TDD with Microsoft.Activities.UnitTesting

[Microsoft.Activities.UnitTesting](http://wf.codeplex.com/) is a unit testing framework designed to make it easier to unit test activities and workflow services built with Windows Workflow Foundation (WF4).

The Framework contains

* Test Hosts which wrap WorkflowInvoker, WorkflowApplication and WorkflowServiceHost
* Assert helper classes for asserting out arguments and tracking records

These classes are test framework independent so you can use them with MSTest, NUnit, xUnit or whatever test framework you like. In the walkthroughs we will use MSTest

## Using Microsoft.Activities.UnitTesting.dll in your project

1. Download the binaries for the platform you are targeting. You can target .NET Framework 4 or .NET Framework 4 Platform Update 1.
2. Place the binaries into a folder that can be referenced by your project
3. Add a reference and browse to the location with the assembly
4. Add the necessary using or import statements to include types from the Microsoft.Activities.UnitTesting namespace. Extension methods will not appear as an option in intellisense until you do this.

## How to tell which version of binaries you have

If you aren't sure which platform the assembly is targeting, just right click on the file in Explorer and select Properties. Then click on the details tab. Assemblies built for Platform Update 1 will contain the text (Platform Update 1) in the file description.

## C:\Users\rojacobs\AppData\Local\Temp\SNAGHTML1b9c1d1.PNG

# Walkthrough – Hello WF4 Unit Testing

* 1. In this walkthrough you will learn the basics of unit testing a simple workflow with **Microsoft.Activities.UnitTesting**. Our task is to build a workflow which meets the following requirements

|  |  |  |
| --- | --- | --- |
| Given | When | Then |
| Two positive integers | The workflow is invoked with the integers passed as arguments Num1 and Num2 | The sum is returned in an out argument named Sum |
| One positive and one negative integer | The workflow is invoked with the integers passed as arguments Num1 and Num2 | An **ArgumentOutOfRangeException** is thrown |

Task 1 – Create the Application and Test projects

* 1. Start Visual Studio and create a new Workflow Console Application named **HelloWFTest**
  2. Right click on the solution and select **Add New Project**
  3. Add a new Test Project named **HelloWFTest.Tests**
  4. Add references to **HelloWFTest** and **System.Activities**
  5. Rename **UnitTest1** to **Workflow1Tests**
  6. Right click on **HelloWFTest.Tests** and select **Add / New Folder**
  7. Name the new folder **Reference Assemblies**
  8. Download and extract **Microsoft.Activities.UnitTesting** to the **Reference Assemblies** folder
  9. Add references to **Microsoft.Activities.dll** and **Microsoft.Activities.UnitTesting.dll**

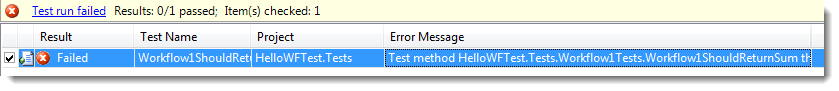
Task 2 – Write the First Test

When you write a test the question you want to ask yourself is "How would I know if this workflow is working correctly?" In our case we can simply verify that an argument named sum is returned with expected values.

* 1. In the **HelloWFTests.Tests** project open **Workflow1Tests.cs**
  2. Rename **TestMethod1** to **Workflow1ShouldReturnSum**
  3. Because our workflow is simple and will not use any bookmarks we will use **WorkflowInvokerTest** to test our activity. Add the following code to your test method.
     1. C#
     2. /// <summary>
     3. /// Given
     4. /// \* Two positive integers
     5. /// When
     6. /// \* The workflow is invoked with the integers passed as Num1 and Num2
     7. /// Then
     8. /// \* The sum is returned in an out argument named Sum
     9. /// </summary>
     10. [TestMethod]
     11. public void Workflow1ShouldReturnSum()
     12. {
     13. const int Expected = 4;
     14. // Create the activity
     15. var activity = new Workflow1();
     16. // Create the test host
     17. var host = new WorkflowInvokerTest(activity);
     18. try
     19. {
     20. // Create our input arguments using a helper class
     21. var input = InputDictionary.Create("Num1", 3, "Num2", 1);
     22. // Will invoke the activity and capture the results
     23. host.TestActivity(input);
     24. // Verify the sum argument
     25. host.AssertOutArgument.AreEqual("Sum", Expected);
     26. }
     27. finally
     28. {
     29. // Dump the tracking information to the test log
     30. host.Tracking.Trace();
     31. }
     32. }
     33. **Why use try/finally in the test?**
     34. All of the test host classes capture tracking information from the workflow as it executes. If your test fails an exception will be thrown. In the finally block we will dump the tracking information to the test log. This will help you to diagnose why the test is failing.

