Deakin University

Object Oriented Development

OnTrack Submission

Multiple Bank Accounts

Submitted By:
Peter Stacey
pstacey
2020/05/01 10:56

Tutor:
Dipto Pratyaksa

Outcome	**Weight**
Evaluate Code	♦♦◊◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦◊◊◊
Design	♦♦♦◊◊
Justify	♦♦◊◊◊

As a continuation of the banking application, overall the app is growing into a larger design and work of coding, however this individual task, while important, was relatively small. It involved evaluating our existing code in order to extend it with new features, and the addition of a new class, which involves new code writing. My video will include additional evidence to align with the learning outcomes.

May 1, 2020

```
using System;
using System.Diagnostics;

namespace Task_6._2P
{
    enum MenuOption
    {
        CreateAccount,
        Withdraw,
        Deposit,
        Transfer,
        Print,
        Quit
    }

    /// <summary>
    /// BankSystem implements a banking system to operate on accounts
    /// </summary>
    class BankSystem
    {
        // Reads string input in the console
        /// <summary>
        /// Reads string input in the console
        /// </summary>
        /// <returns>
        /// The string input of the user
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static String ReadString(String prompt)
        {
            Console.Write(prompt + ": ");
            return Console.ReadLine();
        }

        // Reads integer input in the console
        /// <summary>
        /// Reads integerinput in the console
        /// </summary>
        /// <returns>
        /// The input of the user as an integer
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static int ReadInteger(String prompt)
        {
            int number = 0;
            string numberInput = ReadString(prompt);
            while (!(int.TryParse(numberInput, out number)))
            {
                Console.WriteLine("Please enter a whole number");
                numberInput = ReadString(prompt);
            }
            return Convert.ToInt32(numberInput);
        }

        // Reads integer input in the console between two numbers
        /// <summary>
        /// Reads integer input in the console between two numbers
        /// </summary>
        /// <returns>
        /// The input of the user as an integer
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        /// <param name="minimum">The minimum number allowed</param>
        /// <param name="maximum">The maximum number allowed</param>
        public static int ReadInteger(String prompt, int minimum, int maximum)
        {
            int number = ReadInteger(prompt);
            while (number < minimum || number > maximum)
            {
                Console.WriteLine("Please enter a whole number from " +
                                  minimum + " to " + maximum);
                number = ReadInteger(prompt);
            }
            return number;
        }

        // Reads decimal input in the console
        /// <summary>
        /// Reads decimal input in the console
        /// </summary>
        /// <returns>
        /// The input of the user as a decimal
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static decimal ReadDecimal(String prompt)
        {
            decimal number = 0;
            string numberInput = ReadString(prompt);
            while (!(decimal.TryParse(numberInput, out number)) || number < 0)
            {
                Console.WriteLine("Please enter a decimal number, $0.00 or
                 ↪  greater");
                numberInput = ReadString(prompt);
            }
            return Convert.ToDecimal(numberInput);
        }

        /// <summary>
        /// Displays a menu of possible actions for the user to choose
        /// </summary>
        private static void DisplayMenu()
        {
            Console.WriteLine("\n********************");
            Console.WriteLine("*       Menu       *");
            Console.WriteLine("********************");
            Console.WriteLine("*  1. New Account  *");
            Console.WriteLine("*  2. Withdraw     *");
            Console.WriteLine("*  3. Deposit      *");
            Console.WriteLine("*  4. Transfer     *");
            Console.WriteLine("*  5. Print        *");
            Console.WriteLine("*  6. Quit         *");
            Console.WriteLine("********************");
        }

        /// <summary>
        /// Returns a menu option chosen by the user
        /// </summary>
        /// <returns>
        /// MenuOption chosen by the user
        /// </returns>
        static MenuOption ReadUserOption()
        {
            DisplayMenu();
            int option = ReadInteger("Choose an option", 1,
                Enum.GetNames(typeof(MenuOption)).Length);
            return (MenuOption)(option - 1);
        }

        /// <summary>
        /// Attempts to deposit funds into an account at a bank
        /// </summary>
        /// <param name="bank">The bank holding the account to deposit into</param>
        static void DoDeposit(Bank bank)
        {
            Account account = FindAccount(bank);
            if (account != null)
            {
                decimal amount = ReadDecimal("Enter the amount");
                DepositTransaction transaction = new DepositTransaction(account,
                 ↪  amount);
                try
                {
                    bank.ExecuteTransaction(transaction);
                }
                catch (InvalidOperationException)
                {
                    transaction.Print();
                    return;
