

Testing Document

Commerce Bank System

04/26/2020

Team Members

Michelle Frost

Debbie Kirchner

Brian Roden

Linden Stirk

Issac Zeilinger

Document Control

Change History

Revision	Change Date	Change By	Description of changes
V1.0	04/13/2020	Linden Stirk	Initial release
V1.1	04/15/2020	Linden Stirk	Updated test specifications and test report
V1.2	04/17/2020	Linden Stirk	Various updates

Table of Contents

1 INTRODUCTION	4
1.1 Definitions	4
2 TEST PLAN	4
2.1 Approach	4
2.2 Components Tested with Unit Tests	5
2.3 Components Tests Manually	5
2.4 Identifiers	5
2.5 Testing Tasks and Deliverables	6
3 TEST SPECIFICATIONS	6
3.1 Test Cases	6
3.2 Unit Tests	10
4 TEST REPORTS	13
4.1 Incidents	13
4.2 Unit Test Results	14
4.3 Defects	14
4.4 Summary	15

1 Introduction

The purpose of this document is to outline the test plan for the Commerce Bank System. The Commerce Bank System is a web application built with an MVC (Model-View-Controller) type of architecture that allows customers who bank with Commerce to access information about their accounts online. For specific details in regards to the requirements of the system, reference the Requirements Document and the Architecture Document.

1.1 Definitions

Commerce Bank System: the web-based application that is being described within this document.

Transaction: a record of some amount of money that has moved either in or out of a person's bank account

Notification: a visual indication that a target event has happened

Trigger: a specific event that causes a notification to appear

Use case: describes a goal-oriented interaction between the system and an actor. A use case may define several variants called scenarios that result in different paths through the use case and usually different outcomes.

Scenario: one path through a use case

Actor: user or another software system that receives value from a use case.

Role: category of users that share similar characteristics.

Client: the person or organization for which the Commerce Bank System is being developed.

User: the person or people who will interact with the Commerce Bank System.

Developer: the person or people who are developing the Commerce Bank System.

2 Test Plan

2.1 Approach

For unit testing, the .NET Core project, the open-source testing library, Xunit, was used. For testing individual components and programs within the Commerce Bank System, Xunit enabled the team to use white-box testing techniques to make sure the system works as intended. While it's impossible to say that the software is bug-free, the tests found no critical bugs during the testing phase.

However, a lack of full understanding and experience with the .NET Core framework contributed to the lack of programmatic tests for this application. Using the .NET Core framework with an MVC type architecture, it was, in some respect, difficult to perform tests for integration tests. The main issue stemmed from the controller object, which is instantiated during the program startup and is never explicitly referenced in the program. Here a .NET Core singleton service passes in an instance of the interface and the access service, but this specific object is not accessible outside of the controller object. Instead, the controller was tested manually using a fake access service, and later using the real database access service.

2.2 Components Tested with Unit Tests

The following is a list of backend components that were tested with Xunit tests.

- Transaction Reader
- Fake Database Access Service

2.3 Components Tested Manually

The following is a list of components that were tested manually. Some components were tested with a combination of manual and unit tests, so items may appear in both lists.

- Values Controller
- Values Controller DAO (Interface)
- Fake Database Access Service
- Access Service
- Fill DB

2.4 Identifiers

Each main test is designated with a unique identifier, such as 001, 002, 003, etc. Each unit test comprises several Xunit tests. These are identified by the method name.

2.5 Testing Tasks and Deliverables

The testing tasks are outlined in the Test Specifications section below. These tests include unit tests written with Xunit and manual user tests. Pass/Fail criteria are indicated in the following section with specific details in the unit test code provided

after the specification tables. The Test Report at the end of this document shows detected defects and changes.

