CSC3045

Agile & Component Based Development using .NET

Unit Testing & Test Driven Development

School of Electronics, Electrical Engineering & Computer Science

Queen's University, Belfast

Dr Darryl Stewart

Overview

- What are Unit Tests?
- Using Nunit
- Test Driven Development
  - What is it?
  - How is it done?
  - Why bother?

What are Unit Tests?

THE CLASSIC DEFINITION

A unit test is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A "unit" is a method or function.

- The code being tested is often called ***SUT*** (System Under Test)
- The code that tests the SUT resides in a ***Test Method***

What are Unit Tests?

- This means that you write programs that test the public interfaces of all of the classes in your application
- This is **not** acceptance testing
- It is testing to ensure the methods you write are doing what ***you*** expect them to do

What are Unit Tests?

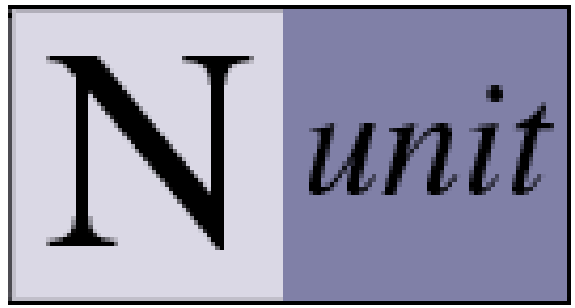
- Traditionally this was carried out by Quality Assurance engineers
- Carried out after the code was completed by the developers
- They used complicated scripting languages and large testing engines

- Now testing is often carried out by the developers
- In XP the tests are written before the code is written – Test Driven Development

- For this to be practical, a toolkit is needed that lets developers write their tests using the same language and IDE that they are using to develop the application – this is called a Testing Framework

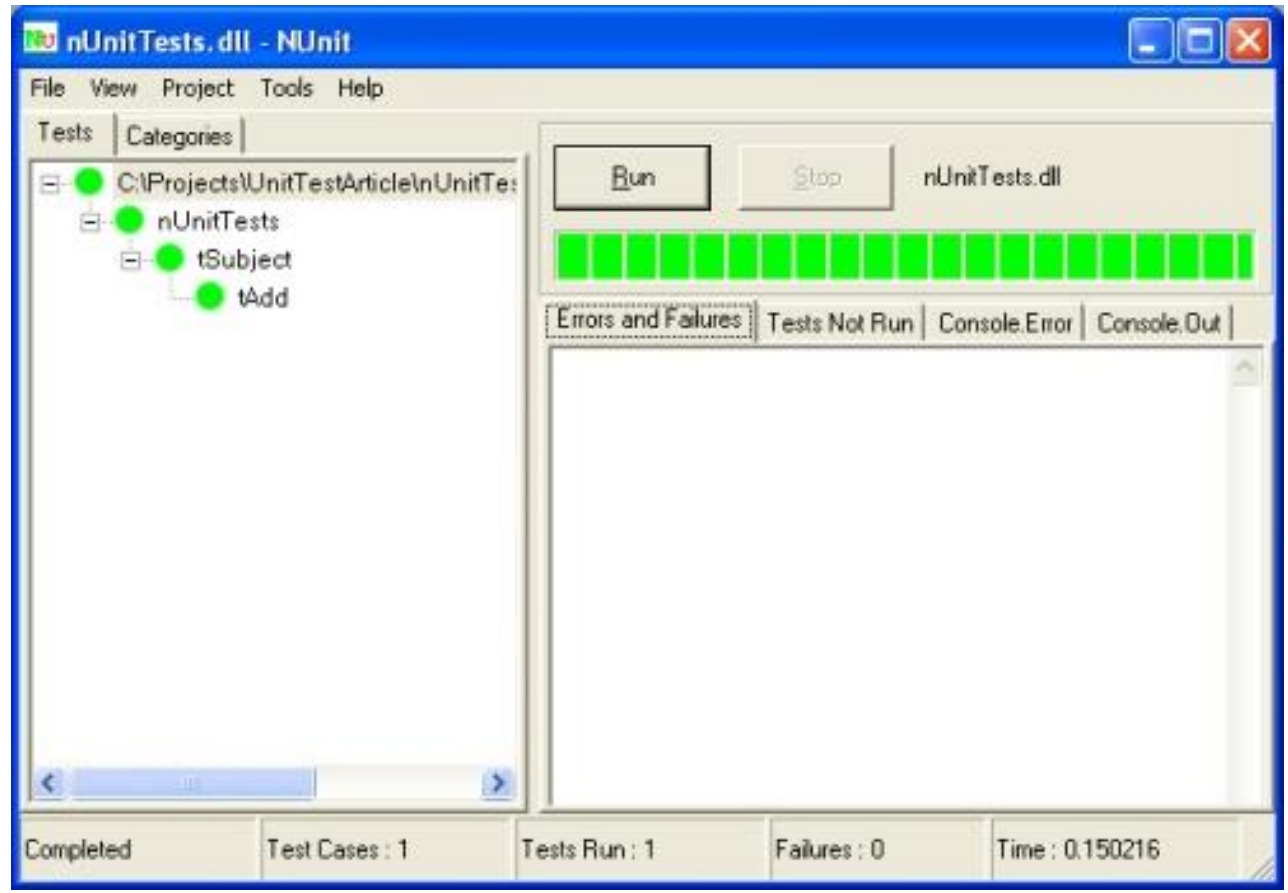
What are Unit Tests?

- Most modern Unit Testing Frameworks are derived from the framework created by Kent Beck for the first XP project, the Chrysler C3 Project
- It was originally written in Smalltalk but has been ported to many new languages including Java (jUnit), C++, VB, Python, Perl and more
- Was the defacto standard testing framework for .NET code has been NUnit although other .NET testing frameworks are available and Visual Studio now has built in support for unit tests
- We will be using NUnit so that your experience will be more general and portable
- NUnit is free to download – Google it



