Unit tests for Generic Methods

Visual Studio 2015

You can generate unit tests for generic methods exactly as you do for other methods, as described in How to: Create and Run a Unit Test. The following sections provide information about and examples of creating unit tests for generic methods.

Type Arguments and Type Constraints

When Visual Studio generates a unit test for a generic class, such as `MyList<T>`, it generates two methods: a generic helper and a test method. If `MyList<T>` has one or more type constraints, the type argument must satisfy all the type constraints. To make sure that the generic code under test works as expected for all permissible inputs, the test method calls the generic helper method with all the constraints that you want to test.

Examples

The following examples illustrate unit tests for generics:

- Editing Generated Test Code. This example has two sections, Generated Test Code and Edited Test Code. It shows how to edit the raw test code that is generated from a generic method into a useful test method.
- Using a Type Constraint. This example shows a unit test for a generic method that uses a type constraint. In this example, the type constraint is not satisfied.

Example 1: Editing Generated Test Code

The test code in this section tests a code-under-test method named `SizeOfLinkedList()`. This method returns an integer that specifies the number of nodes in the linked list.

The first code sample, in the section Generated Test Code, shows the unedited test code as it was generated by Visual Studio Enterprise. The second sample, in the section Edited Test Code, shows how you could make it test the functioning of the SizeOfLinkedList method for two different data types, `int` and `char`.

This code illustrates two methods:

- a test helper method, `SizeOfLinkedListTestHelper<T>()`. By default, a test helper method has "TestHelper" in its name.
- a test method, `SizeOfLinkedListTest()`. Every test method is marked with the TestMethod attribute.

Generated Test Code

The following test code was generated from the `SizeOfLinkedList()` method. Because this is the unedited

generated test, it must be modified to correctly test the SizeOfLinkedList method.

```
public void SizeOfLinkedListTestHelper<T>()
{
    T val = default(T); // TODO: Initialize to an appropriate value
    MyLinkedList<T> target = new MyLinkedList<T>(val); // TODO: Initialize to an
appropriate value
    int expected = 0; // TODO: Initialize to an appropriate value
    int actual;
    actual = target.SizeOfLinkedList();
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("Verify the correctness of this test method.");
}

[TestMethod()]
public void SizeOfLinkedListTest()
{
   SizeOfLinkedListTestHelper<GenericParameterHelper>();
}
```

In the preceding code, the generic type parameter is `GenericParameterHelper`. Whereas you can edit it to supply specific data types, as shown in the following example, you could run the test without editing this statement.

Edited Test Code

In the following code, the test method and the test helper method have been edited to make them successfully test the code-under-test method `SizeOfLinkedList()`.

Test Helper Method

The test helper method performs the following steps, which correspond to lines in the code labeled step 1 through step 5.

1. Create a generic linked list.
2. Append four nodes to the linked list. The data type of the contents of these nodes is unknown.
3. Assign the expected size of the linked list to the variable `expected`.
4. Compute the actual size of the linked list and assign it to the variable `actual`.
5. Compare `actual` with `expected` in an Assert statement. If the actual is not equal to the expected, the test fails.

Test Method

The test method is compiled into the code that is called when you run the test named SizeOfLinkedListTest. It performs the following steps, which correspond to lines in the code labeled step 6 and step 7.

1. Specify `<int>` when you call the test helper method, to verify that the test works for `integer` variables.

2. Specify `<char>` when you call the test helper method, to verify that the test works for `char` variables.

```
public void SizeOfLinkedListTestHelper<T>()
{
    T val = default(T);
    MyLinkedList<T> target = new MyLinkedList<T>(val); // step 1
    for (int i = 0; i < 4; i++) // step 2
    {
        MyLinkedList<T> newNode = new MyLinkedList<T>(val);
        target.Append(newNode);
    }
    int expected = 5; // step 3
    int actual;
    actual = target.SizeOfLinkedList(); // step 4
    Assert.AreEqual(expected, actual); // step 5
}

[TestMethod()]
public void SizeOfLinkedListTest()
{
    SizeOfLinkedListTestHelper<int>();  // step 6
    SizeOfLinkedListTestHelper<char>(); // step 7
}
```

Note

Each time the SizeOfLinkedListTest test runs, its TestHelper method is called two times. The assert statement must evaluate to true every time for the test to pass. If the test fails, it might not be clear whether the call that specified `<int>` or the call that specified `<char>` caused it to fail. To find the answer, you could examine the call stack, or you could set breakpoints in your test method and then debug while running the test. For more information, see How to: Debug while Running a Test in an ASP.NET Solution.

Example 2: Using a Type Constraint

This example shows a unit test for a generic method that uses a type constraint that is not satisfied. The first section

shows code from the code-under-test project. The type constraint is highlighted.

The second section shows code from the test project.

Code-Under-Test Project

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;

namespace ClassLibrary2
{
    public class Employee
    {
        public Employee(string s, int i)
        {
        }
    }

    public class GenericList<T> where T : Employee
    {
        private class Node
        {
            private T data;
            public T Data
            {
                get { return data; }
                set { data = value; }
            }
        }
    }
}
```

Test Project

As with all newly generated unit tests, you must add non-inconclusive Assert statements to this unit test to make it return useful results. You do not add them to the method marked with the TestMethod attribute but to the "TestHelper" method, which for this test is named `DataTestHelper<T>()`.

In this example, the generic type parameter `T` has the constraint `where T : Employee`. This constraint is not satisfied in the test method. Therefore, the `DataTest()` method contains an Assert statement that alerts you to the requirement to supply the type constraint that has been placed on `T`. The message of this Assert statement reads as follows: `("No appropriate type parameter is found to satisfies the type constraint(s) of T. " + "Please call DataTestHelper<T>() with appropriate type parameters.");`

In other words, when you call the `DataTestHelper<T>()` method from the test method, `DataTest()`, you must pass a parameter of type `Employee` or a class derived from `Employee`.