3 Test Specifications

3.1 Test Cases

Test Case ID:	001
Title:	Loading Transactions From a CSV File
Feature/Subfeature:	Transaction Reader
Purpose:	To test the methods in the TransactionReader.cs class.
Initial Conditions:	CustomerATest.csv and CustomerATest_BlankCell.csv are accessible
Test Data:	CustomerATest.csv contains mock data. The spreadsheet contains 74 rows of transaction data including the account number, the account type, the processing date, balance, transaction type, and the transaction description
Test Actions:	There are four unit tests: LoadList() BlankCell() DollarAmount() LastCell() See the unit tests below for details
Expected Results:	LoadList() should return 74 rows BlankCell() should return an empty string "" DollarAmount() should return \$800 LastCell() should return "Starbucks"

Test Case ID:	002
Title:	Fake Database Access Service (Get/Insert Methods)
Feature/Subfeature:	Fake Database Access Service
Purpose:	To test the methods in the Fake Database Access Service, so the Values Controller and DAO interface can be tested manually.

Initial Conditions:	N/A
Test Data:	The FakeDatabaseAccessService serves as an in-memory database to test the controller and interface while the real access service is in development.
Test Actions:	There are three unit tests: GetAllTransactions() GetTransactionByAccount() InsertTransaction() See the unit tests below for details
Expected Results:	GetAllTransactions() and GetTransactionByAccount() return values are compared against an in-memory list. InsertTransaction() should the same value that it inserts See the unit tests below for details

Test Case ID:	003
Title:	Fill Database from the Transaction Reader
Feature/Subfeature:	FillDB
Purpose:	To test the methods of the FillDB.cs class
Initial Conditions:	The Transaction Reader has loaded the transaction data into memory. The commerceDB MS SQL Server database is running.
Test Data:	CustomerATest.csv contains mock data. The spreadsheet contains 74 rows of transaction data including the account number, the account type, the processing date, balance, transaction type, and the transaction description
Test Actions:	populateDB() For each Transaction object in the in-memory list, send a parameterized INSERT command to the database with a stored procedure

Expected Results:	The rows of the CustomerATest.csv should match with the rows of the commerceDB database. This is checked manually.
--------------------------	--

Test Case ID:	004
Title:	Values Controller Interface
Feature/Subfeature:	ValuesDao
Purpose:	To test the methods of the ValuesDao interface.
Initial Conditions:	The web service is running, and the FakeDatabaseAccessService is running in the singleton service in Startup.cs
Test Data:	Transaction objects: "000001", "03/10/20", "3,000.00", "DR", "\$1.00", "Test transaction 1" "000001", "03/12/20", "3,002.00", "DR", "\$2", "Test transaction 2" "000002", "03/12/20", "3,002.00", "DR", "\$2", "Test transaction 3"
Test Actions:	Manually compare the values shown above in the in-memory list with the values in the json object at https://localhost:44373/api/values
Expected Results:	The values in the json objects and in the list should match

Test Case ID:	005
Title:	Values Controller
Feature/Subfeature:	Controller
Purpose:	To test the methods of the Values Controller and to test integration with the Values DAO interface.

Initial Conditions:	The web service is running and the FakeDatabaseAccessService and the ValuesDao interface is running in the singleton service in Startup.cs
Test Data:	Transaction objects: "000001", "03/10/20", "3,000.00", "DR", "\$1.00", "Test transaction 1" "000001", "03/12/20", "3,002.00", "DR", "\$2", "Test transaction 2" "000002", "03/12/20", "3,002.00", "DR", "\$2", "Test transaction 3"
Test Actions:	Manually compare the values shown above in the in-memory list with the values in the json object at https://localhost:44373/api/values
Expected Results:	The values in the json objects and the list should match

Test Case ID:	006
Title:	Access Service
Feature/Subfeature:	MS SQL Server Access Service
Purpose:	To test the methods in the correct access service that communicates with the commerceDB database.
Initial Conditions:	The web service is running and the Access Service and ValuesDao interface is running in the singleton service in Startup.cs
Test Data:	The commerceDB database table is filled with the data from CustomerATest.csv.
Test Actions:	getAllTransactions() Manually compare the values in the commerceDB table with the values in the json object at https://localhost:44373/api/values

Expected Results:	The values in the json objects and in the commerceDB table should match
--------------------------	---