                }
                transaction.Print();
            }
        }

        /// <summary>
        /// Attempts to withdraw funds from an account at a bank
        /// </summary>
        /// <param name="bank">The bank holding account to withdraw from</param>
        static void DoWithdraw(Bank bank)
        {
            Account account = FindAccount(bank);
            if (account != null)
            {
                decimal amount = ReadDecimal("Enter the amount");
                WithdrawTransaction transaction = new WithdrawTransaction(account,
                 ↪  amount);
                try
                {
                    bank.ExecuteTransaction(transaction);
                }
                catch (InvalidOperationException)
                {
                    transaction.Print();
                    return;
                }
                transaction.Print();
            }
        }

        /// <summary>
        /// Attempts to transfer funds between accounts
        /// </summary>
        /// <param name="bank">The bank holding the account
        /// to transfer between</param>
        static void DoTransfer(Bank bank)
        {
            Console.WriteLine("Transfer from:");
            Account from = FindAccount(bank);
            Console.WriteLine("Transfer to:");
            Account to = FindAccount(bank);
            if (from != null && to != null)
            {
                decimal amount = ReadDecimal("Enter the amount");
                try
                {
                    TransferTransaction transaction = new TransferTransaction(from,
                     ↪  to, amount);
                    bank.ExecuteTransaction(transaction);
                    transaction.Print();
                }
                catch (Exception)
                {
                    // Currently this is handled in the TransferTransaction. This
                     ↪  will be changed
                }
            }
        }

        /// <summary>
        /// Outputs the account name and balance
        /// </summary>
        /// <param name="account">The account to print</param>
        static void DoPrint(Bank bank)
        {
            Account account = FindAccount(bank);
            if (account != null)
            {
                account.Print();
            }
        }

        /// <summary>
        /// Creates a new account and adds it to the Bank
        /// </summary>
        /// <param name="bank">The bank to create the account in</param>
        static void CreateAccount(Bank bank)
        {
            string name = ReadString("Enter account name");
            decimal balance = ReadDecimal("Enter the opening balance");
            bank.AddAccount(new Account(name, balance));
        }

        private static Account FindAccount(Bank bank)
        {
            Account account = null;
            string name = ReadString("Enter the account name");
            account = bank.GetAccount(name);
            if (account == null)
            {
                Console.WriteLine("That account name does not exist at this bank");
            }
            return account;
        }

        static void Main(string[] args)
        {
            Bank bank = new Bank();

            do
            {
                MenuOption chosen = ReadUserOption();
                switch (chosen)
                {
                    case MenuOption.CreateAccount:
                        CreateAccount(bank); break;

                    case MenuOption.Withdraw:
                        DoWithdraw(bank); break;

                    case MenuOption.Deposit:
                        DoDeposit(bank); break;

                    case MenuOption.Transfer:
                        DoTransfer(bank); break;

                    case MenuOption.Print:
                        DoPrint(bank); break;

                    case MenuOption.Quit:
                    default:
                        Console.WriteLine("Goodbye");
                        System.Environment.Exit(0); // terminates the program
                        break; // unreachable
                }
            } while (true);
        }
    }
}
```

```
using System;
using System.Collections.Generic;

namespace Task_6._2P
{
    /// <summary>
    /// Prototype for a bank to hold accounts
    /// </summary>
    class Bank
    {
        // Instance variables
        private List<Account> _accounts;

        /// <summary>
        /// Creates an empty bank object with a list for accounts
        /// </summary>
        public Bank()
        {
            _accounts = new List<Account>();
        }

        /// <summary>
        /// Adds an account to the Bank accounts register
        /// </summary>
        /// <param name="account"></param>
        public void AddAccount(Account account)
        {
            _accounts.Add(account);
        }

        /// <summary>
        /// Returns the first Account corresponding to the name, or
        /// null if there is no account matching the criteria
        /// </summary>
        /// <param name="name"></param>
        /// <returns>
        /// Account matching the provided name, or null
        /// </returns>
        public Account GetAccount(string name)
        {
            foreach (Account account in _accounts)
            {
                if (account.Name == name)
                {
                    return account;
                }
            }
            return null;
        }

        /// <summary>
        /// Executes a deposit into an account
        /// </summary>
        /// <param name="transaction">DepositTransaction to execute</param>
