Bank Transactions

**PROGRAM.CS**

using System;

using System.Security.Principal;

//using SplashKitSDK;

public enum MenuOption

{

Withdraw,

Deposit,

Print,

transfer,

Quit,

}

public class Program

{

int account;

private static MenuOption ReadUserOption()

{

int option;

string inputText;

do

{

Console.WriteLine("Choose an option [1-5]: ");

inputText = Console.ReadLine();

option = Convert.ToInt32(inputText);

} while (option < 1 || option > 5);

return (MenuOption)(option - 1);

}

private static void DoWithdraw(Account account)

{

string inputText;

decimal myDecimal;

Console.WriteLine("How much would you like to withdraw?");

do

{

inputText = Console.ReadLine();

myDecimal = Convert.ToDecimal(inputText);

if (myDecimal <= account.\_balance)

break;

}

while (myDecimal <= account.\_balance);

WithdrawTransaction WithdrawTransaction = new WithdrawTransaction(account,

myDecimal);

WithdrawTransaction.Execute();

WithdrawTransaction.Print();

}

private static void DoDeposit(Account account)

{

string inputText;

decimal myDecimal;

Console.WriteLine("How much would you like to deposit?");

inputText = Console.ReadLine();

myDecimal = Convert.ToDecimal(inputText);

DepositTransaction transaction = new DepositTransaction(account, myDecimal);

transaction.Execute();

transaction.Print();

}

private static void DoTransfer(Account FromAccount, Account toAccount, Decimal

Amount)

{

TransferTransaction TransferObject = new TransferTransaction(FromAccount, toAccount, Amount);

TransferObject.Execute();

TransferObject.Print();

}

private static void DoPrint(Account account)

{

account.Print();

}

public static void Main()

{

Account account1 = new Account("Jake's Account", 200000);

// account1.Print();

// account1.Deposit(100);

// account1.Print();

// account1.Withdraw(100);

// account1.Print();

Account account = new Account("Alton's Account", 100000);

// account.Print();

// account.Deposit(500);

// account.Print();

// account.Deposit(100);

// account.Print();

// account.Withdraw(400);

// account.Print();

// account.Withdraw(300);

// account.Print();

String AccountName;

String AccountName2;

Decimal OpeningAmount;

Decimal AmountToBeTransferred;

string inputText;

Console.WriteLine("\* 1 will Withdraw \*");

Console.WriteLine("\* 2 will Deposit \*");

Console.WriteLine("\* 3 will Print Account \*");

Console.WriteLine("\* 4 will Transfer Between Accounts");

Console.WriteLine("\* 5 will Quit \*");

MenuOption userSelection;

userSelection = ReadUserOption();

Console.WriteLine(userSelection);

do

{

userSelection = ReadUserOption();

switch (userSelection)

{

case MenuOption.Withdraw:

DoWithdraw(account);

break;

case MenuOption.Deposit:

DoDeposit(account);

break;

case MenuOption.Print:

DoPrint(account);

break;

case MenuOption.transfer:

Console.WriteLine("Please Amount to be tranferred");

inputText = Console.ReadLine();

AmountToBeTransferred = Convert.ToDecimal(inputText);

Console.WriteLine("Please Enter Name of the Withdrawl Account");

AccountName = Console.ReadLine();

Console.WriteLine("Please Enter Name of the Deposit Account");

AccountName2 = Console.ReadLine();

DoTransfer(account, account1, AmountToBeTransferred);

break;

case MenuOption.Quit:

Console.WriteLine("See you later!");

break;

}

} while (userSelection != MenuOption.Quit);

}

}

**ACCOUNT.CS**

using System;

public class Account

{

public decimal \_balance;

public string \_name;

public Account(string name, decimal startingBalance)

{

\_name = name;

\_balance = startingBalance;

}

public bool Deposit(decimal amountToAdd)

{

if (amountToAdd > 0)

{

\_balance += amountToAdd;

return true;

}

return false;

//\_balance = \_balance + amountToAdd;