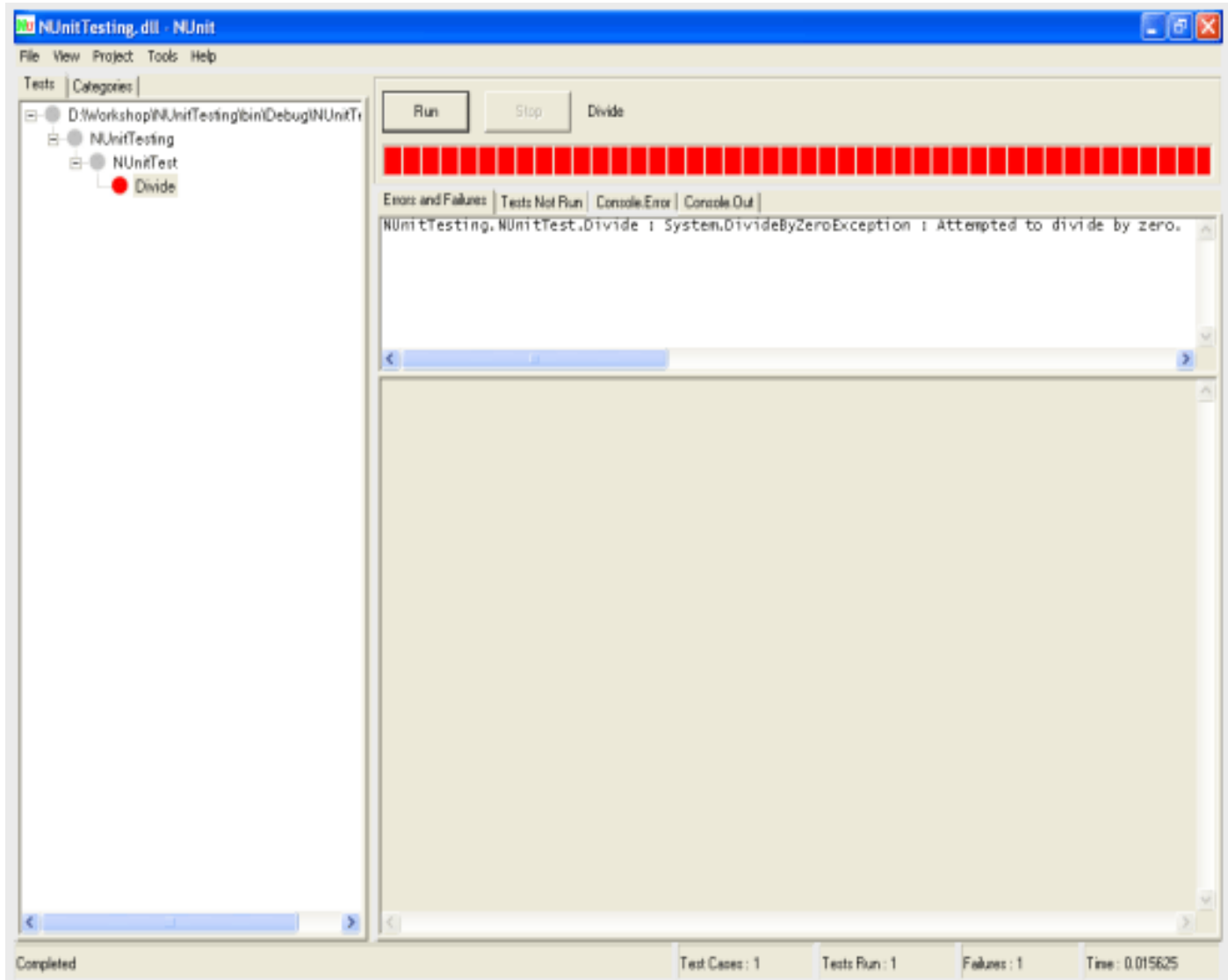
NUnit

- NUnit provides a Test Runner application which will examine your compiled code looking for specific "attribute" features that tell it that the code is test code
- The application then runs the tests that it finds
- You get a green bar if the tests all pass



NUnit

- If any of the tests fail then you get a red bar
- Code should only be committed back to the code repository **when all tests pass**



Calculator Example

- The following (fairly pointless) class is inside the CalculatorLib class library and needs to be tested

```
public class calculator
{
        public int Add(int n1, int n2)
        {
                return n1 + n2;
        }
        public int Subtract(int n1, int n2)
        {
                return n1 - n2;
        }
        public int Multiply(int n1, int n2)
        {
                return n1 * n2;
        }
        public int Divide(int n1, int n2)
        {
                return n1 / n2;
        }
}
```

Getting started with NUnit

Preparing an NUnit test class

1. Create a C# project of type Class Library (this may have already been done)
2. Go in the "solution explorer" and *Add Reference* to the NUnit.Framework DLL
3. Create a blank C# file

   e.g CalculatorTests.cs
4. Import the following namespace: **NUnit.Framework**
5. Create a stand alone class that uses the "TestFixture" attribute

```
[TestFixture]
public class CalculatorTest
{
 }
```

Getting started with NUnit

Writing an NUnit test

1. For every test you'd like to run, create a public method that returns void, takes no arguments and uses the "Test" attribute

```
[Test]
public void Test_Addition() { ... }
```

2. Instantiate the object you'd like to test inside the test method from the previous step

e.g.

```
Calculator calc = new Calculator()
```

Getting started with NUnit

3. Invoke the method(s) you'd like to test and assign the return values to local variables

e.g.

```
int answer= calc.Add(4,6);
```

4. Check the method's return value by using one of NUnit's assert methods. You can get a list of all the assert methods from the NUnit Documentation.

e.g.

```
Assert.AreEqual(10,answer);
```

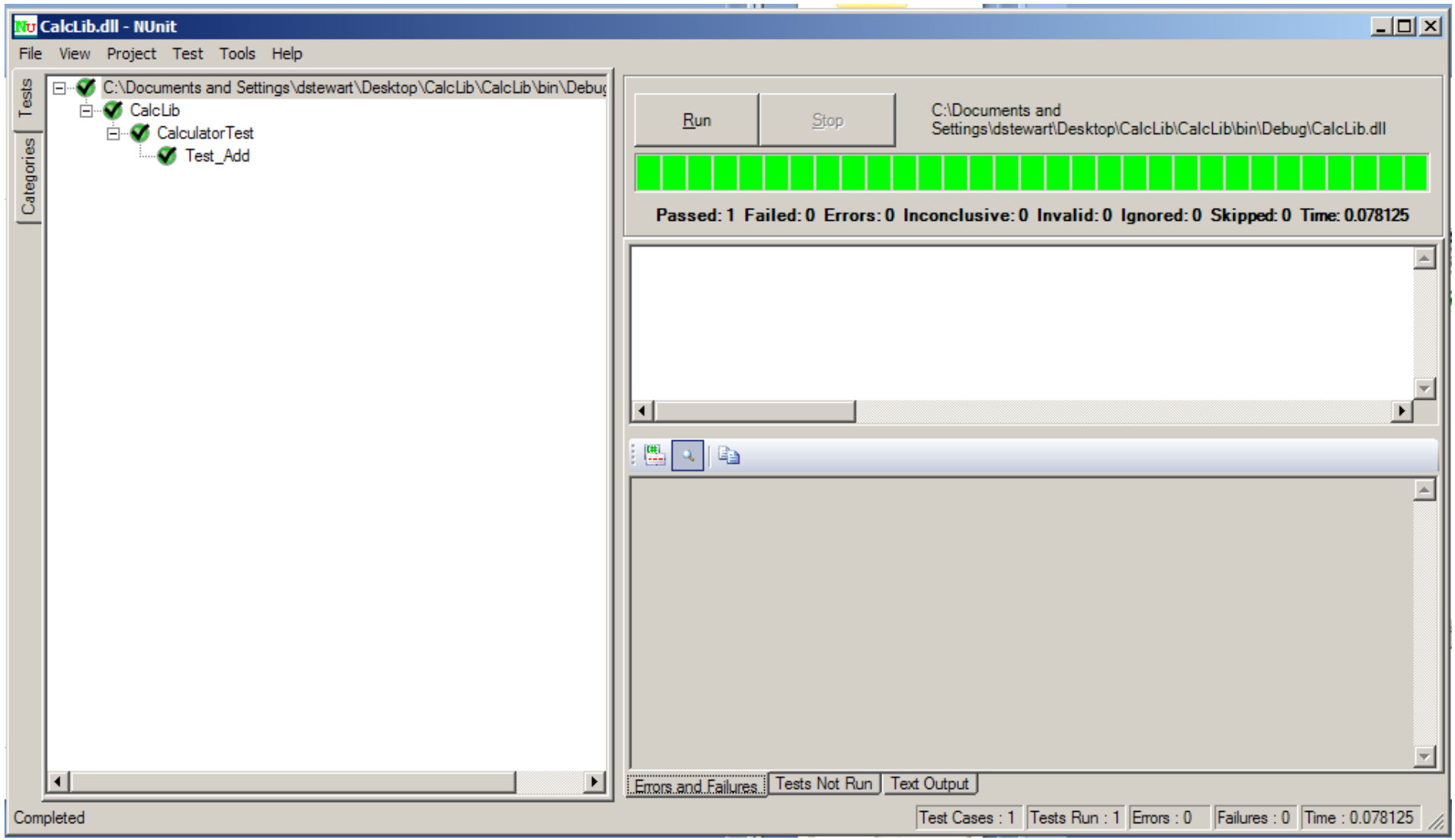
- We then add a test Class called CalculatorTest to our class library
- Add a test Method to the class to test **one** function of the Calculator class

```
using ...
using NUnit.Framework;

namespace CalcLib
{
    [TestFixture]
    class CalculatorTest
    {
        [Test]
        public void Test_Add()
        {
            calculator calc = new calculator();
            int answer = calc.Add(4, 6);
            Assert.AreEqual(10, answer);
        }

    }
}
```