        public void ExecuteTransaction(DepositTransaction transaction)
        {
            try
            {
                transaction.Execute();
            }
            catch (InvalidOperationException exception)
            {
                Console.WriteLine("An error occurred in executing the transaction");
                Console.WriteLine("The error was: " + exception.Message);
            }
        }

        /// <summary>
        /// Executes a WithdrawTransaction on an account
        /// </summary>
        /// <param name="transaction">WithdrawTransaction to execute</param>
        public void ExecuteTransaction(WithdrawTransaction transaction)
        {
            try
            {
                transaction.Execute();
            }
            catch (InvalidOperationException exception)
            {
                Console.WriteLine("An error occurred in executing the transaction");
                Console.WriteLine("The error was: " + exception.Message);
            }
        }

        /// <summary>
        /// Transfers funds between accounts held by the bank
        /// </summary>
        /// <param name="transaction">TransferTransaction to execute</param>
        public void ExecuteTransaction(TransferTransaction transaction)
        {
            try
            {
                transaction.Execute();
            }
            catch (InvalidOperationException exception)
            {
                Console.WriteLine("An error occurred in executing the transaction");
                Console.WriteLine("The error was: " + exception.Message);
            }
        }
    }
}
```

```
using System;

namespace Task_6._2P
{
    /// <summary>
    /// A bank account class to hold the account name and balance details
    /// </summary>
    class Account
    {
        // Instance variables
        private String _name;
        private decimal _balance;

        // Read-only properties
        public String Name { get => _name; }
        public decimal Balance { get => _balance; }


        /// <summary>
        /// Class constructor
        /// </summary>
        /// <param name="name">The name string for the account</param>
        /// <param name="balance">The decimal balance of the account</param>
        public Account(String name, decimal balance = 0)
        {
            _name = name;
            if (balance < 0)
                return;
            _balance = balance;
        }

        /// <summary>
        /// Deposits money into the account
        /// </summary>
        /// <returns>
        /// Boolean whether the deposit was successful (true) or not (false)
        /// </returns>
        /// <param name="amount">The decimal amount to add to the balance</param>
        public Boolean Deposit(decimal amount)
        {
            if ((amount < 0) || (amount == decimal.MaxValue))
                return false;

            _balance += amount;
            return true;
        }

        /// <summary>
        /// Withdraws money from the account (with no overdraw protection currently)
        /// </summary>
        /// <returns>
        /// Boolean whether the withdrawal was successful (true) or not (false)
        /// </returns>
        /// <param name="amount">The amount to subtract from the balance</param>
        public Boolean Withdraw(decimal amount)
        {
            if ((amount < 0) || (amount > _balance))
                return false;

            _balance -= amount;
            return true;
        }

        /// <summary>
        /// Outputs the account name and current balance as a string
        /// </summary>
        public void Print()
        {
            Console.WriteLine("Account Name: {0}, Balance: {1}",
                _name, _balance.ToString("C"));
        }
    }
}
```

```csharp
using System;

namespace Task_6._2P
{
    /// <summary>
    /// Prototype for a Withdraw transaction
    /// </summary>
    class WithdrawTransaction
    {
        // Instance variables
        private Account _account;
        private decimal _amount;
        private Boolean _executed;
        private Boolean _success;
        private Boolean _reversed;

        // Properties
        public Boolean Executed { get => _executed; }
        public Boolean Success { get => _success; }
        public Boolean Reversed { get => _reversed; }

        /// <summary>
        /// Constructs a WithdrawTransaction
        /// </summary>
        /// <param name="account">Account to withdraw from</param>
        /// <param name="amount">Amount to withdraw</param>
        public WithdrawTransaction(Account account, decimal amount)
        {
            _account = account;
            if (amount > 0)
            {
                _amount = amount;
            }
            else
            {
                throw new ArgumentOutOfRangeException("Withdrawal amount must be >
                ↪  $0.00");
            }
            // _executed, _success, _reversed false by default
        }

        /// <summary>
        /// Prints the details and status of the withdrawal
        /// </summary>
        public void Print()
        {
            Console.WriteLine(new String('-', 85));
            Console.WriteLine("|{0, -20}|{1, 20}|{2, 20}|{3, 20}|",
                "ACCOUNT", "WITHDRAW AMOUNT", "STATUS", "CURRENT BALANCE");
            Console.WriteLine(new String('-', 85));
            Console.Write("|{0, -20}|{1, 20}|", _account.Name,
            ↪  _amount.ToString("C"));
            if (!_executed)
            {
                Console.Write("{0, 20}|", "Pending");
            }
            else if (_reversed)
