

Code Coverage and Test-Driven Development using NCover, NCoverExplorer, NUnit and TestDriven.NET

by Craig Murphy

Over the course of this article I plan to demonstrate a number of tools and topics that encompass “testing”. I’ll be looking at code coverage – how much of our code is “exercised” or used. I’ll be looking at tools that can help us with code coverage. I’ll focus on using NUnit for testing and will demonstrate how we can tie it into the code coverage activity. Finally, I’ll be looking at how we can integrate all of this good stuff into the Visual Studio IDE.

As some of you may be aware, I changed jobs in February. Whilst I may have left behind a lot of Delphi code and a lot of internal clients who believed that software is an asset (and it is), sadly I found myself doing less and less real development and more and more administration and maintenance. So, with a new job, a slightly different toolset and some new application domains to get my teeth into, things are looking up. Anyway, I’m sorry to say that my new toolset does not include Delphi. My new toolset comprises Visual Studio 2005, C#, TestDriven.NET, NUnit, NCover and NCoverExplorer, some of which I have been using for a long time now.

I discovered NCover and NCoverExplorer via a couple of blog posts and was suitably impressed – I am always on the lookout for ways of ensuring that my applications are as well tested as they can be. After all, there is nothing worse than a stream of phone calls from your users complaining about a show-stopping crash or feature that does not appear to work. With careful use of the tools mentioned above, we can ensure that our applications are tested and that we have no code that is unused. Code that is unused is often the source of bugs or feature failures. In the past, without tests and without code coverage tools, we had to resort to using a debugger to test all paths through our code, which was a laborious process fraught with repetition and boredom.

Over the course of the article I will go through the following:

1. The creation of a simple class that does something trivial, in this case a calculator (Visual Studio, C#).
2. The creation of a simple front-end application that uses the calculator (Visual Studio, C#).
3. An examination of code coverage using the front-end application (NCover and NCoverExplorer).
4. An examination of code coverage using unit tests (NUnit).

These four topics will demonstrate two things. Firstly, the benefit of using a code coverage tool to help you learn more about your application and the way that it works. Secondly, how that added benefit of a set of unit tests coupled with a code coverage tool can yield increased levels of confidence in your application’s testing strategy. Of course, prior to code coverage tools being automated, the simplest form of code coverage was the debugger: even as recently as 1998 I recall laboriously slaving over the Delphi debugger whilst I was “testing” my application. It was worth the toil, the few additional bugs that came to light meant that the application had years of trouble-free use. Now, with code coverage integrated into the IDE, and with unit tests sitting side-by-side with the application source code, the time required to run the tests and perform a code coverage analysis is so short, it can be done with every build.

The Toolset – an overview

Let’s get the obvious bits out of the way first. Visual Studio 2005 is the IDE and C# is the development language.

Whilst I am going to be touching on unit testing (using NUnit) in this article, you don’t need to be practising test-driven development (TDD) or be using NUnit to benefit from the likes of NCover and NCoverExplorer – the combination of these tools and the service that they provide in their own right is very useful. Thus it is possible to use NCover and NCoverExplorer in the absence of TestDriven.NET and NUnit.

However, I am trying to sell you the whole package: unit testing and code coverage. Your takeaway from this article should be “how well have your tests exercised your code?” If you believe that your tests should touch every single line of code, i.e. 100% coverage, using a combination of NUnit, NCover and NCoverExplorer

will help you determine the percentage of coverage your tests are actually achieving. In reality, it is unlikely that you will achieve 100% coverage – empirical evidence seems to present a figure of 85% coverage being considered acceptable, a point that I will revisit in the Conclusions section of this article.

TestDriven.NET

TestDriven.NET provides an integration layer between a number of testing frameworks, such as NUnit, and the IDE. Before TestDriven.NET, we would write our tests, write some code, build everything and then we would fire up a separate application that would run the tests (e.g. the NUnit front-end). Generally this is/was fine as the benefits of test execution greatly outweighed the use of a secondary application. Whilst the NUnit front-end allows us to choose which tests we want to run (as opposed to running all tests), we still find ourselves leaving the IDE and jumping into another application.