- When the class library is compiled and NUnit is run to test the **dll**, we see that the test has been picked up and executed and it has passed



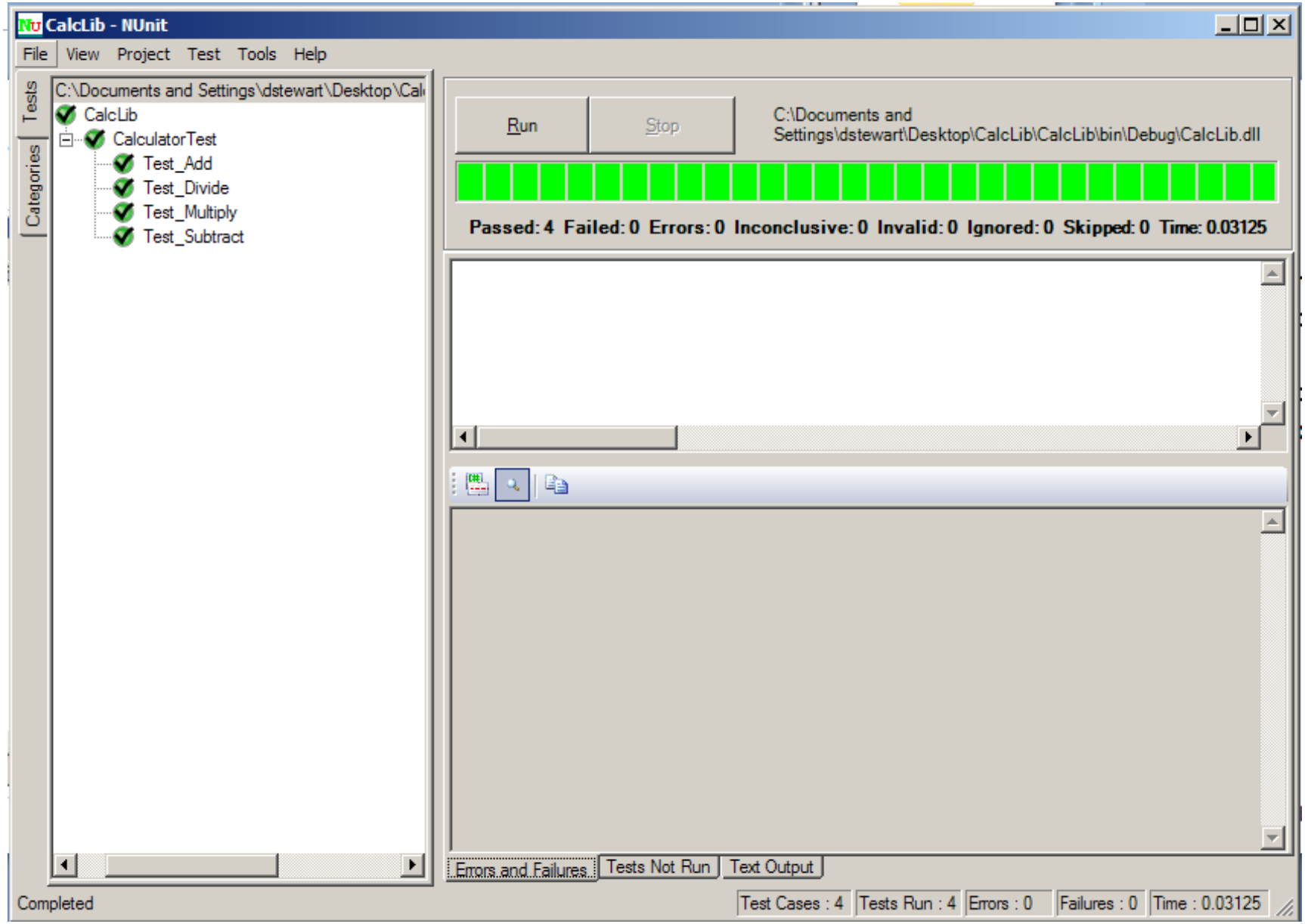
- We then add some more test methods to test the other functions of the Calculator class

```
[Test]
public void Test_Subtract(){
    calculator calc = new calculator();
    int answer = calc.Subtract(2, -2);
    Assert.AreEqual(4, answer);
}

[Test]
public void Test_Multiply(){
    calculator calc = new calculator();
    int answer = calc.Multiply(2, 2);
    Assert.AreEqual(4, answer);
}

[Test]
public void Test_Divide(){
    calculator calc = new calculator();
    int answer = calc.Divide(2, 2);
    Assert.AreEqual(1, answer);
}
```

The tests have all passed!!!! ☺



- The test class can be *refactored* slightly using the **Setup** attribute
- This **Setup** method is executed once before each of the other test methods
- There is also a TearDown attribute for tidying up after each test if needed

```
using ...
using NUnit.Framework;
namespace CalcLib
{
    [TestFixture]
    class CalculatorTest
    {
        calculator calc;
        [SetUp]
        public void Setup(){
              calc = new calculator();
        }

        [Test]
        public void Test_Add(){
            Assert.AreEqual(10, calc.Add(4, 6));
        }
        [Test]
        public void Test_Subtract(){
            Assert.AreEqual(4, calc.Subtract(2, -2));
        }
       ...
}
```