            {
                Console.Write("{0, 20}|", "Withdraw reversed");
            }
            else if (_success)
            {
                Console.Write("{0, 20}|", "Withdraw complete");
            }
            else if (!_success)
            {
                Console.Write("{0, 20}|", "Insufficient funds");
            }
            Console.WriteLine("{0, 20}|", _account.Balance.ToString("C"));
            Console.WriteLine(new String('-', 85));
        }

        /// <summary>
        /// Executes the withdrawal
        /// </summary>
        /// <exception cref="System.InvalidOperationException">Thrown
        /// when the withdraw is already complete or insufficient funds</exception>
        public void Execute()
        {
            if (_executed && _success)
            {
                throw new InvalidOperationException("Withdraw previously executed");
            }
            _executed = true;

            _success = _account.Withdraw(_amount);
            if (!_success)
            {
                throw new InvalidOperationException("Insufficient funds");
            }
        }

        /// <summary>
        /// Reverses the withdraw if previously executed successfully
        /// </summary>
        /// <exception cref="System.InvalidOperationException">Thrown
        /// if already rolled back or if there are insufficient
        /// funds to complete the rollback</exception>
        public void Rollback()
        {
            if (_reversed)
            {
                throw new InvalidOperationException("Transaction already reversed");
            }
            else if (!_success)
            {
                throw new InvalidOperationException(
                    "Withdraw not successfully executed. Nothing to rollback.");
            }
            _reversed = _account.Deposit(_amount); // Deposit returns boolean
            if (!_reversed) // Deposit didn't occur
            {
                throw new InvalidOperationException("Invalid amount");
            }
            _reversed = true;
        }
    }
}
```

```
using System;

namespace Task_6._2P
{
    /// <summary>
    /// Prototype for a deposit transaction
    /// </summary>
    class DepositTransaction
    {
        // Instance variables
        private Account _account;
        private decimal _amount;
        private Boolean _executed;
        private Boolean _success;
        private Boolean _reversed;

        // Properties
        public Boolean Executed { get => _executed; }
        public Boolean Success { get => _success; }
        public Boolean Reversed { get => _reversed; }

        /// <summary>
        /// Constructs a deposit transaction object
        /// </summary>
        /// <param name="account">Account to deposit into</param>
        /// <param name="amount">Amount to deposit</param>
        public DepositTransaction(Account account, decimal amount)
        {
            _account = account;
            if (amount > 0)
            {
                _amount = amount;
            }
            else
            {
                throw new ArgumentOutOfRangeException(
                    "Deposit amount invalid: {0}", amount.ToString("C"));
            }
            // _executed, _success, _reversed false by default
        }

        /// <summary>
        /// Prints the details and status of a deposit
        /// </summary>
        public void Print()
        {
            Console.WriteLine(new String('-', 85));
            Console.WriteLine("|{0, -20}|{1, 20}|{2, 20}|{3, 20}|",
                "ACCOUNT", "DEPOSIT AMOUNT", "STATUS", "CURRENT BALANCE");
            Console.WriteLine(new String('-', 85));
            Console.Write("|{0, -20}|{1, 20}|", _account.Name,
            ↪ _amount.ToString("C"));
            if (!_executed)
            {
                Console.Write("{0, 20}|", "Pending");
            }
            else if (_reversed)
            {
                Console.Write("{0, 20}|", "Deposit reversed");
            }
            else if (_success)
            {
                Console.Write("{0, 20}|", "Deposit complete");
            }
            else if (!_success)
            {
                Console.Write("{0, 20}|", "Invalid deposit");
            }
            Console.WriteLine("{0, 20}|", _account.Balance.ToString("C"));
            Console.WriteLine(new String('-', 85));
        }

        /// <summary>
        /// Executes a deposit transaction
        /// </summary>
        public void Execute()
        {
            if (_executed && _success)
            {
                throw new InvalidOperationException("Deposit previously executed");
            }
            _executed = true;

            _success = _account.Deposit(_amount);
            if (!_success)
            {
                _executed = false;
                throw new InvalidOperationException("Deposit amount invalid");
            }
        }

        /// <summary>
        /// Reverses a deposit if previously executed successfully
        /// </summary>
        public void Rollback()
        {
            if (_reversed)
            {
                throw new InvalidOperationException("Transaction already reversed");
            }
            else if (!_success)
            {
                throw new InvalidOperationException(
                    "Deposit not successfully executed. Nothing to rollback.");
            }
            _reversed = _account.Withdraw(_amount); // Withdraw returns boolean
            if (!_reversed) // Withdraw didn't occur
            {
                throw new InvalidOperationException("Insufficient funds to
