Bank ATM Machine Simulation Using Multiple Inheritance and Polymorphism in C++

Akram Alhinnawi, Haiyang Wang
Advisor: Abhilasha Tibrewal
Department of Computer Science
University of Bridgeport, Bridgeport, CT

Abstract

ATM is one of the main sources of withdrawal and/or deposit of money nowadays. In this project, a simulation of a location with multiple ATM Machines (say ATM booth outside the banks) is attempted. The ATM Machine must be capable of servicing different types of account holders (customers) depending on account type and transaction type.

A customer can perform four transactions i.e. Withdrawal, Deposit, Transfer and Balance Inquiry. Also, a customer can have two types of Accounts, namely, Personal and Business accounts with four possible subtypes of each. Thus, a three level inheritance model is used. The important technique is to find similar data members among them and use multiple level inheritance to maximize code reuse as well as to use method overriding to implement polymorphism. Multiple inheritance creates a diamond problem, with which, the UML diagram looks like a diamond and the derived class will inherent indirectly multiple copies from the first base class it's protected data members and public methods making them ambiguous, so, they will not get past the compiler; while using overriding technique in polymorphism to use two accessor methods for the hierarchy creates a problem of accessing many Data Types using only these two methods. This project will show the solution of how to solve diamond problem and overriding accessor methods as well as the important components used.

Main Components

1. Bank Class

The central unit to which all ATMs will report. It generates the customers and populates them in the Big customer Queue, keep tracking of statistics about transactions performed in each ATM and generates the customer traffic to do transaction on ATMs. Also, it controls the system using TimingWheel class.

2. ATM Class

Every ATM would include multiple customers waiting in the customer queue of the ATM. A person would join the shortest queue on arrival but might want to change queues if another booth becomes available earlier. A process() method is called to fetch the next customer from the front() of the customer Queue. For Withdraw() method, a check of the ATM machine is being done to make sure it has enough cash as well as the customer has a valid amount of cash in his account. Also, a check on the "From account" in Transaction() method to transfer money from one account into another is done to validate that it has sufficient amount otherwise a decline transaction message will be displayed. Other methods, i.e. Deposit() and BalanceInquiry() is being conducted also by the process() method. A pointer to BankPolicy object is being used to check the constraints and limitations that any bank enforce in its every daily transactions. At this phase, the statistics is being collected through time.

3. BankPolicy Class

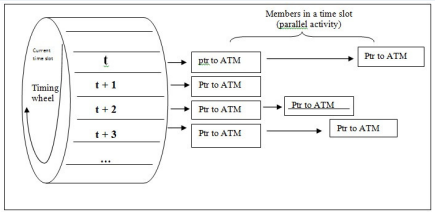
This class gathers all the policy variables enforced in the bank whether it is for customers accounts or ATM. If the bank changes some of its policy, then only the specific variable(s) is/are changed. This will affect all the methods in the Bank as well as ATM Classes.

4. Transactions Class

Only four methods i.e. Withdrawal(), Deposit(), Transfer() and BalanceInquiry() have been used by the ATM Class.

5. Timing_Wheel Class

It has many slots, each of them contains many partitions (as many as the number of ATMs that will become available). A time slot also represents concurrent activity at that point in time. A method, insert(), will insert a partition in the timing wheel with the appropriate delay i.e. the time spent by a customer from the current time. A traffic_generator() method is used to distribute customers fairly to the available ATM where the customer will move to the less queue.



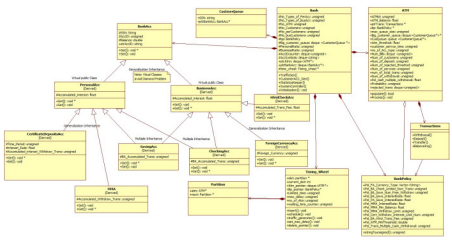
Timing Wheel object and its components in action.

Solving Diamond Problem

Multiple inheritance hierarchies can be complex, which may lead to the situation in which a derived class inherits multiple times from the same indirect base class. This would lead to an issue called Diamond problem in programming. In the presented project, two classes, CheckingAcc and SavingAcc inherits from two parents i.e. PersonalAcc and BusinessACC classes; also, inherits indirectly from the main base class namely, BankAcc. Thus CheckingAcc and SavingAcc inherits two versions of the public and protected data members of BankAcc.