- Sometimes you will want to test that an **exception** will be thrown by a method in certain circumstances
- You do this by specifying the **expected exception attribute** along with the **Test** attribute

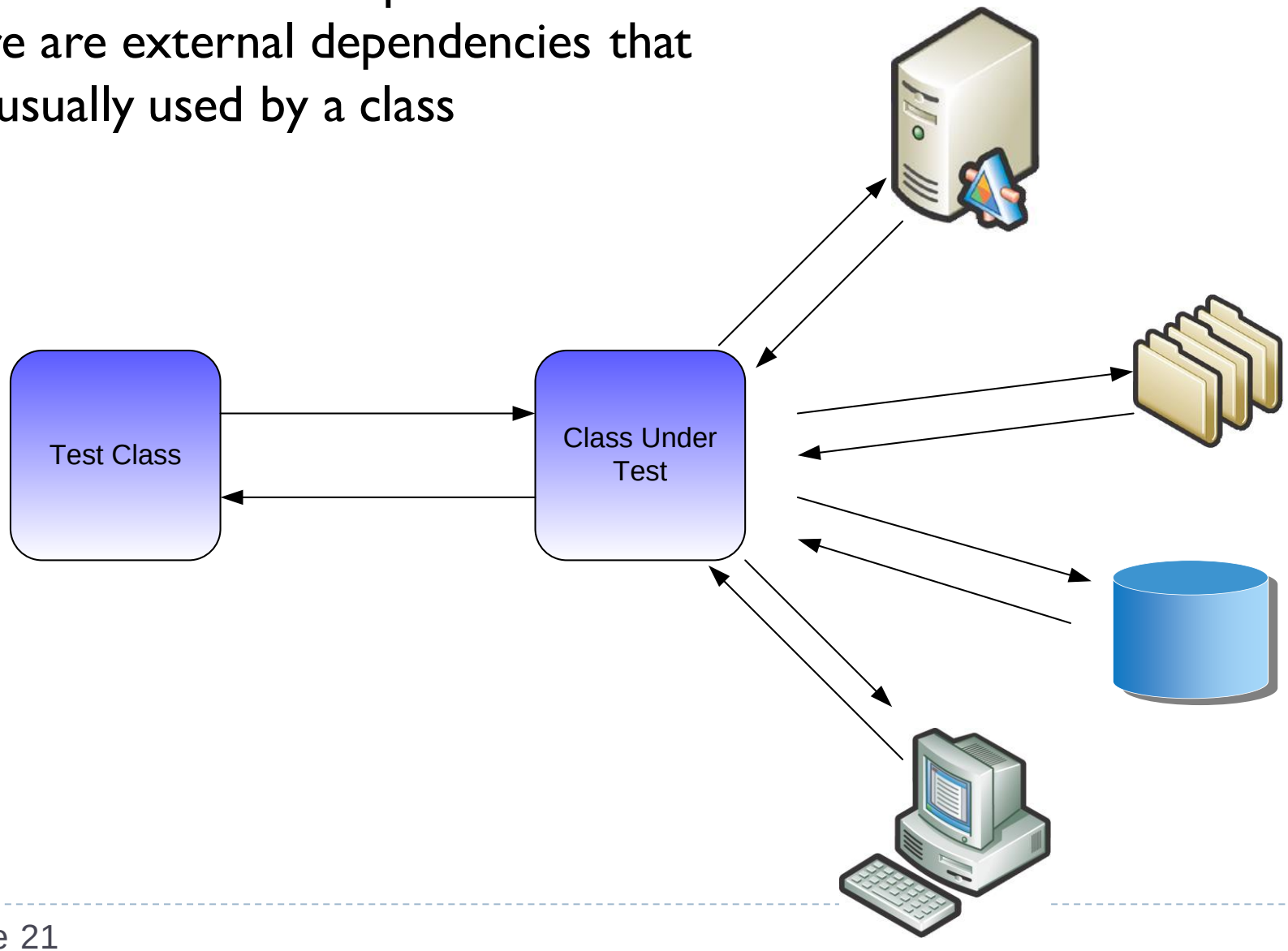
```
...
        [Test]
        public void Test_Multiply()
        {
            Assert.AreEqual(4, calc.Multiply(2, 2));
        }
        [Test]
        public void Test_Divide()
        {
            Assert.AreEqual(1, calc.Divide(2, 2));
        }
        [Test]
        [ExpectedException(typeof(OverflowException))]
        public void Test_AddShouldThrowOverFlowException()
        {
            Assert.AreEqual(1,calc.Add(int.MaxValue, 1));
        }
...
```