So, in addition to integrating NUnit into the Visual Studio IDE, it also provides integration with NCover and NCoverExplorer.

NCover

NCover is a command-line tool that scans your application's code and records information about which lines of code were executed. NCover uses the "sequence points" that the compiler generates and stores in the .pdb file (debug information) to determine which lines of code are executed and how frequently. Without getting bogged down in detail, a sequence point is essentially a single program state or a line of code.

NCover's command-line syntax is:

```
Usage: NCover /c <command line> [/a <assembly list>]

/c Command line to launch profiled application.
/a List of assemblies to profile. i.e. "MyAssembly1;MyAssembly2"
/v Enable verbose logging (show instrumented code)
```

After NCover has analysed your code, it creates two files: coverage.log and coverage.xml. The .log file contains the events and messages that NCover creates as it analyses your code base. The .xml file is where it all happens: it contains NCover's analysis of your code base. There is a third file, coverage.xsl. It's an XSL stylesheet that takes coverage.xml as its input, allowing it to be displayed in a neat tabular fashion inside Internet Explorer.

NCoverExplorer

Whilst the output of NCover is reasonably presentable inside Internet Explorer, it is necessary manually to locate any code that NCover flags as being untouched. With NCoverExplorer installed, not only do we receive a pleasant treeview depicting the organisation of our code, but we are able to see the code, colour-coded too! I've included a screenshot of NCoverExplorer in action; read on to find out how to create it.

First Steps

For the purposes of this article, I am going to do and explain things in a slightly out of order fashion. I know that I have mentioned test-driven development already, and by rights we should be developing our application in that fashion. However, I would like to introduce code coverage first. Luckily, the example that I plan to use is so simple it could almost be tested without the need for test-driven development. The example to demonstrate code coverage is that of a simple calculator – I could have been more original: I apologise for my banality!

The Calculator Class

Implementing a simple calculator isn't rocket science, but since I need you to be at least familiar with the code layout, here's the code that I am using:

```
namespace ClassUnderTest
{
    public class MyCalculator
    {
        public int subtract(int a, int b)
```

```

    {
        return a - b;
    }

    public int add(int a, int b)
    {
        return a+b;
    }

    public int divide(int a, int b)
    {
        return a / b;
    }

    public int multiply(int a, int b)
    {
        return a * b;
    }
}

```

Using The Calculator

Purely for the purposes of this article, I have used the namespace `ClassUnderTest` to house my calculator class. Obviously this isn't the namespace of choice for production classes, so please feel free to change it to suit your environment.

I created a small application that lets us use the calculator class. Again, it's not a work of art, but it does the job. We are getting closer to the good stuff, please stay on board!

Figure 1 presents a screenshot of the application, with apologies to Dr.Bob:

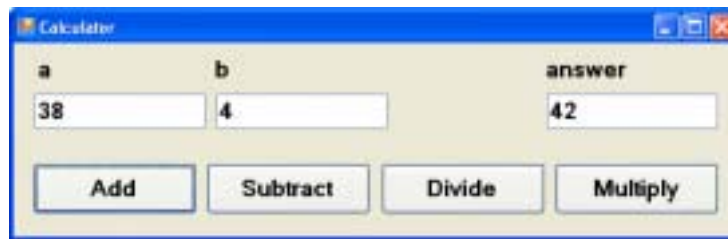


Figure 1 – the simple calculator in action

Here's the abbreviated code that I am using to implement the "add" functionality:

```

using ClassUnderTest;

namespace CalcApp
{
    public partial class FormMain : Form
    {
        private MyCalculator calc = new MyCalculator();

        private void btnAdd_Click(object sender, EventArgs e)
        {
            int answer;
            int a, b;

            // for demonstration purposes only
            a = System.Convert.ToInt32(tbA.Text);
            b = System.Convert.ToInt32(tbB.Text);

            answer = calc.add(a, b);

            tbAnswer.Text = answer.ToString();
        }
    }
}