3.2 Unit Tests

The following unit tests use Xunit. Since Xunit does not perform a deep comparison on custom objects, several tests use a series of *Assert.Equal* statements to each compare the

Test Case ID 001: Transaction Loader

```
public class TransactionLoaderTest
{
    // expected values are based on the values inside the CustomerATest.csv file

    [Fact]
    public void Test_TransactionReader_LoadList()
    {
        int expectedVal = 74; // number of rows
        int actualVal =
        CommerceApi.TransactionReader.
        ImportTransactions("test_input/CustomerATest.csv").
        Count;
        Assert.Equal(expectedVal, actualVal);
    }

    [Fact]
    public void Test_TransactionReader_BlankCell()
    {
        string expectedVal = "";
        Transaction actualVal =
        CommerceApi.TransactionReader.ImportTransactions("test_input/CustomerATest_BlankCe
        ll.csv")[0];
        Assert.Equal(expectedVal, actualVal.processDate);
    }

    [Fact]
    public void Test_TransactionReader_DollarAmount()
    {
        string expectedVal = "$800.00";
        Transaction actualVal =
        CommerceApi.TransactionReader.ImportTransactions("test_input/CustomerATest.csv")[2
        ];
        Assert.Equal(expectedVal, actualVal.amount);
    }

    [Fact]
    public void Test_TransactionReader_LastCell()
    {
```

```
string expectedVal = "Starbucks";
Transaction actualVal =
CommerceApi.TransactionReader.ImportTransactions("test_input/CustomerATest.csv")[1
];
Assert.Equal(expectedVal, actualVal.description);
}
}
```

Test Case ID 002: Fake Database Access Service

```
public class FakeDBAccessServiceTests
{

    FakeDatabaseAccessService fakeDatabaseAccessService = new
    FakeDatabaseAccessService();

    // lists and objects for expected values
    List<Transaction> actualTransactions = new List<Transaction>();

    List<Transaction> expectedTransactions = new List<Transaction>();
    Transaction expectedTransaction01 = new Transaction("000001", "03/10/20",
    "3,000.00", "DR", "$1.00", "Test transaction 1");
    Transaction expectedTransaction02 = new Transaction("000001", "03/12/20",
    "3,002.00", "DR", "$2", "Test transaction 2");

    List<Transaction> expectedTransactionsByAccNum = new List<Transaction>();
    Transaction expectedTransactionByAccNum01 = new Transaction("000002", "03/12/20",
    "3,002.00", "DR", "$2", "Test transaction 3");

    public FakeDBAccessServiceTests()
    {
        actualTransactions = fakeDatabaseAccessService.getAllTransactions();
        expectedTransactions.Add(expectedTransaction01);
        expectedTransactions.Add(expectedTransaction02);
        expectedTransactionsByAccNum.Add(expectedTransactionByAccNum01);
    }

    [Fact]
    public void testGetAllTransactions()
    {
        // Xunit cannot perform deep comparison on two objects,
        // in the loop it compares each expected object's values against each actual
        object's values
        for (int i = 0; i < actualTransactions.Count; i++)
        {
            Assert.Equal(expectedTransactions[i].accountNumber,
            actualTransactions[i].accountNumber);
            Assert.Equal(expectedTransactions[i].balance, actualTransactions[i].balance);
            Assert.Equal(expectedTransactions[i].amount, actualTransactions[i].amount);
            Assert.Equal(expectedTransactions[i].processDate,
            actualTransactions[i].processDate);
            Assert.Equal(expectedTransactions[i].description,
            actualTransactions[i].description);
        }
    }
}
```

```
Assert.Equal(expectedTransactions[i].accountType,
actualTransactions[i].accountType);
}
}

[Fact]
public void testInsertTransaction()
{
    // add expected transaction to local list
    Transaction expectedInsertTransaction = new Transaction("000001", "03/15/20",
    "4,000.00", "DR", "$1000.00", "Test insert transaction");
    expectedTransactions.Add(expectedInsertTransaction);

    // call access service
    Transaction actualInsertTransaction =
    fakeDatabaseAccessService.insertTransaction(expectedInsertTransaction);

    // compare return type
    Assert.Equal(expectedInsertTransaction.accountNumber,
    actualInsertTransaction.accountNumber);
    Assert.Equal(expectedInsertTransaction.balance, actualInsertTransaction.balance);
    Assert.Equal(expectedInsertTransaction.amount, actualInsertTransaction.amount);
    Assert.Equal(expectedInsertTransaction.processDate,
    actualInsertTransaction.processDate);
    Assert.Equal(expectedInsertTransaction.description,
    actualInsertTransaction.description);
    Assert.Equal(expectedInsertTransaction.accountType,
    actualInsertTransaction.accountType);

    // get all transactions again (now including the added one)
    testGetAllTransactions();
}

[Fact]
public void testGetTransactionByAccountNumber()
{
    // call access service
    List<Transaction> actualTransactionsByAccNum =
    fakeDatabaseAccessService.getTransactionByAccountNumber(000002);

    // compare return type
    for (int i = 0; i < actualTransactionsByAccNum.Count; i++)
    {
        Assert.Equal(expectedTransactionsByAccNum[i].accountNumber,
        actualTransactionsByAccNum[i].accountNumber);
        Assert.Equal(expectedTransactionsByAccNum[i].balance,
        actualTransactionsByAccNum[i].balance);
        Assert.Equal(expectedTransactionsByAccNum[i].amount,
        actualTransactionsByAccNum[i].amount);
        Assert.Equal(expectedTransactionsByAccNum[i].processDate,
        actualTransactionsByAccNum[i].processDate);
        Assert.Equal(expectedTransactionsByAccNum[i].description,
        actualTransactionsByAccNum[i].description);
        Assert.Equal(expectedTransactionsByAccNum[i].accountType,
        actualTransactionsByAccNum[i].accountType);
    }
}
```

