# Unit Testing in C# with nUnit – Level 2

In this set of work we extend beyond Level 1 with raising awareness on Test First Development and Thread Safety and how to use the TPL to test with multiple threads. Please complete Level 1 prior to working on this Level.

### Adding a feature, and tests (Test first!)

So now that we have a working BankAccount class library, our customer is just not satisfied with just the ability to Debit. They came back and asked ust o also add the ability to Credit on accounts.

First, let’s write a test which will attempt to use the API to perform the credit:

[Test]

public void Credit\_Simple\_Amount\_UpdatesBalance()

{

// arrange

double beginningBalance = 11.99;

double creditAmount = 4.55;

double expected = 16.53;

BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

// act

account.Credit(creditAmount);

// assert

double actual = account.Balance;

Assert.AreEqual(expected, actual, 0.01, "Account not credited correctly");

}

Writing a test first allows us to think about how our API will actually be consumed. Is it consistent with the rest of the API? Does it make sense? Is it easy to consume?

Can you think of any other unit tests we could create for a “Credit” method? What about negative tests? Build your suite up now!

Since we have not implemented the Credit method in the BankAccount class we will not be able to build in VS.NET.

What we need to do is consider the above use, what is the minimum API we could create in order to satisfy this, and any other tests we created around the ability to Credit a bank account?

In BankAccount.cs

public void Credit(double amount)

{

Balance += Math.Abs(amount);

}

This satisfies the requirements as stated by the unit tests.

### Thread Safety

A significant portion of the code which we will author will be executed in a multi-threaded environment. Let’s take a look at our BankAccount class from the threading perspective.

If an instance of the BankAccount class was shared across threads, we could see a possible race condition between multiple calls to the Debit (balance checking) methods. That is, if someone calls Debit a few times, depending on where each thread gets too, they account could get into an overdraft state quite quickly. Notice that we are calculating the new balance at the top of the function, performing our validation, and then when it all passes we set the Balance on the class.

Let’s write a unit test to attempt to prove that this will fail.

[Test]

public void Debit\_Multi\_Threaded()

{

// arrange

double beginningBalance = 10;

double debitAmount = 1;

double expected = 0;

BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

// act

List<Task> tasks = new List<Task>();

for (int i = 0; i < 10; i++)

{

tasks.Add(Task.Run(() => account.Debit(debitAmount)));

}

Task.WaitAll(tasks.ToArray());

// assert

double actual = account.Balance;

Assert.AreEqual(expected, actual, 0.01, "Account not debited correctly");

}

In the code above, we see that we will call the Debit function on the same account object, in parallel. Depending on the load on the CPU and the scheduler each of these calls to debit may get called out of order, and may not finish completely during execution. If you execute this test as it, it most likely will not fail. This is because your machine is probably not under heavy load, and the calls to Debit will probably fully complete during each CPU cycle. If you hit run enough, you might be able to trip the failed case.

Let’s inject a temporary call to Sleep in our Debit method to demonstrate more fully.

Here is the updated Debit function:

public void Debit(double amount)

{

double newBalance = Balance - Math.Abs(amount);

if (amount > Balance)

{

throw new ArgumentOutOfRangeException("amount", amount, DebitAmountExceedsBalanceMessage);

}

if (newBalance < 0)

{

throw new ArgumentOutOfRangeException("amount", amount, DebitAmountLessThanZeroMessage);

}

System.Threading.Thread.Sleep(0);

Balance = newBalance;

}

Notice the call to Thread.Sleep(0). 0 is sufficient to cause the scheduler to allow for other Tasks to kick in and execute. Now run the unit test. It should fail.

Account not debited correctly Expected: 0.0d +/- 0.01d But was: 6.0d

In this case, the value it resulted in was 6. We have an initial value of 10, spawned 10 tasks each debiting $1 from the balance, so the final value should be $0. This means that our algorithm only worked 40% of the time! Very poor.

### Refactor for Thread Safety

Let’s refactor our Debit algorithm to be thread safe.

The easiest way would be to implement a lock() construct, as such:

private readonly object \_balanceLock = new object();

public void Debit(double amount)

{

lock (\_balanceLock)

{

double newBalance = Balance - Math.Abs(amount);

if (amount > Balance)

{

throw new ArgumentOutOfRangeException("amount", amount, DebitAmountExceedsBalanceMessage);

}

if (newBalance < 0)

{

throw new ArgumentOutOfRangeException("amount", amount, DebitAmountLessThanZeroMessage);

}

System.Threading.Thread.Sleep(0);

Balance = newBalance;

}

}

<http://www.informit.com/articles/article.aspx?p=1231461>

<http://msdn.microsoft.com/en-us/library/c5kehkcz.aspx>

Executing the unit test will now result in a pass, 100% of the time!

Keep in mind that we do pay a penalty for using the lock() construct. As each thread enters the Debit() method, they hit the lock, this will force that thread to sit and wait for the lock to be released. Essentially blocking all subsequent calls and preventing them from executing.

What about the credit method…?

If we had multi-threaded operations around both credit and debits, could this result in a bad account balance?

Lets write a test..

[Test]

public void Debit\_And\_Credit\_Multi\_Threaded()

{

// arrange

double beginningBalance = 10;

double debitAmount = -1;

double creditAmount = 1;

double expected = 10;

BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

// act

List<Task> tasks = new List<Task>();

for (int i = 0; i < 10; i++)

{

tasks.Add(Task.Run(() => account.Debit(debitAmount)));

tasks.Add(Task.Run(() => account.Credit(creditAmount)));

}

Task.WaitAll(tasks.ToArray());

// assert

double actual = account.Balance;

Assert.AreEqual(expected, actual, 0.01, " Account balance is not correct");

}

On my solution, this fails with a bad balance once out of every 3 or 4 calls. Again, we are not thread safe. How would you fix this situation?

Lets lock down the Credit call as well.

public void Credit(double amount)

{

lock (\_balanceLock)

{

System.Threading.Thread.Sleep(10);

Balance += Math.Abs(amount);

}

}

Notice that it is find to re-use the same instance variable, \_balance lock.