```
using ClassLibrary2;

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TestProject1
```

```
{
    [TestClass()]
    public class GenericList_NodeTest
    {

        public void DataTestHelper<T>()
            where T : Employee
        {
            GenericList_Shadow<T>.Node target = new GenericList_Shadow<T>.Node();
// TODO: Initialize to an appropriate value
            T expected = default(T); // TODO: Initialize to an appropriate value
            T actual;
            target.Data = expected;
            actual = target.Data;
            Assert.AreEqual(expected, actual);
            Assert.Inconclusive("Verify the correctness of this test method.");
        }

        [TestMethod()]
        public void DataTest()
        {
            Assert.Inconclusive("No appropriate type parameter is found to
satisfies the type constraint(s) of T. " +
            "Please call DataTestHelper<T>() with appropriate type parameters.");
        }
    }
}
```

See Also

Anatomy of a Unit Test
Unit Test Your Code

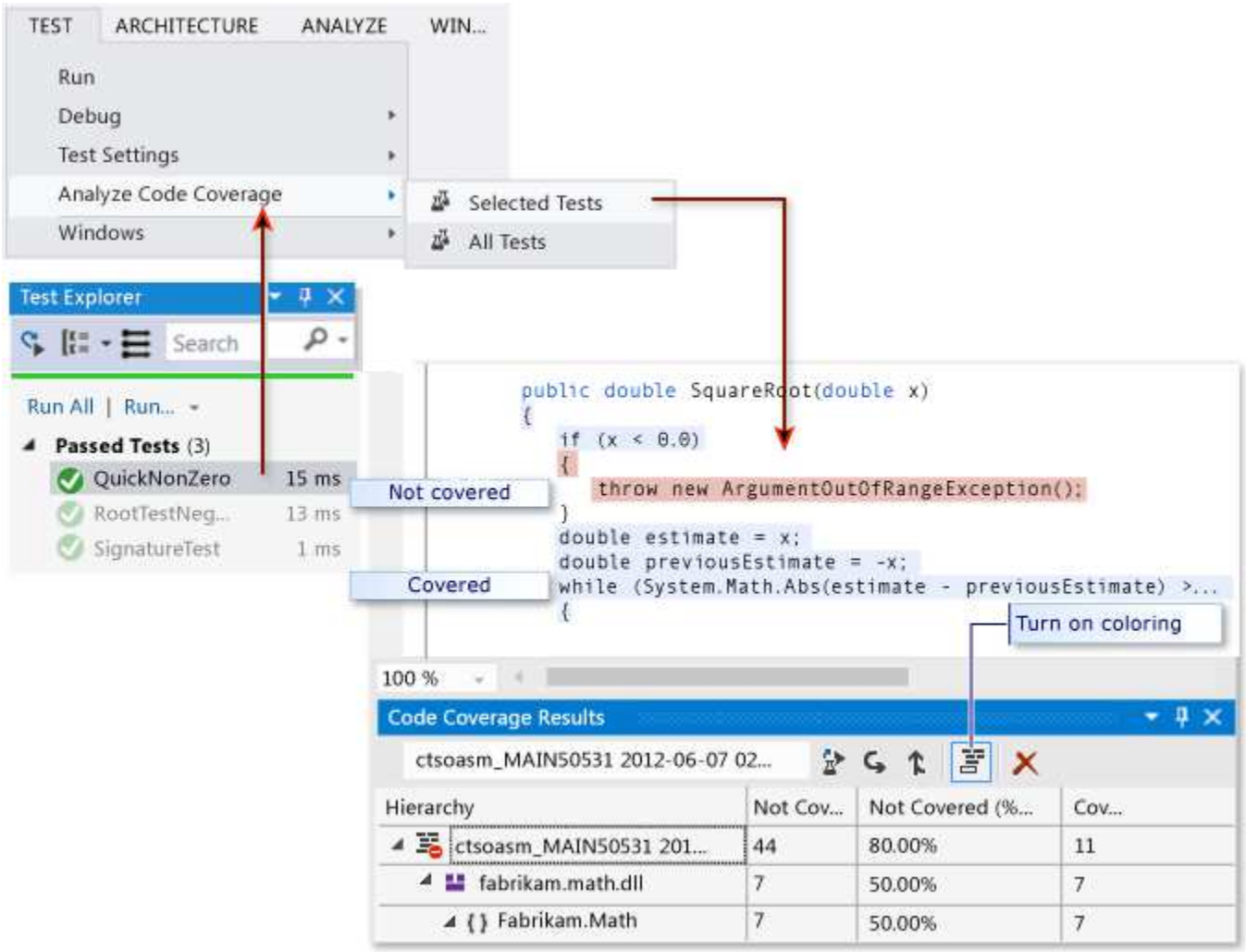
Using Code Coverage to Determine How Much Code is being Tested

Visual Studio 2015

To determine what proportion of your project's code is actually being tested by coded tests such as unit tests, you can use the code coverage feature of Visual Studio. To guard effectively against bugs, your tests should exercise or 'cover' a large proportion of your code.

Code coverage analysis can be applied to both managed (CLI) and unmanaged (native) code.

Code coverage is an option when you run test methods using Test Explorer. The results table shows the percentage of the code that was run in each assembly, class, and method. In addition, the source editor shows you which code has been tested.



Requirements

- Visual Studio Enterprise

To analyze code coverage on unit tests in Test Explorer

1. On the **Test** menu, choose **Analyze Code Coverage**.
2. To see which lines have been run, choose **Show Code Coverage Coloring**.

   To alter the colors, or to use bold face, choose **Tools**, **Options**, **Environment**, **Fonts and Colors**, **Show settings for: Text Editor**. Under **Display Items**, adjust the Coverage items.
3. If the results show low coverage, investigate which parts of the code are not being exercised, and write more tests to cover them. Development teams typically aim for about 80% code coverage. In some situations, lower coverage is acceptable. For example, lower coverage is acceptable where some code is generated from a standard template.

Tip

To get accurate results:

- Make sure that compiler optimization is turned off.

  If you are working with unmanaged (native) code, use a debug build.
- Make sure that you are generating .pdb (symbol) files for each assembly.

If you don't get the results you expect, see Troubleshooting Code Coverage. . Don't forget to run code coverage again after updating your code. Coverage results and code coloring are not automatically updated after you modify your code or when you run tests.

Reporting in blocks or lines

Code coverage is counted in *blocks*. A block is a piece of code with exactly one entry and exit point. If the program's control flow passes through a block during a test run, that block is counted as covered. The number of times the block is used has no effect on the result.

You can also have the results displayed in terms of lines by choosing **Add/Remove Columns** in the table header. If the test run exercised all the code blocks in any line of code, it is counted as one line. Where a line contains some code blocks that were exercised and some that were not, that is counted as a partial line.

Some users prefer a count of lines because the percentages correspond more closely to the size of the fragments that you see in the source code. A long block of calculation would count as a single block even if it occupies many lines.

Managing code coverage results

The Code Coverage Results window usually shows the result of the most recent run. The results will vary if you change your test data, or if you run only some of your tests each time.

The code coverage window can also be used to view previous results, or results obtained on other computers.

You can merge the results of several runs, for example from runs that use different test data.

- **To view a previous set of results**, select it from the drop-down menu. The menu shows a temporary list that is cleared when you open a new solution.
- **To view results from a previous session**, choose **Import Code Coverage Results**, navigate to the TestResults folder in your solution, and import a .coverage file.

  The coverage coloring might be incorrect if the source code has changed since the .coverage file was generated.
- **To make results readable as text**, choose **Export Code Coverage Results**. This generates a readable .coveragexml file which you could process with other tools or send easily in mail.
- **To send results to someone else**, send either a .coverage file or an exported .coveragexml file. They can then import the file. If they have the same version of the source code, they can see coverage coloring.

Merging results from different runs

In some situations, different blocks in your code will be used depending on the test data. Therefore, you might want to combine the results from different test runs.

For example, suppose that when you run a test with input "2", you find that 50% of a particular function is covered. When you run the test a second time with the input "-2" you see in the coverage coloring view that the other 50% of the function is covered. Now you merge the results from the two test runs, and the report and coverage coloring view show that 100% of the function was covered.

Use **Merge Code Coverage Results** to do this. You can choose any combination of recent runs or imported results. If you want to combine exported results, you must import them first.

Use **Export Code Coverage Results** to save the results of a merge operation.

Limitations in merging

- If you merge coverage data from different versions of the code, the results are shown separately, but they are not combined. To get fully combined results, use the same build of the code, changing only the test data.
- If you merge a results file that has been exported and then imported, you can only view the results by lines, not by blocks. Use the **Add/Remove Columns** command to show the line data.
- If you merge results from tests of an ASP.NET project, the results for the separate tests are displayed, but not combined. This applies only to the ASP.NET artifacts themselves: results for any other assemblies will be combined.

Excluding elements from the code coverage results

You might want to exclude specific elements in your code from the coverage scores, for example if the code is generated from a text template. Add the attribute `System.Diagnostics.CodeAnalysis.ExcludeFromCodeCoverage` to any of the following code elements: class, struct, method, property, property setter or getter, event. Note that excluding a class

does not exclude its derived classes.

For example:

VB