Demo

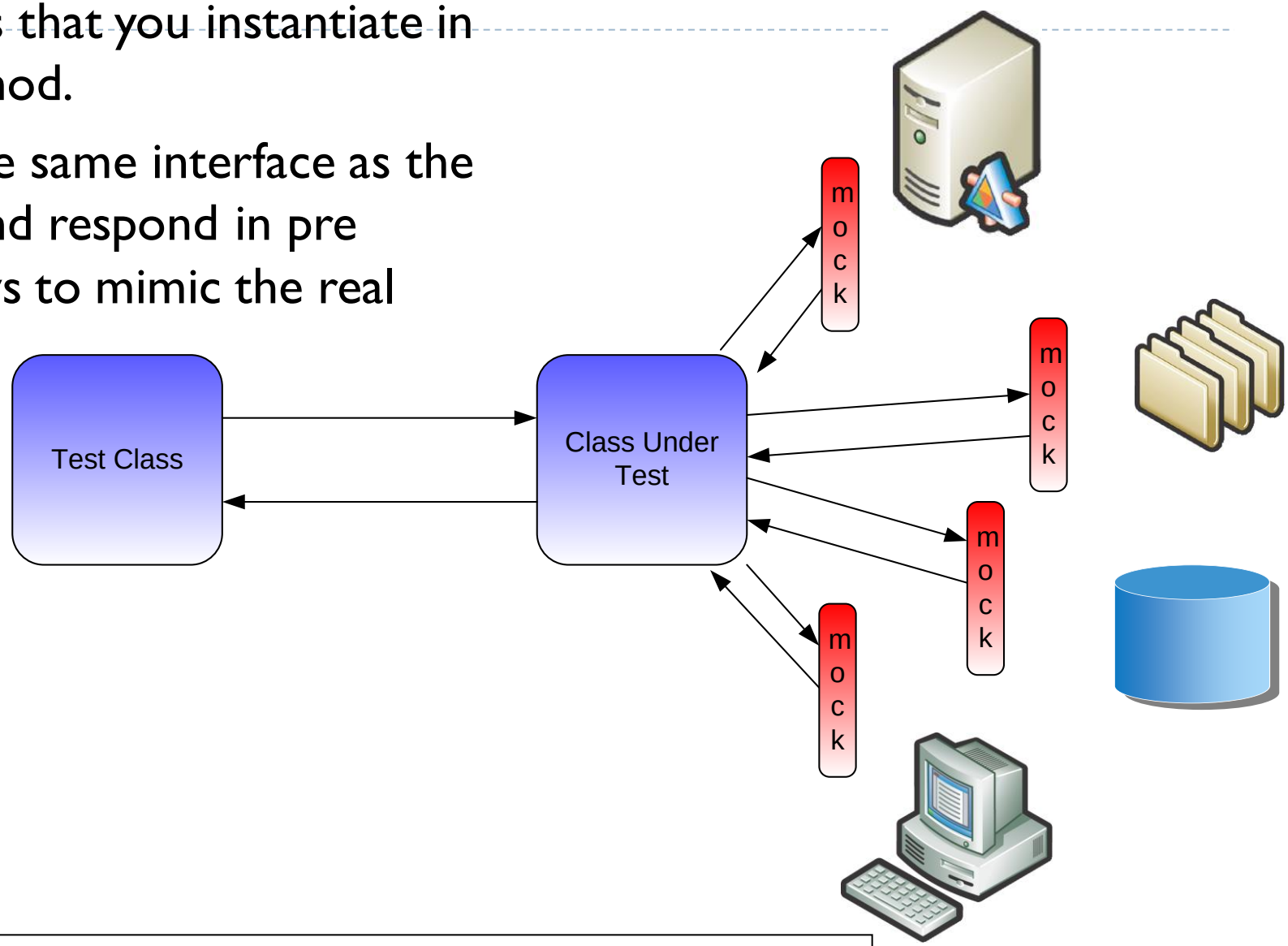
Removing Dependencies

- Testing using NUnit can be reasonably straightforward when the tests can be done in isolation

- It gets a bit more complicated when there are external dependencies that are usually used by a class



- One way to remove these dependencies is through the use of Mock objects that you instantiate in the test method.
- They have the same interface as the real object and respond in pre arranged ways to mimic the real object

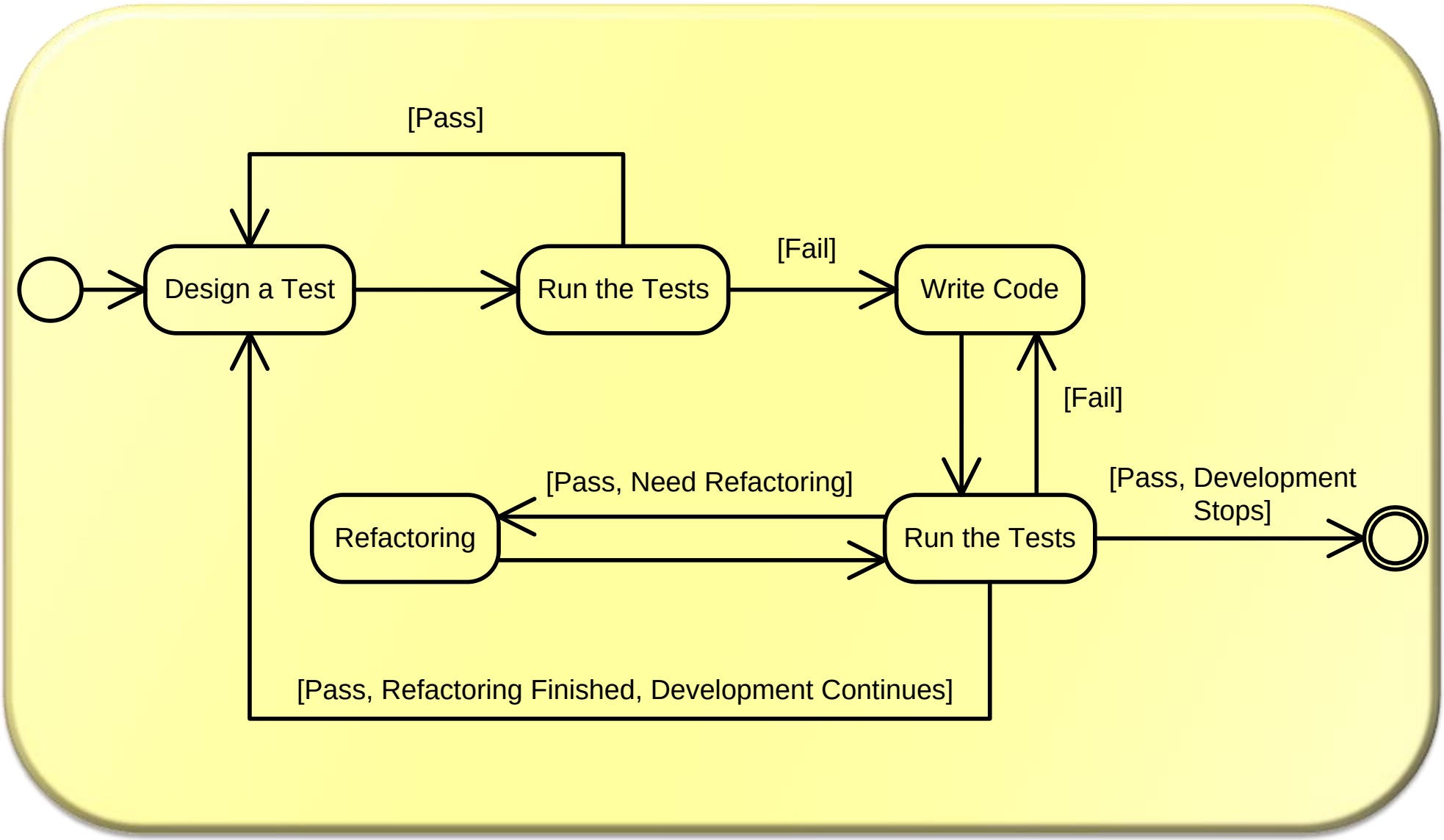


I recommend that you research and learn how this is done

Test Driven Development

- New test cases covering the desired improvement or new functionality are written first
- Then the production code necessary to pass the tests is implemented
- Finally the software is refactored to accommodate changes
- Considered by many as a method of **design** not just a testing method

TDD - The Process



Test Driven Development Cycle

RED GREEN REFACTOR

Step 1 Add a Test

```
using NUnit.Framework;
namespace CalcLib {
    [TestFixture]
    class CalculatorTest {
        calculator calc;
        [SetUp]
        public void Setup() {
             calc = new calculator();
        }
        [Test]
        public void Test_AddThreeIntegers() {
            Assert.AreEqual(1, calc.Add(-10,6,5));
        }
...
```