4 Test Reports

Incidents are designated with a unique identifier with a three number prefix followed by 'U' to indicate a unit test incident or an 'M' to indicate a manual test incident.

4.1 Incidents

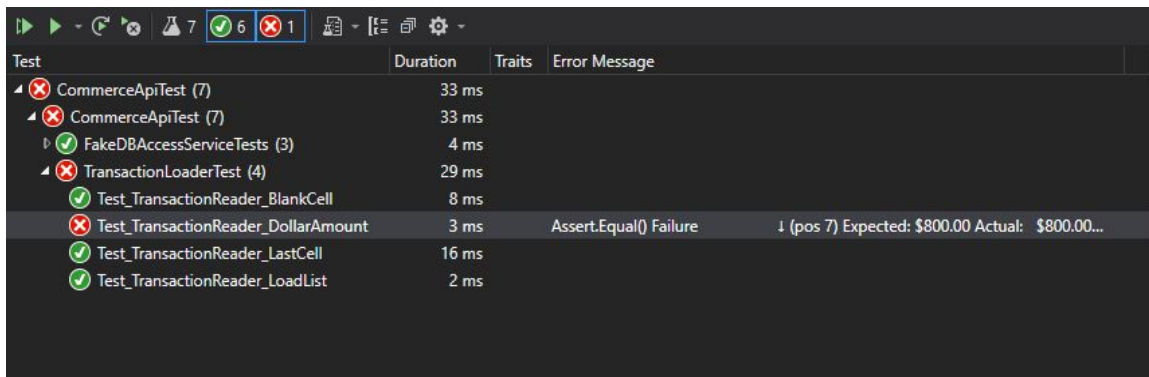
Incident ID:	001-U
Description:	Issue with the Transaction Loader parsing amounts with the '\$' sign
Originator:	Linden - Project Manager/Tester
Discover Date:	04/13/2020
Severity:	Low
Steps Required to Produce Incident:	Run CommerceApiTest.TransactionLoaderTest unit tests
Responder:	Linden
Current Status:	Closed
Cause:	Possible issue with Xunit's Assert.Equal() method causing the expected and actual values to be
Resolution:	Valid, No Pass
Addressed Date:	04/13/2020
Creation Phase:	Implementation
Detection Phase:	Testing
Correction Time:	Less than 1 hour