```
Imports System.Diagnostics.CodeAnalysis


Class ExampleClass1
    <ExcludeFromCodeCoverage()>
    Public Sub ExampleSub1()
        ...
    End Sub

    ' Exclude property
    < ExcludeFromCodeCoverage()>
    Property ExampleProperty1 As Integer
        ...
    End Property

    ' Exclude setter
    Property ExampleProperty2 As Integer
        Get
            ...
        End Get
        <ExcludeFromCodeCoverage()>
        Set(ByVal value As Integer)
            ...
        End Set
    End Property
End Class

<ExcludeFromCodeCoverage()>
Class ExampleClass2
...
End Class
```

Excluding elements in Native C++ code

To exclude unmanaged (native) elements in C++ code:

C++

```
#include <CodeCoverage\CodeCoverage.h>
...

// Exclusions must be compiled as unmanaged (native):
#pragma managed(push, off)

// Exclude a particular function:
```

```
ExcludeFromCodeCoverage(Exclusion1, L"MyNamespace::MyClass::MyFunction");

// Exclude all the functions in a particular class:
ExcludeFromCodeCoverage(Exclusion2, L"MyNamespace::MyClass2::*");

// Exclude all the functions generated from a particular template:
ExcludeFromCodeCoverage(Exclusion3, L"*::MyFunction<*>");


// Exclude all the code from a particular .cpp file:
ExcludeSourceFromCodeCoverage(Exclusion4, L"*\\unittest1.cpp");

// After setting exclusions, restore the previous managed/unmanaged state:
#pragma managed(pop)
```

Use the following macros:

```
ExcludeFromCodeCoverage(ExclusionName, L"FunctionName");