```

So, having compiled `CalcApp`, we can submit the `.exe` to NCover. NCover's output:

```

C:\Program Files\NCover>NCOVER.CONSOLE
    "...dev\VS2005\Test\DDG\CalcApp\CalcApp\bin\Debug\CalcApp.exe"

NCover.Console v1.5.4 - Code Coverage Analysis for .NET - http://ncover.org
Copyright (c) 2004-2005 Peter Waldschmidt

Command: ...dev\VS2005\Test\DDG\Calc
App\CalcApp\bin\Debug\CalcApp.exe
Command Args:
Working Directory:
Assemblies:
Coverage Xml: Coverage.Xml
Coverage Log: Coverage.Log

Waiting for profiled application to connect...
***** Program Output *****
***** End Program Output *****

```

Now, you will have noticed that I have only implemented the “add” calculation. This is deliberate as I need to use the missing calculations to demonstrate code coverage.

Coverage.xml is too large to reprint here and it’s not all that easy to read. Fortunately, NCover’s author realised this and created an XSL (stylesheet) that transforms the XML into something a) more readable and b) more useful. Figure 2 presents a snapshot of that output – notice the green bars and red bars, we’re clearly in the ‘testing’ domain now.

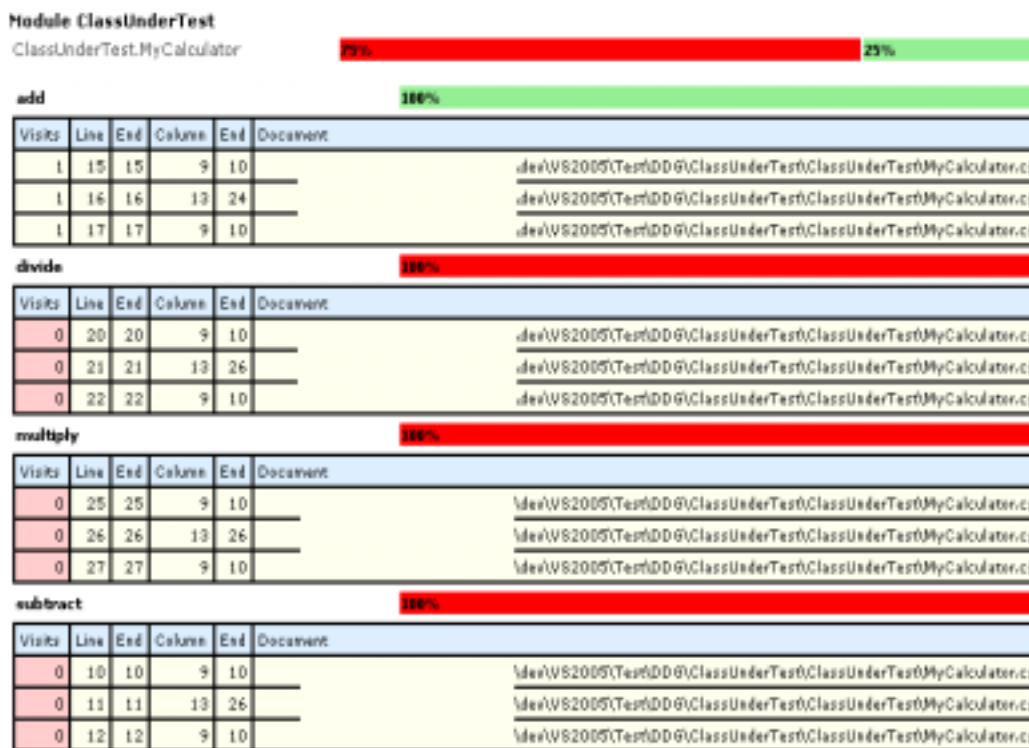


Figure 2 –F’s stylesheet creates a neat representation of Coverage.xml

We can see from Figure 2 that MyCalculator only enjoys 25% code coverage. This is made up from the fact that divide, multiple and subtract account for 75% of the class’s functionality, and they aren’t used (exercised) at all. I know it’s very obvious in this small example, however imagine a large production system with hundreds if not thousands of classes – even with rigorous manual testing, it can be very easy to forget to hook up a method call that results in a piece of code being unused. Of course in the case of this simple example, we’ve clearly not put any code behind the subtract, divide and multiply buttons!