                 ↪  rollback");
            }
            _reversed = true;
        }
    }
}
```

```csharp
using System;

namespace Task_6._2P
{
    /// <summary>
    /// Prototype for a transfer transaction
    /// </summary>
    class TransferTransaction
    {
        // Instance variables
        private Account _fromAccount;
        private Account _toAccount;
        private decimal _amount;
        private DepositTransaction _deposit;
        private WithdrawTransaction _withdraw;
        private bool _executed;
        private bool _reversed;

        // Properties
        public bool Executed { get => _executed; }
        public bool Reversed { get => _reversed; }
        public bool Success { get => (_deposit.Success && _withdraw.Success); }

        /// <summary>
        /// Constructor for a transfer transaction
        /// </summary>
        /// <param name="fromAccount">The account to transfer from</param>
        /// <param name="toAccount">The account to transfer to</param>
        /// <param name="amount">The amount to transfer</param>
        /// <exception cref="System.ArgumentOutOfRangeException">Thrown
        /// when the amount is negative</exception>
        public TransferTransaction(Account fromAccount, Account toAccount, decimal
        ↪   amount)
        {
            _fromAccount = fromAccount;
            _toAccount = toAccount;
            if (amount < 0)
            {
                throw new ArgumentOutOfRangeException("Negative transfer amount");
            }
            _amount = amount;

            _withdraw = new WithdrawTransaction(_fromAccount, _amount);
            _deposit = new DepositTransaction(_toAccount, _amount);
        }

        /// <summary>
        /// Prints the details of the transfer
        /// </summary>
        public void Print()
        {
            Console.WriteLine(new String('-', 85));
            Console.WriteLine("|{0, -20}|{1, 20}|{2, 20}|{3, 20}|",
                "FROM ACCOUNT", "To ACCOUNT", "TRANSFER AMOUNT", "STATUS");
            Console.WriteLine(new String('-', 85));
            Console.Write("|{0, -20}|{1, 20}|{2, 20}|", _fromAccount.Name,
             ↪  _toAccount.Name, _amount.ToString("C"));
            if (!_executed)
            {
                Console.WriteLine("{0, 20}|", "Pending");
            }
            else if (_reversed)
            {
                Console.WriteLine("{0, 20}|", "Transfer reversed");
            }
            else if (Success)
            {
                Console.WriteLine("{0, 20}|", "Transfer complete");
            }
            else if (!Success)
            {
                Console.WriteLine("{0, 20}|", "Transfer failed");
            }
            Console.WriteLine(new String('-', 85));
        }

        /// <summary>
        /// Executes the transfer
        /// </summary>
        /// <exception cref="System.InvalidOperationException">Thrown
        /// when previously executed or deposit or withdraw fail</exception>
        public void Execute()
        {
            if (_executed)
            {
                throw new InvalidOperationException("Transfer previously executed");
            }
            _executed = true;

            try
            {
                _withdraw.Execute();
            }
            catch (InvalidOperationException exception)
            {
                Console.WriteLine("Transfer failed with reason: " +
                 ↪  exception.Message);
                _withdraw.Print();
            }

            if (_withdraw.Success)
            {
                try
                {
                    _deposit.Execute();
                }
                catch (InvalidOperationException exception)
                {
                    Console.WriteLine("Transfer failed with reason: " + exception.Message);
                    _deposit.Print();
                    try
                    {
                        _withdraw.Rollback();
                    }
                    catch (InvalidOperationException e)
                    {
                        Console.WriteLine("Withdraw could not be reversed with reason: " + e.Message);
                        _withdraw.Print();
                    }
                }
            }
        }

        /// <summary>
        /// Rolls the transfer back
        /// </summary>
        /// <exception cref="System.InvalidOperationException">Thrown
        /// when the rollback has already been executed or it fails</exception>
        public void Rollback()
        {
            if (!_executed)
            {
                throw new InvalidOperationException("Transfer not executed. Nothing to rollback.");
            }

            if (_reversed)
            {
                throw new InvalidOperationException("Transfer already rolled back");
            }

            if (this.Success)
            {
                try
                {
                    _deposit.Rollback();
                }
                catch (InvalidOperationException exception)
                {
                    Console.WriteLine("Failed to rollback deposit: "
                        + exception.Message);
                    return;
                }

                try
                {
                    _withdraw.Rollback();
                }
                catch (InvalidOperationException exception)
                {
                    Console.WriteLine("Failed to rollback withdraw: "
                        + exception.Message);
                    return;
                }
            }
            _reversed = true;
        }
    }
}
```