ExcludeSourceFromCodeCoverage(ExclusionName, L"SourceFilePath");
```

- *ExclusionName* is any unique name.
- *FunctionName* is a fully qualified function name. It may contain wildcards. For example, to exclude all the functions of a class, write `MyNamespace::MyClass::*`
- *SourceFilePath* is the local or UNC path of a .cpp file. It may contain wildcards. The following example excludes all files in a particular directory: `\\MyComputer\Source\UnitTests\*.cpp`
- `#include <CodeCoverage\CodeCoverage.h>`
- Place calls to the exclusion macros in the global namespace, not within any namespace or class.
- You can place the exclusions either in the unit test code file or the application code file.
- The exclusions must be compiled as unmanaged (native) code, either by setting the compiler option or by using `#pragma managed(off)`.

Note

To exclude functions in C++/CLI code, apply the attribute `[System::Diagnostics::CodeAnalysis::ExcludeFromCodeCoverage]` to the function. This is the same as for C#.

Including or excluding additional elements

Code coverage analysis is performed only on assemblies that are loaded and for which a .pdb file is available in the

same directory as the .dll or .exe file. Therefore in some circumstances, you can extend the set of assemblies that is included by getting copies of the appropriate .pdb files.

You can exercise more control over which assemblies and elements are selected for code coverage analysis by writing a .runsettings file. For example, you can exclude assemblies of particular kinds without having to add attributes to their classes. For more information, see Customizing Code Coverage Analysis.

Analyzing code coverage in the build service

When you check in your code, your tests will run on the build server, along with all the other tests from other team members. (If you haven't already set this up, see Run tests in your build process.) It's useful to analyze code coverage on the build service, because that gives the most up-to-date and comprehensive picture of coverage in the whole project. It will also include automated system tests and other coded tests that you don't usually run on the development machines.

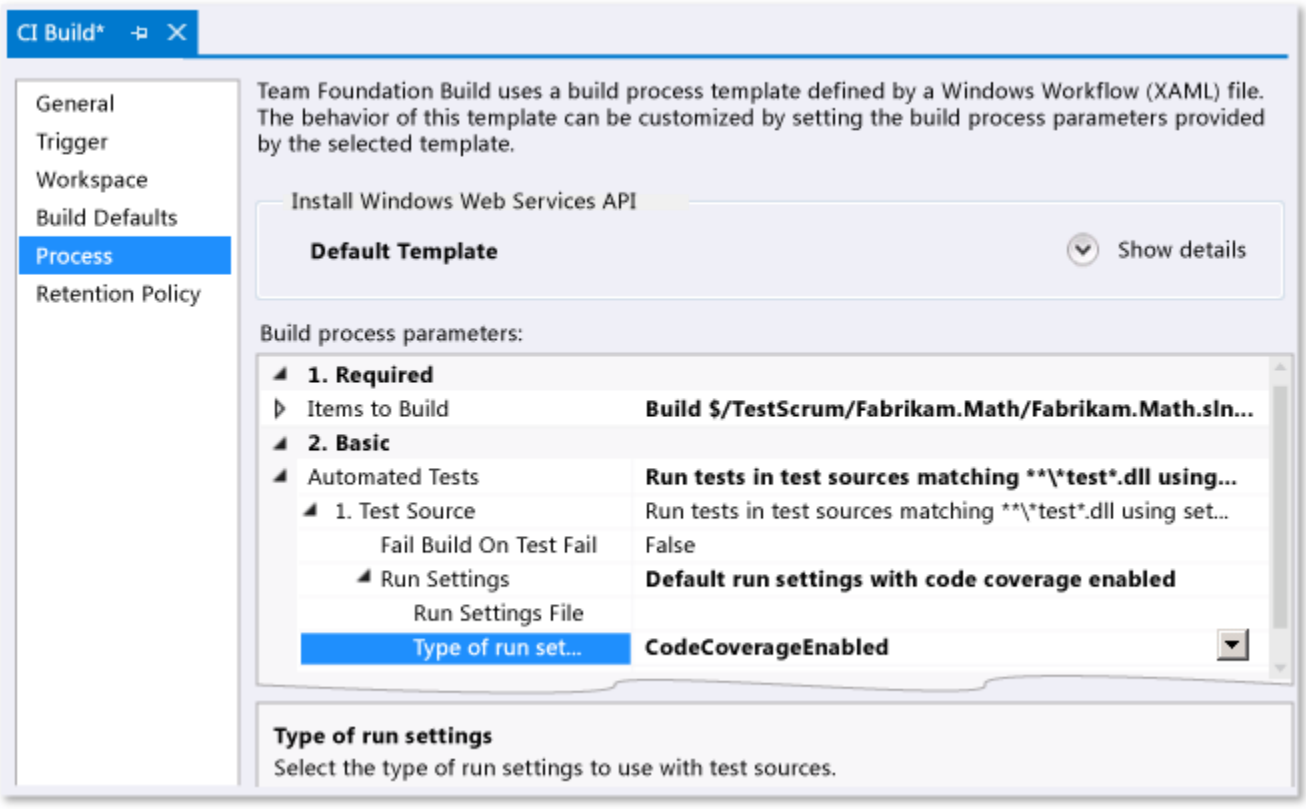
1. In Team Explorer, open **Builds**, and then add or edit a build definition.

2. On the **Process** page, expand **Automated Tests**, **Test Source**, **Run Settings**. Set **Type of Run Settings File** to **Code Coverage Enabled**.

   If you have more than one Test Source definition, repeat this step for each one.

   - *But there is no field named **Type of Run Settings File**.*

     Under **Automated Tests**, select **Test Assembly** and choose the ellipsis button **[...]** at the end of the line. In the **Add/Edit Test Run** dialog box, under **Test Runner**, choose **Visual Studio Test Runner**.



After the build runs, the code coverage results are attached to the test run and appear in the build summary.

Analyzing Code Coverage in a Command Line

To run tests from the command line, use vstest.console.exe. Code coverage is an option of this utility. For more information, see VSTest.Console.exe command-line options.

1. Launch the Visual Studio Developer Command Prompt:

   On the Windows **Start** menu, choose **All Programs**, **Microsoft Visual Studio**, **Visual Studio Tools**, **Developer Command Prompt**.

2. Run:

```
vstest.console.exe MyTestAssembly.dll /EnableCodeCoverage
```

Troubleshooting

If you do not see code coverage results, see Troubleshooting Code Coverage.

External resources

Guidance

Testing for Continuous Delivery with Visual Studio 2012 – Chapter 2: Unit Testing: Testing the Inside

See Also

Customizing Code Coverage Analysis
Troubleshooting Code Coverage
Unit Test Your Code

Walkthrough: Run tests and view code coverage

[This documentation is for preview only, and is subject to change in later releases. Blank topics are included as placeholders.]

To see what proportion of your project's code is actually being tested, use the code coverage feature of Visual Studio 2012 RC. To do this, first edit the run configuration to indicate the assembly containing the code whose coverage you want to measure; then, run tests on that code. Detailed code coverage statistics appear in a window, and you can also see, line-by-line, which code has been tested.

Prerequisites

- Visual Studio Premium 2010 or Visual Studio Ultimate 2010
- Perform the steps in the procedure "Run a Unit Test and Fix Your Code" in Walkthrough: Creating and running unit tests for managed code. This creates the two tests that you will run in the following procedure.

Run Tests and View Code Coverage

To run tests and view code coverage

1. In Solution Explorer, note the name of your solution. If you used the project from Walkthrough: Creating and running unit tests for managed code, the solution's name is Bank. This solution contains the code-under-test.
2. In Solution Explorer, under Solution Items, double-click the test settings file, Local.testsettings.

   The **Test Settings** dialog box is displayed.
3. Select **Data and Diagnostics**.
4. Under **Role**, select **<Local machine only>** as the role to use to collect code coverage data.

Caution
For code coverage data this must be the role that will run the tests.
5. To modify the default code coverage settings, in the list of data diagnostic adapters select the check box for **Code Coverage** and then click **Configure** located immediately above the list of data diagnostic adapters.

The **Code Coverage Detail** dialog box to configure code coverage collection is displayed.

> **Caution**
>
> Collecting code coverage data does not work if you also have the test setting configured to collect IntelliTrace information.

6. From the list, select the artifacts that you want to instrument.
7. (Optional) To add another assembly that is not displayed, click **Add Assembly**.

   The **Choose Assemblies to Instrument** dialog box is displayed.

   a. Locate the assembly file (.exe, .dll, or .ocx) that you want to include in code coverage and then click **Open**. The file is added to the list.
8. (Optional) Select **Instrument assemblies in place** to instrument the files in the location where they are built or after you copy them to a deployment directory. For more information about where to instrument your assemblies, see Choosing the Instrumentation Folder.
9. (Optional) If any one of your assemblies that you added have a strong name, you might have to re-sign these assemblies. Click (**...**) to locate the key file that must be used when they are re-signed. For more information about how assemblies are re-signed, see Instrumenting and Re-Signing Assemblies.
10. Click **OK**. The code coverage settings are now configured and saved for your test settings.

> **Note**
>
> To reset the configuration for this diagnostic data adapter, click **Reset to default configuration**.

11. Click **Save As** and then click **Save** in the dialog box. A message box appears, prompting you to save the existing file. Click **Yes** in the message box to replace the existing file.
12. On the **Test** menu, point to **Select Active Test Settings**. A submenu displays all the test settings in the solution. Put a check mark next to the test settings that you just edited, Local.testsettings. This makes it the active test settings.
13. In the **Test List Editor**, select the check boxes next to **CreditTest** and **DebitTest**, right-click, and then click **Run Checked Tests**.

    The two tests run.
14. On the **Test Tools** toolbar, click **Code Coverage Results**.

    The **Code Coverage Results** window opens.
15. In the **Code Coverage Results** window, the **Hierarchy** column displays one node that contains data for all the code coverage achieved in the latest test run. The test run node is named using the format <user name>@<computer name> <date> <time>. Expand this node.

16. Expand the node for the assembly, Bank.dll, for the namespace, BankAccountNS, and for the BankAccount class.

17. The rows within the BankAccount class represent its methods. The columns in this table display coverage statistics for individual methods, for classes, and for the entire namespace.

18. Double-click the row for the **Debit** method.

    The Class1.cs source-code file opens to the Debit method. In this file, you can see code highlighting. Lines highlighted light blue were exercised in the test run, lines highlighted beige were partially exercised and lines highlighted reddish brown were not exercised at all. By scrolling, you can see the coverage for the other methods in this file.

    If you selected the check box for TestProject1.dll in step 7, you can open Class1Test.cs, the source-code file that contains your unit tests, to see which test methods were exercised. The same highlighting scheme applies: light blue indicates exercised code; beige indicates a partially exercised code path, and reddish brown indicates a code path that was untraveled in the test run.

See Also

Tasks

Walkthrough: Creating and running unit tests for managed code
Sample project for creating unit tests
Create Test Settings to Run Automated Tests from Visual Studio
How to: Apply Test Settings from Microsoft Visual Studio

Walkthrough: using the command-line test utility

This walkthrough shows you how to run unit tests from a command-line prompt and then view the results.

Prerequisites

- In the walkthrough entitled, Walkthrough: Creating and Running Unit Tests for Managed Code perform the following procedures: "Prepare the Walkthrough", "Create a Unit Test," and "Run a Unit Test and Fix Your Code".