From this report, Figure 2, we can easily see that we’ve only covered 25% of the MyCalculator class. However, and this is a key point, in order to get this far, we had to perform manual testing. We had run the application through NCover such that it could watch what was happening. We had to enter the number 38 and 4 and we had to move the mouse and click on the Add button. Whilst this is a better than stepping through code with the debugger, it’s not automated therefore it’s not repeatable.

NCoverExplorer

Moving things a step further down the line, NCover's output, whilst nice, is greatly enhanced if you pass it to NCoverExplorer. Figure 3 presents NCoverExplorer viewing the recently created Coverage.xml file.

There are few interesting facets in Figure 3. Firstly, we can see our code in the code view pane. Secondly, it is colour coded: red if the code hasn't been exercised and blue if it has. There's also a nifty right-click context menu that offers us the chance to edit the given code inside Visual Studio.net – although part of me thinks that we should be able to double click on the code to invoke this function rather than use a menu.

The treewiew on the left-hand side is also rather useful. It groups all the coverage items together, letting us drill down into those items that are of interest, usually those classes that have low coverage such as MyCalculator. It's also possible to right click on the treeview where you can find a "Remove from Results" menu option – this can be useful if you just want to focus on specific items, perhaps with very low coverage.

Code coverage comes into its own when it reveals code that you previously thought was exercised and tested, more so if we combine unit testing using frameworks such as NUnit with code coverage tools such as NCover and NCoverExplorer. But it gets better: if we use TestDriven.NET as well, we can have all this functionality integrated within the Visual Studio IDE.

