# COMS-280 Final Project

## Skeleton Code Document

// Base class (Abstract)

// Defines a blueprint for various types of bank accounts.

// Implements core properties such as owner name and balance.

// Provides pure virtual functions for deposit, withdrawal, and displaying account details.

class BankAccount {

protected:

string owner;

double balance;

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;

double getBalance() const { return balance; }

};

// Derived class - Savings Account

// A specialized bank account with an additional interest rate field.

// Implements deposit and withdrawal operations.

// Ensures that withdrawal does not exceed available balance.

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; }

void withdraw(double amount) override {

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

balance -= amount;

}

void display() const override {

cout << "Savings Account: " << owner << " | Balance: $" << balance << " | Interest Rate: " << interestRate << "%\n";

}

};

// Derived class - Checking Account

// A specialized bank account with overdraft protection.

// Allows withdrawals beyond the balance within a specified overdraft limit.

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; }

void withdraw(double amount) override {

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

balance -= amount;

}

void display() const override {

cout << "Checking Account: " << owner << " | Balance: $" << balance << " | Overdraft Limit: $" << overdraftLimit << "\n";

}

};

// Transaction Node (for Linked List)

// Represents a single transaction stored in a linked list.

struct Transaction {

string type;

double amount;

Transaction(string t, double a) : type(t), amount(a) {}

};

// Transaction List (Linked List)

// Manages a linked list of transactions for an account.

// Provides functionality to add and display transactions.

class TransactionHistory {

private:

list<Transaction> transactions;

public:

void addTransaction(string type, double amount) {

transactions.emplace\_back(type, amount);

}

void displayHistory() const {

cout << "Transaction History:\n";

for (const auto& t : transactions) {

cout << t.type << " of $" << t.amount << "\n";

}

}

};

// Template Class: AccountManager<T>

// A generic account manager that can handle different types of bank accounts.

// Uses a vector of unique\_ptr for efficient memory management.

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();

}

}

};

// Function: performBankingOperations

// Displays all accounts managed by AccountManager.

// Demonstrates error handling for withdrawal exceptions.

template <typename T>

void performBankingOperations(AccountManager<T>& manager) {

try {

manager.displayAccounts();

// Example operations (would be user-driven in a full application)

} catch (const exception& e) {

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

}

}

// Mainc Function

// Entry point of the banking system.

// Creates instances of SavingsAccount and CheckingAccount.

// Adds accounts to AccountManager and performs banking operations.

int main() {

AccountManager<BankAccount> manager;

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

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

performBankingOperations(manager);

return 0;

}