}

public bool Withdraw(decimal amountToDeduct)

{

if (amountToDeduct > 0)

{

\_balance = \_balance - amountToDeduct;

return true;

}

return false;

}

public string Name

{

get { return \_name; }

}

public decimal Amount

{

get { return \_balance; }

}

public void Print()

{

Console.WriteLine("This is " + \_name + " and your balance is " + \_balance);

}

public static implicit operator string(Account v)

{

throw new NotImplementedException();

}

}

**WITHDRAWTRANSACTION.CS**

using System;

public class WithdrawTransaction

{

private Account \_account;

private decimal \_amount;

private bool \_executed = false;

private bool \_success = false;

private bool \_reversed = false;

private bool \_fail = false;

public bool Executed

{

get

{

return \_executed;

}

}

public bool Success

{

get

{

return \_success;

}

}

public bool Reversed

{

get

{

return \_reversed;

}

}

public WithdrawTransaction(Account account, decimal amount)

{

\_account = account;

\_amount = amount;

}

public void Execute()

{

if (\_executed)

{

throw new Exception("Cannot execute this transaction as it has already been executed.");

}

\_executed = true;

\_success = \_account.Withdraw(\_amount);

}

public void Rollback()

{

if (\_executed == false)

{

throw new Exception("Transaction has not been executed.");

}

if (\_reversed == true)

{

throw new Exception("Transaction has not been reversed.");

}

\_reversed = true;

\_success = true;

\_success = \_account.Deposit(\_amount);

// {

// return \_reversed;

// }

// if ( \_executed == false && \_reversed == true )

// {

// \_fail = true;

// }

}

public void Print()

{

if (\_executed == true)

{

Console.WriteLine("The withdrawal is succesfull.");

Console.WriteLine($"Account:" + \_account.Name);

Console.WriteLine($"Withdrawal Amount: {\_amount}");

}

else

{

Console.WriteLine("The amount was reversed");

}

}

}

**DEPOSITTRANSACTION.CS**

using System;

public class DepositTransaction

{

private Account \_account;

private decimal \_amount;

private bool \_executed = false;

private bool \_success = false;

private bool \_reversed = false;

public bool Executed

{

get

{

return \_executed;

}

}

public bool Success

{

get

{

return \_success;

}

}

public bool Reversed

{

get

{

return \_reversed;

}

}

public DepositTransaction(Account account, decimal amount)

{

\_account = account;

\_amount = amount;

}

public void Execute()

{

if (\_executed)

{

throw new Exception("Cannot execute this transaction as it has already been executed.");

}

\_executed = true;

\_success = \_account.Deposit(\_amount);

}

public void Rollback()

{

if (\_executed)

{

throw new Exception("Cannot execute this transaction as it has not been executed.");

}

\_executed = false;

if (\_reversed)

{

throw new Exception("Cannot reverse this transaction as it has already been reversed.");

}

\_reversed = true;

\_success = \_account.Deposit(\_amount);

}

public void Print()

{

if (\_success)

{

Console.WriteLine("The amount deposited from the account is " +

\_amount);

}

else if (\_reversed)

{

Console.WriteLine("The amount was reversed");

}

}

}

**TRANSFERTRANSACTION.CS**

using System;

public class TransferTransaction

{

private Account \_toAccount;

private Account \_fromAccount;

private decimal \_amount;

private DepositTransaction \_theDeposit;

private WithdrawTransaction \_theWithdraw;

private bool \_executed = false;

private bool \_success = false;

private bool \_reversed = false;

public bool Executed

{

get

{

return \_executed;

}

}

public bool Success

{

get

{

return \_success;

}

}

public bool Reversed

{

get

{

return \_reversed;

}

}

public TransferTransaction(Account FromAccount, Account toAccount, decimal amount)

{

\_fromAccount = FromAccount;

\_toAccount = toAccount;

\_amount = amount;

\_theWithdraw = new WithdrawTransaction(\_fromAccount, \_amount);

\_theDeposit = new DepositTransaction(\_toAccount, \_amount);