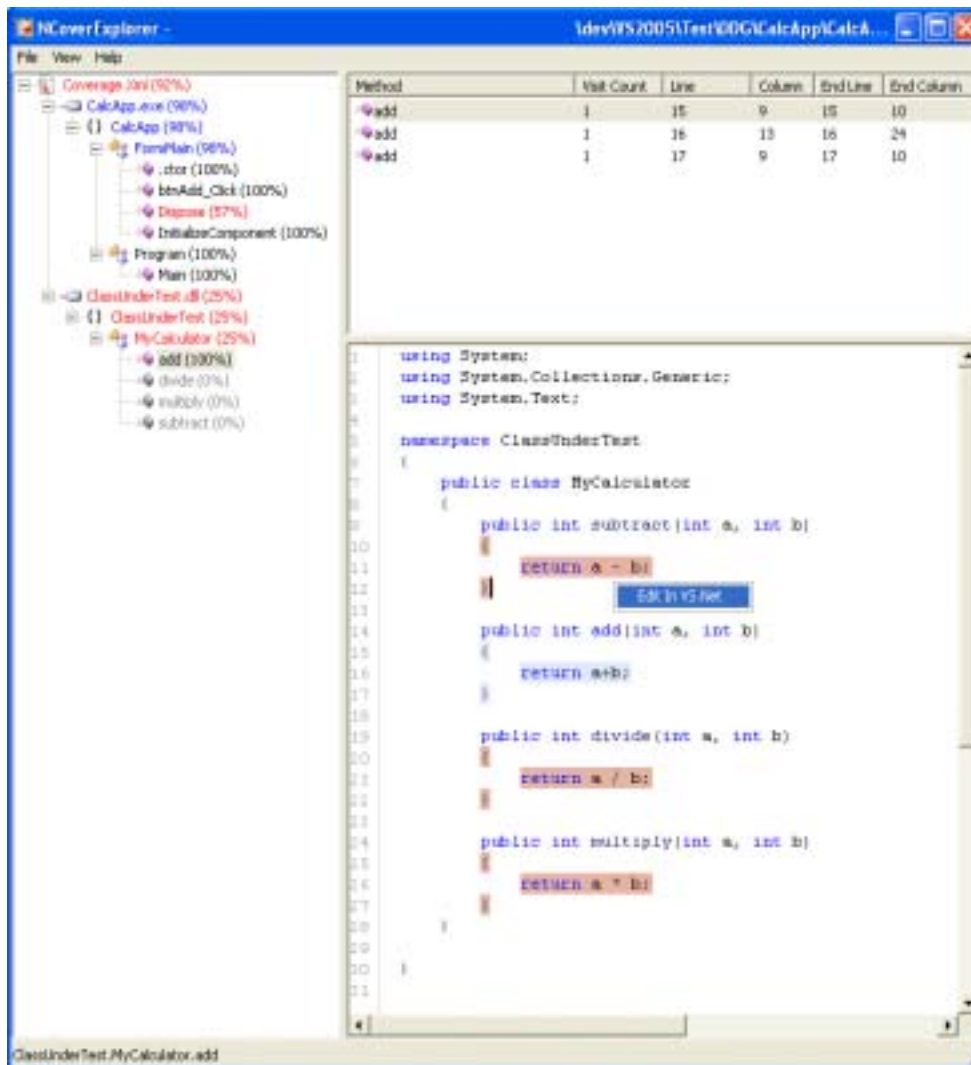


Figure 3 – NCover's output as viewed using NCoverExplorer

IDE Integration

Without going into too much detail about test-driven development and NUnit (further information can be found in the Resources section of this article), our next step is to prepare a new class that allows us to test the MyCalculator class. In reality, we would have written our tests before writing the MyCalculator class: we're doing things in a slightly different order for the sake of this article.

So, using Visual Studio we simply add a new Class Library to our solution, add a reference to the ClassUnderTest namespace and we're off. The following code demonstrates how we might use the NUnit testing framework to exercise MyCalculator. Whilst not an NUnit requirement, I prefer to name my test methods with the prefix "test": other unit testing frameworks may vary. As you can see, we're simply recreating the manual test that we performed using the desktop application and we're still only testing the "add" method.

```
using NUnit.Framework;
using ClassUnderTest;

namespace TestLibrary
{
    [TestFixture]
    public class MyTests
    {
        [SetUp]
        public void Test_SetUp()
        {
            MyCalculator calc = new MyCalculator();
        }

        [Test]
        public void testAdd()
        {
            int n = calc.add(38, 4);
            Assert.AreEqual(42, n);
        }

        [TearDown]
        public void Test_TearDown()
        {
        }
    }
}
```

How do we invoke this test? Well, we have a number of options open to us. In the absence of TestDriven.NET, we could use the NUnit GUI, as shown in Figure 4. Alternatively, we could right click on the TestLibrary tests and use TestDriven.NET's NUnit integration to run the tests for us, thus we can run the tests from within the IDE.

