# COMS-280 Final Project

## Code Implementation

#include <iostream>

#include <vector>

#include <list>

#include <memory>

#include <stdexcept>

#include <iomanip>

#include <sstream>

using namespace std;

class BankAccount {

protected:

string owner;

double balance;

list<string> transactionHistory;

public:

BankAccount(string name, double initialBalance) : owner(name), balance(initialBalance) {}

virtual ~BankAccount() {}

virtual void deposit(double amount) = 0;

virtual void withdraw(double amount) = 0;

virtual void display() const = 0;

void addTransaction(const string& transaction) {

transactionHistory.push\_back(transaction);

}

void displayTransactionHistory() const {

cout << "Transaction History for " << owner << ":\n";

for (const auto& transaction : transactionHistory) {

cout << transaction << endl;

}

}

double getBalance() const { return balance; }

string getOwner() const { return owner; }

};

class SavingsAccount : public BankAccount {

private:

double interestRate;

public:

SavingsAccount(string name, double balance, double rate) : BankAccount(name, balance), interestRate(rate) {}

void deposit(double amount) override {

balance += amount;

addTransaction("Deposited: $" + formatAmount(amount));

}

void withdraw(double amount) override {

if (amount > balance) throw runtime\_error("Insufficient funds");

balance -= amount;

addTransaction("Withdrawn: $" + formatAmount(amount));

}

void display() const override {

cout << "Savings Account: " << owner << " | Balance: $" << fixed << setprecision(2) << balance << " | Interest Rate: " << interestRate << "%\n";

}

string formatAmount(double amount) const {

ostringstream stream;

stream << fixed << setprecision(2) << amount;

return stream.str();

}

};

class CheckingAccount : public BankAccount {

private:

double overdraftLimit;

public:

CheckingAccount(string name, double balance, double overdraft) : BankAccount(name, balance), overdraftLimit(overdraft) {}

void deposit(double amount) override {

balance += amount;

addTransaction("Deposited: $" + formatAmount(amount));

}

void withdraw(double amount) override {

if (amount > balance + overdraftLimit) throw runtime\_error("Overdraft limit exceeded");

balance -= amount;

addTransaction("Withdrawn: $" + formatAmount(amount));

}

void display() const override {

cout << "Checking Account: " << owner << " | Balance: $" << fixed << setprecision(2) << balance << " | Overdraft Limit: $" << overdraftLimit << "\n";

}

string formatAmount(double amount) const {

ostringstream stream;

stream << fixed << setprecision(2) << amount;

return stream.str();

}

};

template <typename T>

class AccountManager {

private:

vector<unique\_ptr<T>> accounts;

public:

void addAccount(unique\_ptr<T> account) {

accounts.push\_back(move(account));

}

void displayAccounts() const {

for (const auto& acc : accounts) {

acc->display();

}

}

T\* getAccount(string name) {

for (const auto& acc : accounts) {

if (acc->getOwner() == name) {

return acc.get();

}

}

return nullptr;

}

};

template <typename T>

void performBankingOperations(AccountManager<T>& manager) {

string name;

char choice;

double amount;

while (true) {

cout << "\nEnter account owner name (or 'exit' to quit): ";

cin >> name;

if (name == "exit") break;

T\* account = manager.getAccount(name);

if (!account) {

cout << "Account not found.\n";

continue;

}

cout << "\nChoose operation: \nD - Deposit\nW - Withdraw\nS - Show Account\nH - Show Transaction History\nE - Exit\nChoice: ";

cin >> choice;

switch (choice) {

case 'D': case 'd':

cout << "Enter deposit amount: ";

cin >> amount;

account->deposit(amount);

cout << "Deposit successful.\n";

break;

case 'W': case 'w':

cout << "Enter withdrawal amount: ";

cin >> amount;

try {

account->withdraw(amount);

cout << "Withdrawal successful.\n";

}

catch (const exception& e) {

cout << "Error: " << e.what() << endl;

}

break;

case 'S': case 's':

account->display();

break;

case 'H': case 'h':

account->displayTransactionHistory();

break;

case 'E': case 'e':

return;

default:

cout << "Invalid choice.\n";

}

}

}

int main() {

AccountManager<BankAccount> manager;

manager.addAccount(make\_unique<SavingsAccount>("Laurie", 5000, 2.5));

manager.addAccount(make\_unique<CheckingAccount>("Larry", 1000, 500));

manager.addAccount(make\_unique<SavingsAccount>("David", 10000, 2.5));

manager.addAccount(make\_unique<CheckingAccount>("Luis", 2000, 500));

performBankingOperations(manager);

return 0;

}