Task 3 – Run the test and see it fail

* + 1. It is important to see your test fail. You may accidentally write a test that always passes.
  1. Run the test. At this point you don't need to debug it. You can quickly run the unit test without debugging by pressing **Ctrl+R, T** or select **Test / Run / Tests in Current Context** from the menu.



* + 1. The test will fail because we have not implemented the workflow

Task 4 – Add the Input Arguments

* + 1. As you make the test pass, you should do only what the test requires you to do. When we examine the results of the first test run we see an exception was thrown because of the input arguments. In this task we will fix this problem. Of course we know that there will be other problems that we have to fix later. However by fixing just this problem we ensure that our workflow only does what the test demands of it and nothing more.
    2. If we find that the workflow doesn't do something it needs to do then we need to make the tests demand more. We keep repeating this process of running the test and fixing issues until the test passes. If we follow this process carefully at the end we will have a workflow that does only what the test asks of it and a test that demonstrates that the workflow does exactly what it is supposed to do and nothing more.
  1. Open **Workflow1.xaml**
  2. Open the Arguments window and add two arguments

|  |  |  |
| --- | --- | --- |
| Name | Direction | Argument type |
| Num1 | In | Int32 |
| Num2 | In | Int32 |

* 1. Run the unit test again, it will now fail because there is no out argument named Sum.
  2. Double click on the failed test. Notice how the Debug Trace area provides a dump of the tracking information
     1. Tracking
     2. \*\*\* Tracking data follows \*\*\*
     3. WorkflowInstance <Workflow1> is <Started> at 10:38:55.8250
     4. Activity <null> is scheduled child activity <Workflow1> at 10:38:55.8250
     5. Activity <Workflow1> state is Executing at 10:38:55.8250
     6. {
     7. Arguments
     8. Num1: 3
     9. Num2: 1
     10. }
     11. Activity <Workflow1> state is Closed at 10:38:55.8250
     12. {
     13. Arguments
     14. Num1: 3
     15. Num2: 1
     16. }
     17. WorkflowInstance <Workflow1> is <Completed> at 10:38:55.8406

Task 5 – Add an Out Argument named Sum

* 1. Add another argument

|  |  |  |
| --- | --- | --- |
| Name | Direction | Argument type |
| Sum | Out | Int32 |

* 1. Run the unit test again. This time the test fails because the Sum argument has a value of zero.

Task 6 – Assign a value to Sum

* + 1. At this point you are probably thinking it is about time that we implemented the workflow. After all we know that we want to assign Sum = Num1+Num2 don't we? Actually we think we want that but it is a good exercise to avoid doing anything more than what the tests actually require. After all the test thus far requires only that we return Sum = 4 so let's implement just that.
  1. Open Workflow1.xaml
  2. Drop an Assign activity and set the properties

|  |  |
| --- | --- |
| Property | Value |
| To | Sum |
| Value | 4 |

* 1. Run the unit test again. It passes

Task 7 – Refactor

* + 1. "Red, Green, Refactor" is the plan. So far we have seen the test fail (Red) then pass (Green) and now it is time to refactor. At this stage we look at our code and ask ourselves could we make it better? If we want to change it we can do so more confidently because we have a test to verify the behavior.

For our purposes we know that our test is passing but we also know we are not done. The scenario indicates that for any two positive integers we should be able to get the sum. Right now our workflow only functions for any two integers where the sum is 4. This means we need to write another test which simply passes different numbers to our workflow. Because everything else in the test is going to be the same this is a good opportunity to refactor.