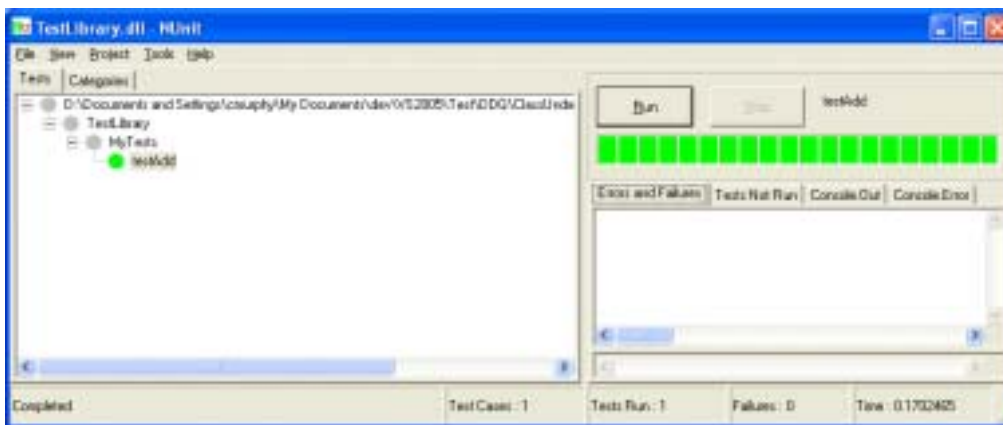


Figure 4 – NUnit in action

Figure 4 is fine; it uses the green bar/red bar notion to indicate success and failure. What it doesn't tell us is the fact that our test (testAdd) does not exercise MyCalculator well enough. For that, we need to go back to NCover to work out how much of the code (in MyCalculator) is exercised by the tests in TestLibrary.

I don't know about you, but I wasn't all that fussed about using the command-line for NCover. Would it not be great if it was integrated into the IDE? Luckily, that's what TestDriven.NET does for us – we can right click, use a context menu to run tests or choose the tool/framework that we would like to test with. Figure 5 presents the context menu offering us the chance to run these tests through "Coverage" which will, under-the-hood, invoke NCover and NCoverExplorer for us.

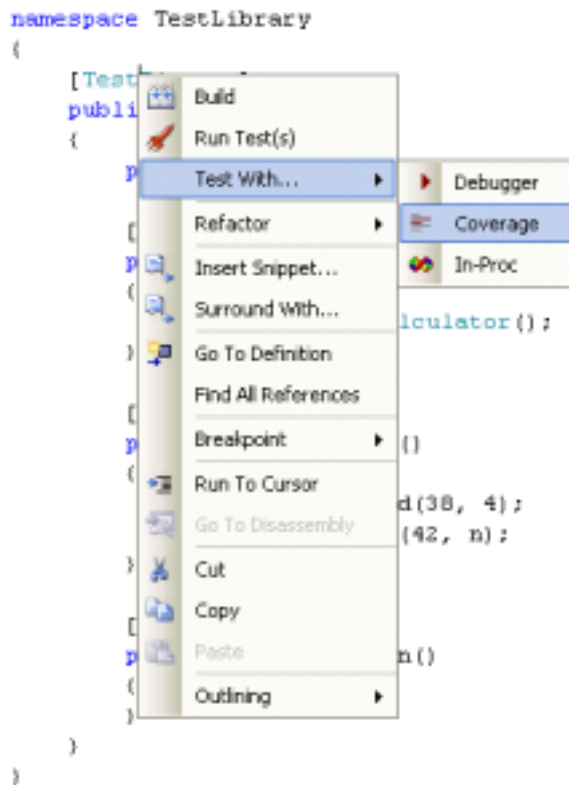


Figure 5 – Context menu courtesy of TestDriven.NET

After clicking on Test With → Coverage, we are presented with Figure 6. On the surface you might be thinking that Figure 6 is very similar to Figure 3. Graphically yes, it is. However the treeview on the left-hand side is subtly different. It is telling us that our tests in TestLibrary enjoy 100% coverage, whereas the MyCalculator is only benefiting from 25% coverage.

On the premise that we would normally write our tests (TestLibrary) before we wrote an application (CalcApp), the combination of NUnit with code coverage is a powerful addition to our development toolset. It puts us in the position of being able to write further tests that ensure MyCalculator enjoys 100% coverage (which can be achieved in this simple example), before we agree that MyCalculator is fit for consumption in a production application.