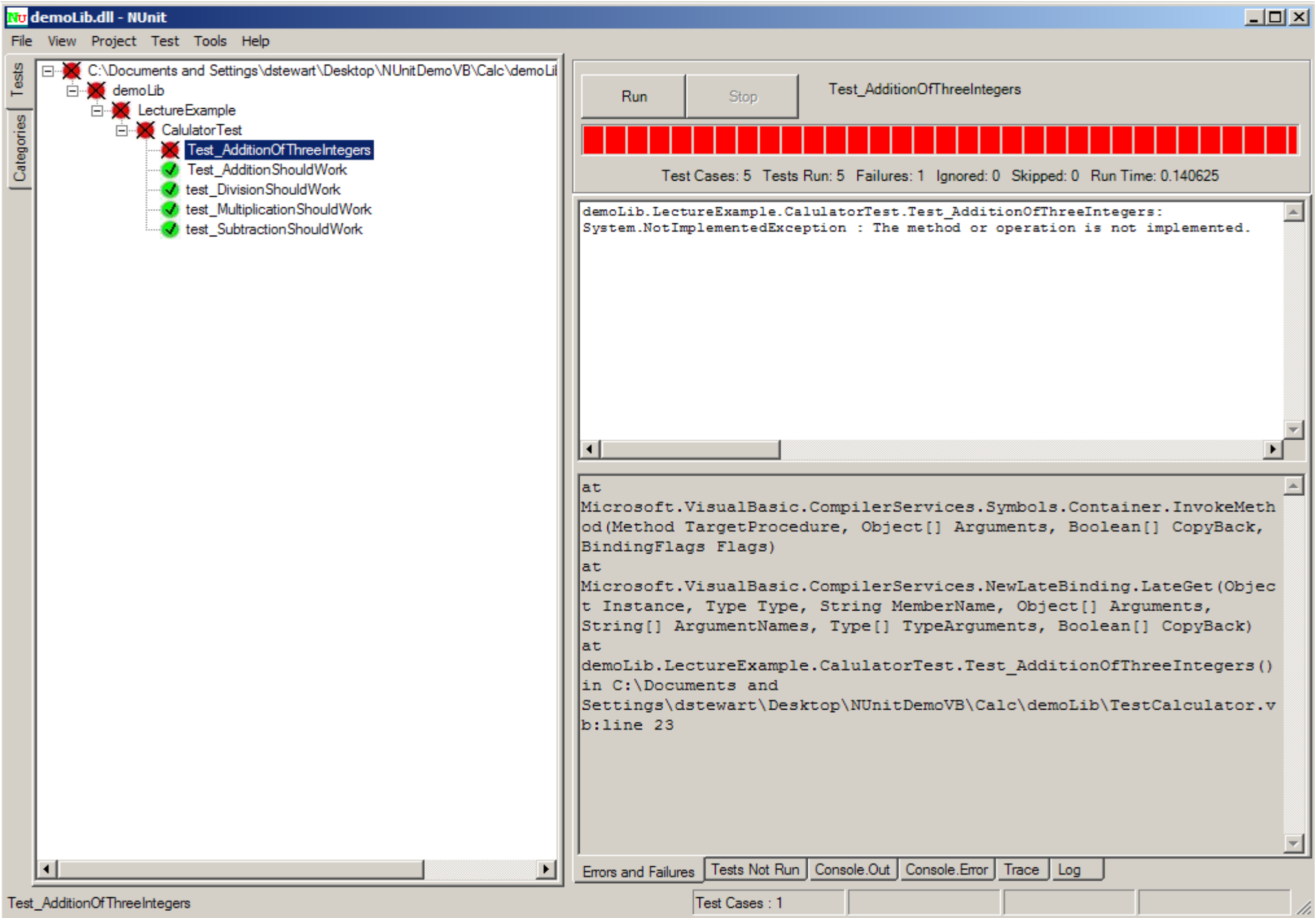
Test Driven Development Cycle

Optional step - Implement a Stub for this method

```
Public Class Calculator
{
...
      public int Add(int n1, int n2, int n3)
      {
            throw new NotImplementedException();
      }
...

}
```

- Run your test and watch it **Fail**
- Failure is good here – A passed test is NOT Good!


demoLib.dll - NUnit
File View Project Test Tools Help
Tests
Categories
C:\Documents and Settings\dstewart\Desktop\NUnitDemoVB\Calc\demoLil
demoLib
LectureExample
CalulatorTest
Test_AdditionOfThreeIntegers
Test_AdditionShouldWork
test_DivisionShouldWork
test_MultiplicationShouldWork
test_SubtractionShouldWork
Run
Stop
Test_AdditionOfThreeIntegers
Test Cases: 5 Tests Run: 5 Failures: 1 Ignored: 0 Skipped: 0 Run Time: 0.140625
demoLib.LectureExample.CalulatorTest.Test_AdditionOfThreeIntegers:
System.NotImplementedException : The method or operation is not implemented.
at
Microsoft.VisualBasic.CompilerServices.Symbols.Container.InvokeMeth
od(Method TargetProcedure, Object[] Arguments, Boolean[] CopyBack,
BindingFlags Flags)
at
Microsoft.VisualBasic.CompilerServices.NewLateBinding.LateGet(Objec
t Instance, Type Type, String MemberName, Object[] Arguments,
String[] ArgumentNames, Type[] TypeArguments, Boolean[] CopyBack)
at
demoLib.LectureExample.CalulatorTest.Test_AdditionOfThreeIntegers()
in C:\Documents and
Settings\dstewart\Desktop\NUnitDemoVB\Calc\demoLib\TestCalculator.v
b:line 23
Errors and Failures
Tests Not Run
Console.Out
Console.Error
Trace
Log
Test_AdditionOfThreeIntegers
Test Cases : 1