- The Woodgrove Bank project. See Sample Project for Creating Unit Tests.

Use the Command-line Test Utility

To use the command-line test utility

1. Open a Visual Studio command prompt.

   To do this, choose **Start**, point to **All Programs**, point to **Microsoft Visual Studio 2012**, point to **Visual Studio Tools**, and then choose **Developer Command Prompt**.

   The command prompt opens to the folder: <drive>:\Program Files\Microsoft Visual Studio 11.0\VC

2. Change directory to the folder that contains the assembly built from your test project.

   To do this, first change directory to your solution folder. For the Bank solution that was created in the prerequisite walkthrough, this folder is: <drive>:\Documents and Settings\<username>\My Documents\Visual Studio\Projects\Bank. Then change directory to the folder for your test project by typing the following command at the command prompt:

   cd TestProject1\bin\Debug

   This folder contains the test project you created in the procedures for creating and running unit tests. The test project assembly, TestProject1.dll, contains just a few unit tests.

   > **Note**
   >
   > Your production code project and your test project will produce distinct assemblies. Make sure to run the command-line utility on the assembly of the test project, not on the assembly of your production code project.

3. MSTest.exe is a command-line utility that lets you start and control the execution of tests. You can view the choices

that MSTest.exe makes available through its options by typing the following at the command prompt:

MSTest /?

4. Use the command-line utility to test the application.

   Type the following at the command prompt:

   MSTest /testcontainer:TestProject1.dll

   This command runs all three tests and returns results such as the following:

   Loading TestProject1.dll...

   Starting Execution...

   Results Top Level Tests

   -------- ------------------

   Inconclusive TestProject1.BankAccountTest.CreditTest

   Passed TestProject1.BankAccountTest.DebitTest

   Passed TestProject1.BankAccountTest.FreezeAccountTest

   2/3 test(s) Passed, 1 Inconclusive

   Summary

   Test Run Inconclusive.

   Inconclusive 1

   Passed 2

   Total 3

   Results file: <path>\<test run name>.trx

   Test Settings: Default Test Settings

> **Note**
>
> If you complete the procedure, "Create and Run a Unit Test for a Private Method" in Walkthrough: Creating and Running Unit Tests for Managed Code, this command will also show results for the GetAccountTestType unit test.

5. Run the tests again and save test results to a specified file.

   Type the following at the command prompt:

   MSTest /testcontainer:TestProject1.dll /resultsfile:testResults1.trx

   This command runs all three tests and returns the same results as in the previous step. It also creates a file that is named testResults1.trx, and writes test results to that file, formatted for viewing in an XML viewer such as Microsoft Internet Explorer or Microsoft Visual Studio. If testResults1.trx already exists, MSTest.exe will not run and will show an error stating that a file with that name already exists.

   > **Note**
   >
   > For more information about the full range of options you can use with the MSTest command, see MSTest.exe command-line options.

6. (Optional) View the test results file. Type the following at the command prompt:

   testResults1.trx

   This opens Internet Explorer and displays the test results. Alternatively, you can open this file in the Visual Studio integrated development environment (IDE), as follows:

   a. Choose **File**, point to **Open** and then choose **File**.

   b. In the **Open File** dialog box, open the folder that contains the .xml file.

   c. Double-click **testResults1.xml**.

The command-line utility MSTest.exe is especially useful for automating test runs, to be started in batch files or other utilities.

See Also

Run automated tests from the command line using MSTest
MSTest.exe command-line options
Walkthrough: Creating and Running Unit Tests for Managed Code

JANUARY 2015 VOLUME 30 NUMBER 1

Visual Studio 2015 - Web-Based Test Case Management with TFS

By Manoj Bableshwar | January 2015

Application Lifecycle Management with Team Foundation Server (TFS) is all about leveraging an integrated toolset to manage your software projects, from planning and development through testing and deployment. As a core part of Team Foundation Server, the Test Hub enables you to create and run manual tests through an easy-to-use Web-based interface that can be accessed via all major browsers on any platform. In this article, I'll delve into the phases of manual testing—planning and creating tests, reviewing them with stakeholders, running the tests, and tracking the test progress of the team. I'll touch on different value propositions, such as the flexibility to customize workflows; end-to-end traceability; criteria-based test selection; change tracking and audit; sharing test steps and test data; stakeholder review; and most important, ease of use, especially for testers who've been using Excel-based frameworks for manual testing. To access the Test Hub, you can navigate to it by clicking on the Test tab in on-premises TFS, just the way you access the Work tab to manage backlogs or the Build tab to monitor builds. Alternatively, you can sign up for a free Visual Studio Online (VSO) account at visualstudio.com and activate the 90-day account trial to try out Test Hub.

Plan Test Activity for the Sprint

