

};