Test Driven Development Cycle

RED GREEN REFACTOR

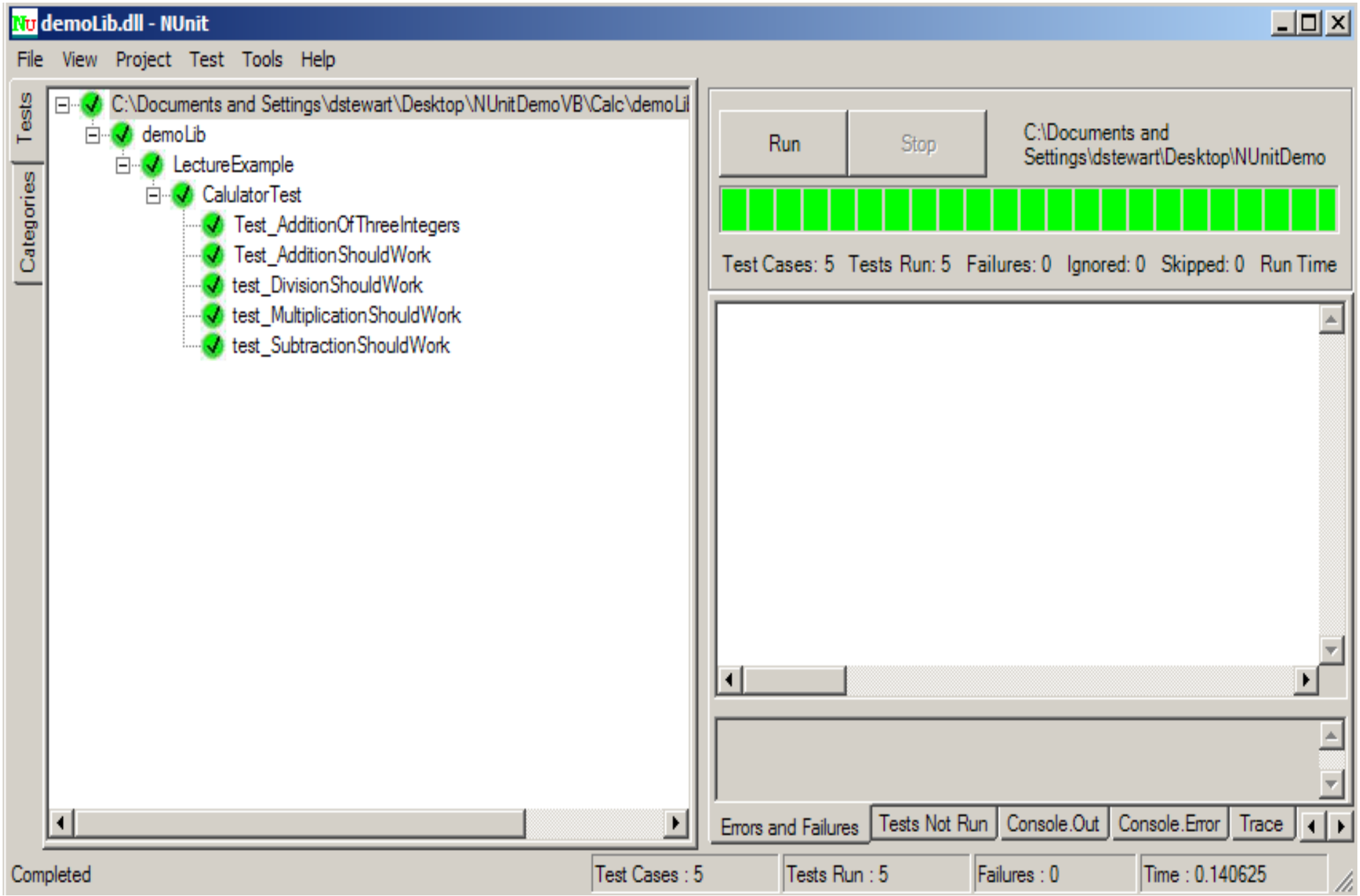
Step 2

- Write the simplest code possible to make the test pass

```
Public Class Calculator
{
...
      public int Add(int n1, int n2, int n3)
      {
            return checked((n1+n2+n3));
      }
...

}
```

- Run the tests again and hopefully it will pass…


demoLib.dll - NUnit
File View Project Test Tools Help
Tests
Categories
C:\Documents and Settings\dstewart\Desktop\NUnitDemoVB\Calc\demoLi
demoLib
LectureExample
CalulatorTest
Test_AdditionOfThreeIntegers
Test_AdditionShouldWork
test_DivisionShouldWork
test_MultiplicationShouldWork
test_SubtractionShouldWork
Run
Stop
C:\Documents and Settings\dstewart\Desktop\NUnitDemo
Test Cases: 5 Tests Run: 5 Failures: 0 Ignored: 0 Skipped: 0 Run Time
Errors and Failures
Tests Not Run
Console.Out
Console.Error
Trace
Completed
Test Cases : 5
Tests Run : 5
Failures : 0
Time : 0.140625

Test Driven Development Cycle

RED GREEN REFACTOR

Step 3

Refactoring involves changing the code to :

- Improve its readability
- Simplify its design and structure
- Basically make it more 'agile'

Some obvious things to look for and remove:

- Repetition – of code or strings
- Magic numbers
- Other "Code Smells"

Test Driven Development Cycle

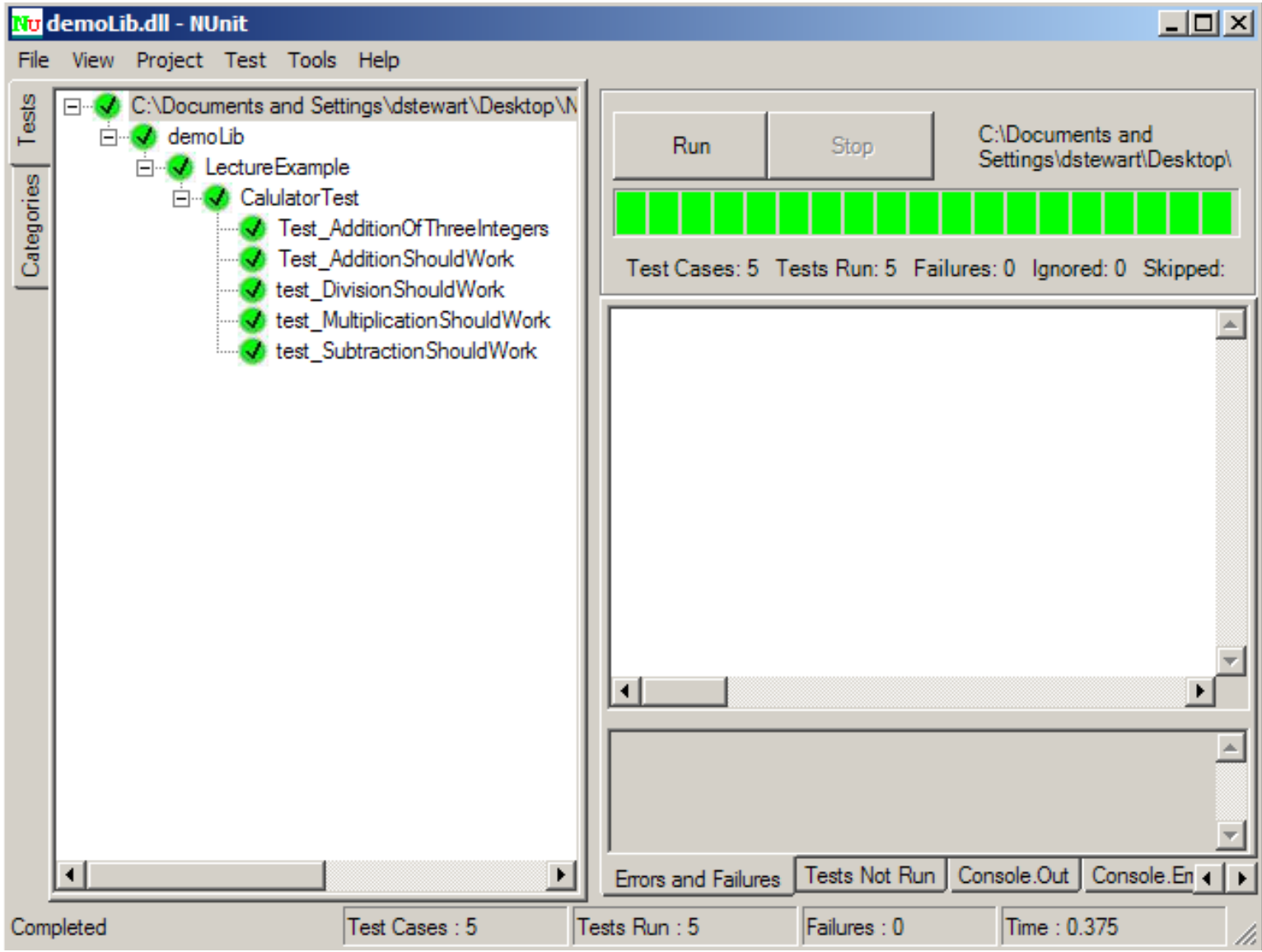
RED GREEN REFACTOR

Step 3

Refactoring example

```
Public Class Calculator
{
...
      public int Add(int n1, int n2, int n3)
      {
            return Add(n1,n2) + n3;
      }
...
```

}You can do refactoring with confidence because if you do break anything you will find out immediately from the tests



- Now start the cycle again by adding a new test for the next piece of functionality that is needed

What are the benefits?