Sprints or iterations are units of planning for teams that practice Agile or Scrum methodologies. It makes sense to plan test efforts for a sprint, just as it's done for user stories. To get started with test planning, create a test plan by providing a name and associating it with a team and a sprint. The test plan can have an owner and test cycle dates for out-of-band test activity such as a beta release sign-off or a user acceptance test cycle. Test plans in TFS are work items, so you get all the benefits of work items, such as change-tracking with work item history; permissions based on area paths; rich-text summary fields; file attachments and more. However, the most important benefit of work items is customization. Work item customization makes it possible to align the workflows and fields of artifacts used for tracking activities with the business processes used by the organization. This concept can be extended to better reflect the test activities practiced as part of your software development model, by customizing test plan work items. Moreover, the process of customizing test plan work items is similar to that of other work items, such as bugs or user stories. For example, the default states of a test plan can be changed from Active and Inactive to, say, Authoring, Testing, or Archived. Additional user fields such as reviewers, approvers, signoff owner, and so forth, needed for accountability or audit requirements, can be added to the test plan. As you integrate your processes into the test plan, you may want to restrict access to it, so that only certain people, such as team leads or test managers, have access to create and modify test plans. The Manage Test

Plans permission can be used to moderate access to test plans at a user or team level.

Once you've set up a test plan, you'll be eager to create and run tests. But before that, it's important to think about the best way to organize those tests to enable reuse and end-to-end traceability of test efforts. Test suites are artifacts that are contained in a test plan and enable grouping of test cases into logical units. Test suites are of three types: requirement-based test suites (RBS), query-based test suites (QBS) and static test suites. Static test suites work like folders to organize RBS and QBS. If you want to group test cases yourself, you can manually pick and add test cases to static test suites.

Like test plans, test suites are work items, so all the customization benefits mentioned earlier apply to test suites. Some examples of custom fields for a test suite are summary fields describing instructions to set up the test application and fields to describe the nature of tests such as functional or integration, test complexity, and so on. Just as with test plans, you can moderate access to test suites at a user or team level with the Manage Test Suites permission. Changes to test cases contained in the suite, owner, state or other fields can be tracked in the test suite work item history.

End-to-end Traceability with Requirement-Based Suites

Requirement-based suites correspond to user stories (or product backlog items for scrum and requirements for CMMI-based projects) that the team is working on in the current sprint. The goal of creating an RBS by picking a user story is to enable traceability. Test cases created in an RBS are automatically linked to a user story, making it easy to find the scenarios covered to test the user story. Bugs, if any, that are filed while running these test cases are also linked to the user story and the test case, thus providing end-to-end visibility of a user story, its test scenarios and open bugs. This helps you measure the quality and ship-readiness of a feature.

Criteria-Based Testing with Query-Based Suites

Regression-test coverage is as important as test coverage for new features. Teams typically set up regression-test coverage based on criteria—all priority 1 tests, all end-to-end scenario tests, all automated tests and so forth. Test Hub supports criteria-based testing with QBS; these suites are created by defining a query on test cases. Test cases that match the query criteria are automatically populated in the QBS, without any need to manually refresh the QBS. QBS can also be used in other scenarios, such as tracking test cases for bugs that are being fixed in the current sprint.

Creating Test Cases with an Excel-Like Grid

Test cases are the basic units of testing, each containing test steps that describe a set of actions to be performed, and expected results that describe what has to be validated at each test step. Each test step can have an optional attachment, for example, a screenshot that illustrates the output. Like test plans and test suites, test cases are work items, so all benefits of work item customization apply to test cases, as well.

There are two ways to create test cases. The first option is to use the test case work item form, which lets you create one test case at a time. The second option, and the one that really lets you breeze through creating test cases, is the Excel-like grid shown in **Figure 1**. The grid resonates very well with manual testers, who, typically, would've written and tested their test cases in Excel.

With the grid, testers can create multiple test cases at a time, fluently typing test titles, steps, and expected results while navigating the grid with tabs, arrows, and the Enter key. It's a simple experience to insert, delete, cut, copy and paste rows. What's more, the grid can display all test case fields, such as state, tags, automation status and so on, plus these fields can be bulk-marked for multiple test cases. If you have an intermittent Internet connection or are just more comfortable writing test cases in Excel, you're welcome to do that. Just copy and paste all the test cases you've written in Excel into the grid and save them to populate them into the system. In fact, if your team is just adopting the TFS Test Hub for testing, the grid can help you import your test cases from Excel. Check out the Test Case Migrator Plus utility at tcmimport.codeplex.com for advanced import requirements from Excel.

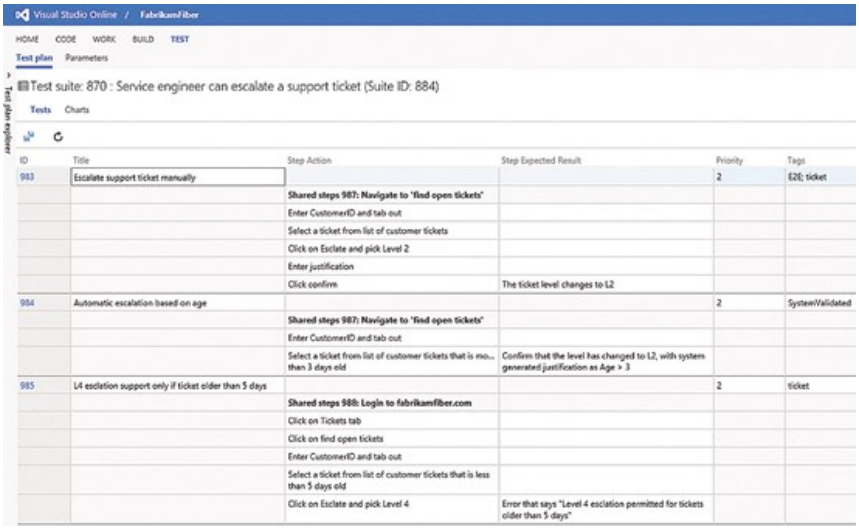


Figure 1 The Excel-Like Grid Can Be Used to Create Multiple Tests

Share Test Steps and Test Data

Some test scenarios need specific test data as input to be meaningfully tested. Also, it makes sense to repeat tests with different variants of test data, for example, valid and invalid input sets or different combinations of items in a shopping basket. Parameters can be used to associate a test case with test data. With mature test teams that cover large and complex test scenarios, it's quite possible that many test cases use similar test data to drive testing. Shared parameters can help you consolidate and centrally manage such test data. You can also import test data from Excel and use it to drive tests through shared parameters.