To solve this problem, virtual inheritance is needed. So, a change into virtual base classes has been implemented on the PreronalAcc and BusinessAcc classes. This has corrected the problem by sending a single copy of the data members of the BankAcc class to be inherited by the CheckingAcc and SavingAcc. The following code excerpt illustrate the solution:

```
class BankAcc // Main Base class for the whole hierarchy
{
public: // Get() and Set() accessor methods
protected: // Data members
};
// Virtual Base classes
class PersonalAcc : virtual public BankAcc
{};
class BusinessAcc : virtual public BankAcc
{};
// Multiple inheritance classes
class SavingAcc : public PersonalAcc, public BusinessAcc
{};
class CheckingAcc : public PersonalAcc, public BusinessAcc
{};
```



UML diagram that shows the whole hierarchy as well as all the components of the project.

Solving Two Accessor Methods To Deal with multiple Data Types

When designing an inheritance hierarchy, the best technique to resort to is using Polymorphism to maximize reusing the same code for methods and data members. The purpose of polymorphism is to use the same method with the same signature in multiple classes which can only exist in the hierarchy. The problem stems when trying to use accessor methods that have the same signature i.e. Get() to fetch the values of the data members and/or Set() to assign a new value to the data members of all the classes of the hierarchy. Because these two accessor methods are needed to exist in all the classes of the hierarchy, the problem becomes more clearer. How to deal with various data types, passing them through parameters and returning their various data types values. There are several solution for this problem. One of them is to overload the method, each for a different data type.

Although this could work, but it is not maximizing the reuse of the code, on the first hand, and not utilizing the polymorphism technique that could effectively reduce the code dramatically, in the second hand.

The solution presented in this project is using two techniques. The first one, of which without it the solution could not be implemented, is the polymorphism technique. So, for each accessor method, in each class in the hierarchy, the visibility is public and, of course, to override the inherited method, a *virtual* keyword in C++ is being used. The second one, is using void pointers, which has no data type, to pass the values as parameters and return them. The second technique requires an additional step which is static casting the void pointer's stored value into the needed data type again. The following code excerpts shows this for the Set() method:

```
virtual void Set( void *pValue, string ClassName, string varName)
{
 if (varName=="SSN" && ClassName ==typeid(*this).name())
   SSN = *static_cast<string*>(pValue);
 else if (varName=="AccID" && ClassName ==typeid(*this).name())
   AccID = *static_cast<unsigned*>(pValue);
 else if (varName=="Balance" && ClassName ==typeid(*this).name())
  Balance = *static_cast<float*>(pValue);
  // The rest of the code….
}
```

Calling Set() method:

```
string className = typeid(*ptrBankAcc[No_of_Accounts]).name();
string varName = "SSN";                    //set ssn
string str = SSN[j];
ptrBankAcc[No_of_Accounts]->Set(&SSN[j], className ,varName);
```

Get() method:

```
virtual void* Get(string ClassName, string &varName){
    if (varName=="SSN" && ClassName == typeid(*this).name())
    {
        varName = typeid(SSN).name();
        return &SSN;
    }
    else if (varName=="AccID" && ClassName == typeid(*this).name())
    {
        varName = typeid(AccID).name();
        return &AccID;
    }
    else if (varName=="Balance" && ClassName ==typeid(*this).name())
    {
        varName = typeid(Balance).name();
        return &Balance;
    }
    return NULL;
}
```

Calling Get() method:

```
varName = "Balance";
// Get the balance for the second account
ptr = ToAccObj->Get(typeid(*ToAccObj).name(), varName);
ToAccBalance = *static_cast<float*>(ptr);
```


Parts of screen shots taken upon running the simulation.

CONCLUSION

With this project , we conclude that using virtual inheritance will solve the diamond problem and by utilizing polymorphism along with void pointers we provided a solution to use accessor methods in a multiple in heritance hierarchy to deal with multiple data types.

The project simulated the whole procedure of ATM Banking system, where proper transaction was dealt based on corresponding bank account specifics. Also, it imitated the real world time and customer queue by using timing wheel system, and queue structure . All in all, the project achieved the communication of Bank Customers and ATMs.