- Only the code that is really needed is developed – neat, short, concise code
- The focus is on how the code will be used and that drives the development – fits with the YAGNI principle ("You aint gonna need it")
- Allows requirements changes and reshaping of systems with the confidence that the safety net of unit tests are there
- The unit test coverage is improved compared to a test-after process
- Fewer defects are created
- Defects are detected much sooner (minutes later rather than months later) which is the key to easy (cheap) bug fixing
- Encourages decoupling of code – better structured code
- Instant rewarding feedback to the developer – happy with their work!

Is it worth the effort?

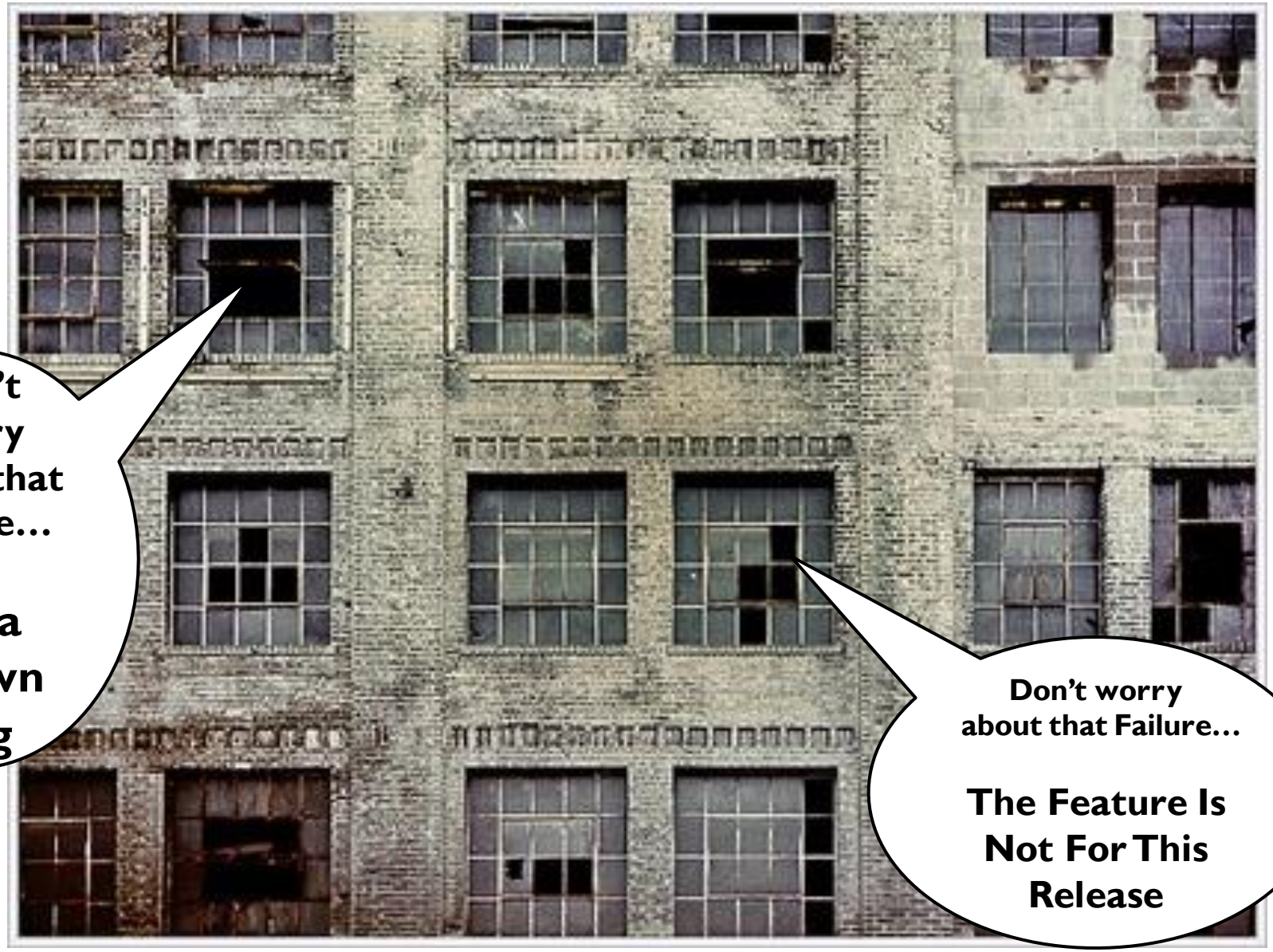
- Microsoft Case Study
  - TDD project has twice the code quality
  - Writing tests requires 15% more time
- IBM Case Study
  - 40% fewer defects
  - No impact on the team's productivity
- John Deere / Ericsson Case Study
  - TDD produces higher quality code
  - Impact of 16% on the team's productivity

Software Errors Cost U.S. Economy $59.5 Billion Annually – NIST Study

- Software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated **$59.5 billion** annually, or about 0.6 percent of the gross domestic product
- Over half of the costs are borne by software users and the remainder by software developers/vendors.
- Although all errors can't be removed, more than a third of costs, or an estimated **$22.2 billion, could be eliminated by an improved testing infrastructure** that enables earlier, more effective identification and removal of software defects

- These are the savings associated with finding an increased percentage (but not 100%) of errors **closer to the development stages in which they are introduced**.
- Currently, over **half of all errors are not found until "downstream"** in the development process or during post-sale software use.

Don’t Tolerate Broken Windows



What people have said about it

- “At first I didn't like that I needed to write tests for my code, but now after using it for more than 10 months I can‘t program without it.”
- “Helps to come up with better APIs.”
- “It gives confidence that our software is working well at all times. Even after making major changes and/or changing software we are dependent on.”
- “Productivity increases - you might loose some when you make the initial tests, but you'll get it back later. The code covered by tests is 'insured' against future changes.”
- “It works well for libraries, not so well for GUI applications.”
- “Some things cannot be tested easily like server errors, unless you use mock-objects.”

Recap

- Unit tests improve code quality
- They aid refactoring and code changes by providing a safety net – highly relevant in agile development
- Select your test cases carefully
- Remove dependencies in tests using mocking
- TDD is an XP practice
- Follows the **RED GREEN REFACTOR** cycle
- Focuses the developer on the requirements and on the interface
- Leads to improved code structure
- May take longer to develop same functionality
- The quality will be higher and hence this practice will provide better value in the long run
- Requires patience to learn – but once learned it becomes normal and habitual

Would strongly encourage you to practice this – persist after the module is over