}

public void Execute()

{

if (\_executed)

{

throw new Exception("Cannot execute this transaction as it has already been executed.");

}

\_executed = true;

\_theWithdraw.Execute();

if (\_theWithdraw.Success)

{

\_theDeposit.Execute();

\_success = true;

}

else

{

throw new Exception("Cannot execute this transaction as the raw didn't go through");

}

}

public void Rollback()

{

if (\_executed)

{

throw new Exception("Cannot execute this transaction as it has been executed.");

}

\_executed = false;

if (\_reversed)

{

throw new Exception("Cannot reverse this transaction as it has eady been reversed.");

}

if (\_theWithdraw.Success)

{

\_theWithdraw.Rollback();

}

if (\_theDeposit.Success)

{

\_theDeposit.Rollback();

}

}

public void Print()

{

if (\_success)

{

Console.WriteLine("Transferred " + \_amount + " from " + \_fromAccount.Name + "'s Account to " + \_toAccount.Name + "'s Account ");

}

else if (\_reversed)

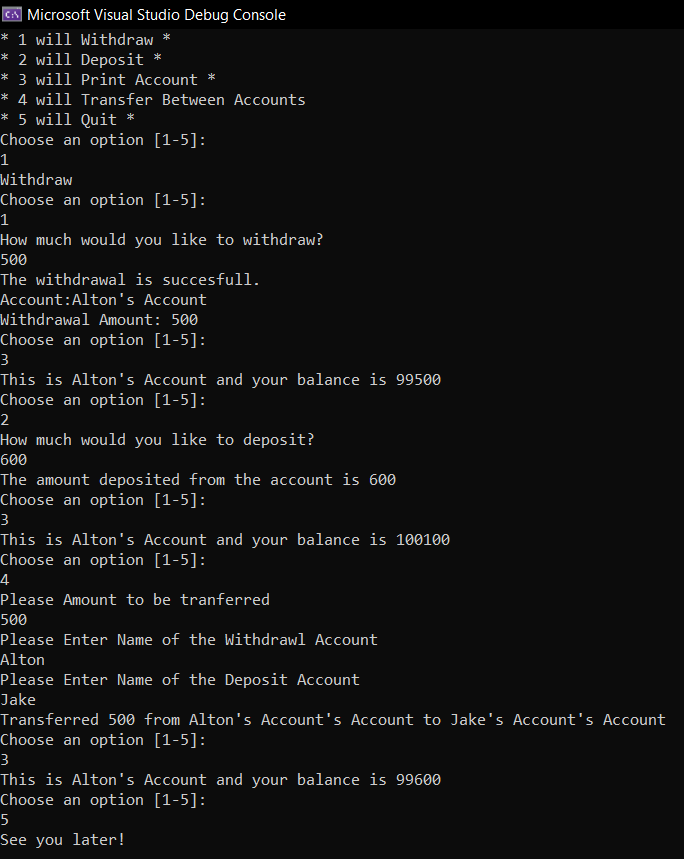
{

Console.WriteLine("The transaction didn't go through");

}

}

}

**SCREENSHOT OF RUNNING PROGRAM**

**Defining Object Roles through Abstraction**

Using classes allows us to define different roles that objects can play through the process of abstraction. Each class encapsulates the details and complexities of the role it represents. For example, our WithdrawTransaction, DepositTransaction and TransferTransaction classes abstract away the logic and details of performing those financial transactions.

**Encapsulation in the TransferTransaction Class**

Encapsulation is very evident in the TransferTransaction class. It handles transferring money between two accounts, but externally we have no visibility or need to know that it uses a WithdrawTransaction and DepositTransaction internally to accomplish this.

**Benefits of Defined Transactional Roles**

Overall, using classes to define distinct transactional roles allows our bank program to incorporate abstraction through high-level interfaces, while facilitating encapsulation to hide unnecessary implementation complexities from code that leverages these transaction classes. This lends flexibility, extensibility, and good separation of concerns to our design.