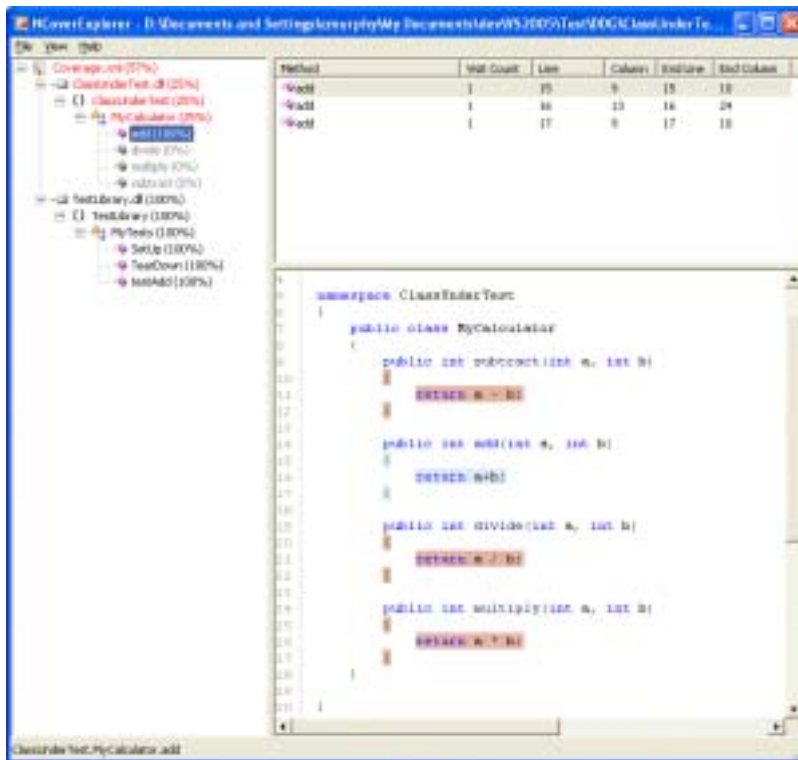


Figure 6 – NcoverExplorer: our tests don't exercise MyCalculator well enough!

The key takeaway from this Figure 6, and indeed this article, is the fact that we have automated our test and our code coverage: we can see in a single screenshot how well our tests are exercising our code.

But what about older versions of Visual Studio?

I know that I have mentioned Visual Studio 2005 a few times, and you are probably wondering if all this good stuff works with earlier versions of Visual Studio. Well, good news: TestDriven.NET supports all versions of Visual Studio (for .NET). It also integrates with all major unit-testing frameworks including NUnit, MbUnit and Microsoft's Team System. NCover does require .net 2.0 for its internal operation, but it can profile applications that target .net 1.1 and .net 2.0.

Conclusions

Sometimes combining a number of tools can pay dividends. In the case of NUnit, NCover, NCoverExplorer and TestDriven.NET it pays huge dividends. The ability to write tests and ensure that they exercise as much of the code being tested as is possible is a boon. You may strive for 100% code coverage, however I would refer you to the NCoverExplorer FAQ where the "satisfaction threshold" is discussed and reasons why 100% coverage might not be possible are mentioned.

Resources

- TestDriven.NET: <http://www.testdriven.net/>
- TestDriven.NET Google Group: <http://groups.google.com/group/TestDrivenUsers/>
- NUnit: <http://www.nunit.org/>
- MbUnit: <http://www.mertner.com/confluence/display/MbUnit/MbUnit+Home>
- Microsoft Team System: <http://lab.msdn.microsoft.com/vs2005/teamsystem/>
- Peter Waldschmidt.'s NCover: <http://www.ncover.org/>
- NCoverExplorer: <http://www.kiwidude.com/blog/>
- NCoverExplorer FAQ: <http://www.kiwidude.com/dotnet/NCoverExplorerFAQ.html>
- Test-Driven Development: a practical guide, Nov/Dec 2003, <http://www.prototypical.co.uk/pdf/test%20driven.pdf>
- Test-Driven Development Using csUnit in C#Builder, Jan/Feb 2004: <http://www.prototypical.co.uk/pdf/TDD.pdf>



Craig is an author, developer, speaker, project manager, Certified ScrumMaster and Microsoft Most Valuable Professional (Connected Systems). He specialises in all things XML, particularly SOAP and XSLT. Craig is evangelical about .NET, C#, Test-Driven Development, Extreme Programming, agile methods and Scrum. He can be reached via e-mail at: bug@craigmurphy.com, or via his web site: <http://www.craigmurphy.com> (where you can also find the source code and PowerPoint files for all of Craig's articles, reviews and presentations).