Incident ID:	001-U
Description:	Xunit not recognizing equal Transaction objects
Originator:	Linden - Project Manager/Tester
Discover Date:	04/13/2020
Severity:	Low
Steps Required to Produce Incident:	Run CommerceApiTest.TransactionLoaderTest unit tests
Responder:	Linden
Current Status:	Closed

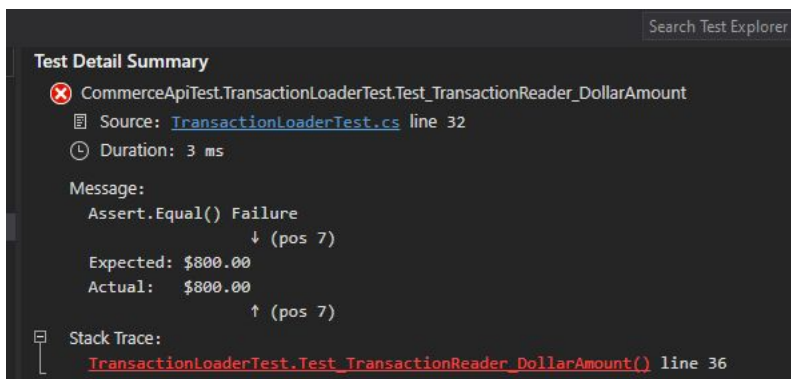
Cause:	Issue with Xunit's Assert.Equal() method, which does not perform a deep comparison between two custom Transaction objects.
Resolution:	Using separate Assert.Equal() methods to compare the Transaction objects' variables value by value.
Addressed Date:	04/13/2020
Creation Phase:	Testing
Detection Phase:	Testing
Correction Time:	Less than 1 hour

4.2 Unit Test Results

The following shows the results of the unit tests. There was only one test failure, but as noted in the test report above, this test was deemed to be a valid no pass because the oracle and the actual values are equivalent.



Test	Duration	Traits	Error Message
CommerceApiTest (7)	33 ms		
CommerceApiTest (7)	33 ms		
FakeDBAccessServiceTests (3)	4 ms		
TransactionLoaderTest (4)	29 ms		
Test_TransactionReader_BlankCell	8 ms		
Test_TransactionReader_DollarAmount	3 ms		Assert.Equal() Failure ↓ (pos 7) Expected: \$800.00 Actual: \$800.00...
Test_TransactionReader_LastCell	16 ms		
Test_TransactionReader_LoadList	2 ms		



Test Detail Summary
CommerceApiTest.TransactionLoaderTest.Test_TransactionReader_DollarAmount
Source: TransactionLoaderTest.cs line 32
Duration: 3 ms
Message:
Assert.Equal() Failure
↓ (pos 7)
Expected: \$800.00
Actual: \$800.00
↑ (pos 7)
Stack Trace:
TransactionLoaderTest.Test_TransactionReader_DollarAmount() line 36

4.3 Defects

No defects in the CommerceApi program were discovered with the unit and manual testing. The reported incidents were in regards to Xunit testing problems, not with the tested program. However, the unit tests only provide partial coverage of the backend components. There may be defects that were not detected by the unit and manual tests.

4.4 Summary

The backend testing phase supports the notion that the Commerce Bank System works as intended, at least from a backend perspective. This includes the Transaction Reader, Fill DB methods, API Values Controller, Access Service, and the Values DAO interface. Defects may be present, but none were discovered in the testing phase. Future testing needs to include the frontend, and unit tests need to be better realized with proper integration testing techniques and better deep comparison functionality. Integration testing relied heavily on manual testing. The temporary solution to the deep comparison issue with Xunit's Assert.Equal() method was to compare the values of the objects' variables one at a time. A more elegant solution would employ a custom comparison function.