* 1. Extract the body of the test method into a new method.
     1. C#
     2. private static void Workflow1ShouldReturnSumOfValues(int num1, int num2)
     3. {
     4. var expected = num1 + num2;
     5. // Create the activity
     6. var activity = new Workflow1();
     7. // Create the test host
     8. var host = new WorkflowInvokerTest(activity);
     9. try
     10. {
     11. // Create our input arguments using a helper class
     12. var input = InputDictionary.Create("Num1", num1, "Num2", num2);
     13. // Will invoke the activity and capture the results
     14. host.TestActivity(input);
     15. // Verify the sum argument
     16. host.AssertOutArgument.AreEqual("Sum", expected);
     17. }
     18. finally
     19. {
     20. // Dump the tracking information to the test log
     21. host.Tracking.Trace();
     22. }
     23. }
  2. Modify your existing test and description
     1. C#
     2. /// <summary>
     3. /// Given
     4. /// \* Two positive integers 3 and 1
     5. /// When
     6. /// \* The workflow is invoked with the integers passed as Num1 and Num2
     7. /// Then
     8. /// \* The sum is returned in an out argument named Sum
     9. /// </summary>
     10. [TestMethod]
     11. public void Workflow1ShouldReturnSumOf3And1()
     12. {
     13. Workflow1ShouldReturnSumOfValues(3, 1);
     14. }
  3. Create a new test that verifies different values
     1. C#
     2. /// <summary>
     3. /// Given
     4. /// \* Two positive integers 0 and int.MaxValue
     5. /// When
     6. /// \* The workflow is invoked with the integers passed as Num1 and Num2
     7. /// Then
     8. /// \* The sum is returned in an out argument named Sum
     9. /// </summary>
     10. [TestMethod]
     11. public void Workflow1ShouldReturnSumOf0AndMaxInt()
     12. {
     13. Workflow1ShouldReturnSumOfValues(0, int.MaxValue);
     14. }
  4. Run both tests, now one will pass and one will fail.
     1. **Does this seem like a lot of work?**
     2. Right now it might be tempting to think about skipping many of these steps because you are already quite familiar with this project. But a year from now when you or another team member look at this code you will be very glad you took the time to build these tests.

Task 8 – Make the test pass

* 1. Open Workflow1.xaml
  2. Modify the assign activity properties

|  |  |
| --- | --- |
| Property | Value |
| To | Sum |
| Value | Num1 + Num2 |

* 1. Run the tests again, they will pass

Task 9 – Test the second scenario

* + 1. So far we have good test coverage of the first scenario but we do not have test coverage for the second. In the second scenario we need to test what happens when our workflow has an unhandled exception. We want to verify that it throws the correct exception type.
    2. Different unit test frameworks provide varying methods for testing exceptions. **Microsoft.Activities.UnitTesting** includes a class called **AssertHelper** to support exception testing which we will use in this example.

|  |  |  |
| --- | --- | --- |
| Given | When | Then |
| One positive and one negative integer | The workflow is invoked with the integers passed as arguments Num1 and Num2 | An **ArgumentOutOfRangeException** is thrown |

* 1. Create another test method to verify this scenario
     1. C#
     2. /// <summary>
     3. /// Given
     4. /// \* One positive integers 1 and one negative integer -1
     5. /// When
     6. /// \* The workflow is invoked with the integers passed as Num1 and Num2
     7. /// Then
     8. /// \* An ArgumentOutOfRangeException is thrown
     9. /// </summary>
     10. [TestMethod]
     11. public void Workflow1ShouldThrowArgOutOfRangeWhenNegativeArgPassed()
     12. {
     13. AssertHelper.Throws<ArgumentOutOfRangeException>(  
          () => Workflow1ShouldReturnSumOfValues(1, -1));
     14. }
  2. Run the test. It will fail because our workflow does not throw an exception yet.

Task 10 – Make the test pass

* 1. Open **Workflow1.xaml**
  2. Right click on the **Assign** activity and select **Cut**
  3. Drop an **If** activity
  4. Inside the **Then** area right click and select **Paste**
  5. Set the condition of the expression (Note: this is incorrect but we want to simulate a bug)
     1. Visual Basic
     2. Num1 < 0 Or Num2 < 0
  6. Drop a **Throw** activity on the **Else** side
  7. Set the Exception property
     1. Visual Basic

New ArgumentOutOfRangeException()

* 1. Run the test, it will fail

Task 11 – Diagnose a Test Failure with Tracking

Often when a workflow is failing it can be difficult to know why. In this case the bug is in a Visual Basic expression in the If condition. Microsoft.Activities.UnitTesting makes it easier to see what is going on inside of your activity as it executed. Our test is failing because the exception was not thrown. In our simple activity we know what the problem is but in more complex activities it might be quite difficult to find out what happened.

* 1. Double click the Test Results and look at the tracking information.
  2. Find where the <If> activity is executing. Notice the arguments. The condition evaluated to True when it should have been false. Now we know where the error is.
  3. Correct the Condition expression
     1. Visual Basic
     2. Num1 >= 0 AndAlso Num2 >= 0
  4. Run the unit tests again, now all are passing
     1. **Why run all the tests again?**
     2. As we changed the workflow we may have broken a scenario that was passing earlier. Unit tests will help you to detect regressions early on and fix them quickly.