Just as with the test data, it's possible the test steps are common across multiple test cases, for example the steps to log into an application or navigate to a form. Such common test steps can be consolidated into shared steps. The advantage of using shared steps is that a change, such as an updated application URL or an additional authentication step while logging in, can be updated in the shared step. Changes to shared parameters or shared steps will reflect across all referenced test cases instantly.

Review Tests with Stakeholders

Before running tests, it's a good idea to share the tests with stakeholders, such as product managers or business analysts, to solicit their comments. In cross-division or cross-organization development and test teams, such as outsourced test projects, a formal signoff may be required before proceeding with test execution. To share tests with stakeholders for review, you can export a test plan or a bunch of test suites by e-mail or print them to PDF or hard

copy. **T**he output of the e-mail dialog can be edited before sending it to stakeholders. You can also copy and paste to Word documents when stakeholders are required to respond with inline review comments.

Running Tests with the Web-Based Test Runner

To prepare the team to run tests, the test lead can assign tests to team members. The owner of a test case and the tester of a test case can be different people; the test lead has the flexibility to shuffle testers or even take the help of vendors to have tests executed. The most valuable feature of the Web-based Test Runner, which is used to run manual tests, is its cross-platform support. Because the Test Runner is browser-based, you can run it on any platform that supports any major browser—Internet Explorer, Chrome, Firefox and Safari.

The Test Runner presents the test steps and expected results in a narrow window, making it easy to read and execute the steps on the application being tested (see **Figure 2**). Image attachments that were created while writing the test case are visible and can be zoomed into. If your test case is driven by test data, then each row of parameter values included in the test case will correspond to one iteration of the test.

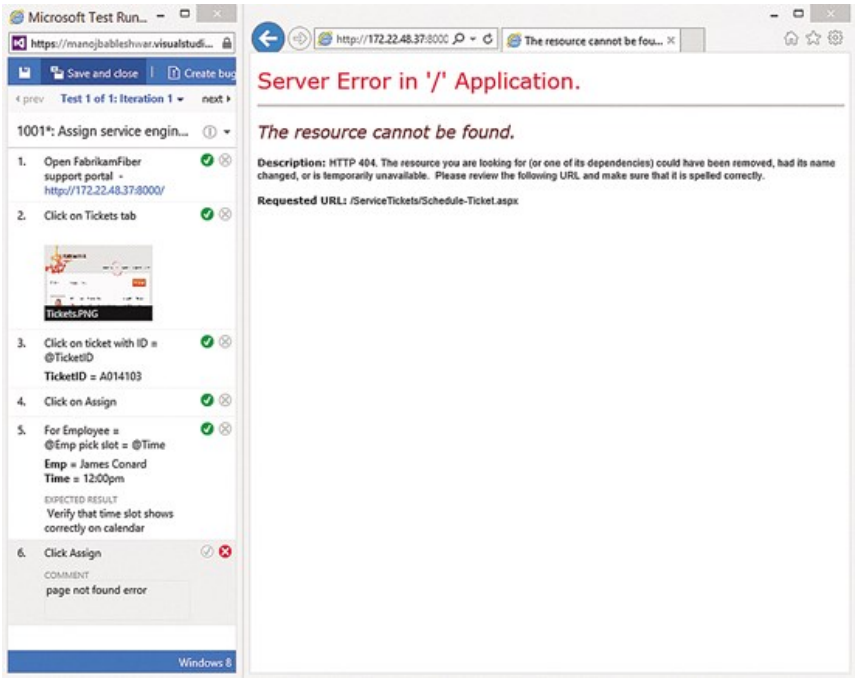


Figure 2 Web-Based Test Runner

A test can have different outcomes—Passed, Failed, Blocked and Not Applicable. The Blocked state can be used when tests are waiting on an external dependency, such as a bug fix, and Not Applicable is useful when a test doesn't apply to the current feature—a service release, for example. As you walk through validating the test steps, you mark them passed or failed. For failed steps, you can jot down comments for issues you observed while testing. You can report the failure to developers by creating a bug, right in the context of the Test Runner session. The bug is auto-populated with all the steps performed before you encountered the issue. The bug can also be updated with additional comments and screenshots before filing it. The bug is linked to the test case that was run while filing it and the requirement being tested, thus enabling end-to-end traceability. On the other hand, if you find that the discrepancy between the expected results and the application is because the application was recently updated, you can fix the test case inline, while it's running. If you're in a really long test session running many tests and

need to take a break, you can pause the tests and resume them later. If you find that a test is failing for you, and wish to find out when it last passed or which team member got to execute it successfully, looking at the recent results of the test case will answer those questions.

While the Test Runner helps you walk through each test step of a test case in detail, the bulk-mark feature helps you pass or fail multiple tests at once. If you're validating high-level test scenarios highlighted by the test case title, but not actually walking through detailed test steps, you can quickly mark each test's outcome, without launching the Test Runner. The bulk-mark feature is particularly helpful when a large number of tests have been executed offline and their status has to be reflected back in the system.

Track Test Progress with Charts

"Is my feature ship-ready?" "Is my team on track to complete testing this sprint?" "Are all the test cases I planned for this sprint ready to run?" These are some of the questions in which test leads, test managers and stakeholders are interested. The Test Hub lets you create a rich set of charts to help answer such questions (see **Figure 3**). Charts come in two sets: test case charts that can be used to track the progress of test authoring activities, and test result charts that can be used to track test execution activities. And these charts can be different kinds of visualizations—pie, column, stacked bar, pivot table and so forth. Test case fields, such as owners, state, priority and the like can be used as pivots for test case charts. Test result charts come with the test suite, outcome, tester, run by, priority and more as pivots. For example, to find the test status of user stories, you can create a stacked bar chart with test suite and outcome as pivots for all the requirements-based suites being tested in the current sprint. These charts can either be created for a bunch of test suites or for a test plan to roll up information for the entire test plan. You can also share the insights with stakeholders by pinning the charts to the homepage. Finally, all the charts display real-time metrics, without any lag or processing delay.

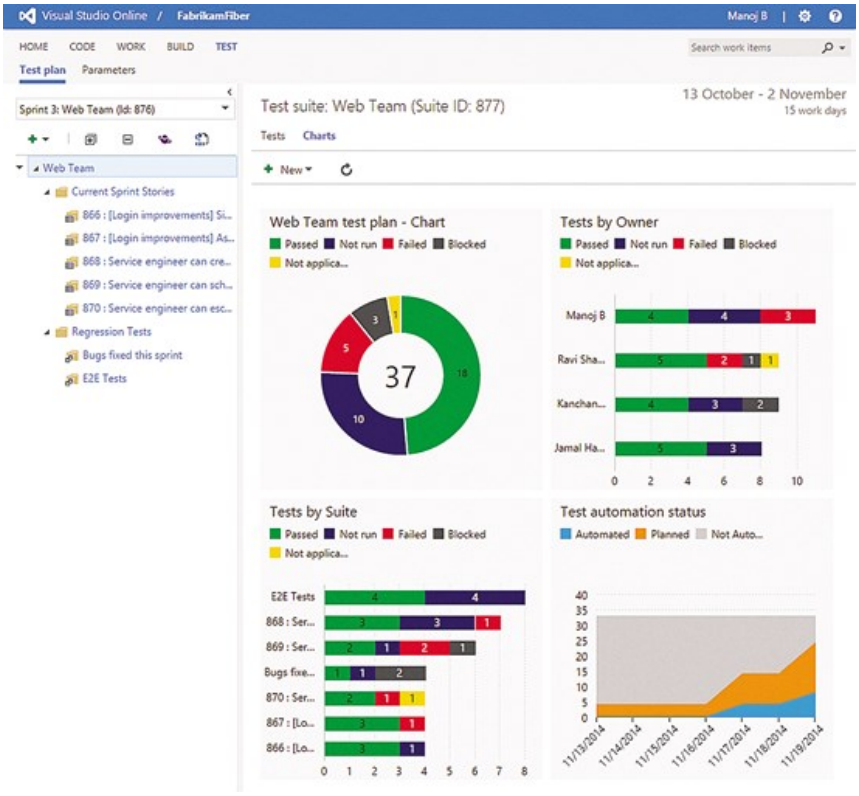


Figure 3 Tracking Test Results

Wrapping Up

The Test Hub isn't just for manual testers. It's a tool that product owners and business analysts can use to gauge how their features measure up against the acceptance criteria. The grid can be used to keep track of acceptance criteria for requirements, and can later be used for sign-off. To summarize, the Test Hub offers:

- Customization of workflows with test plan, test suite and test case work items.
- End-to-end traceability from requirements to test cases and bugs with requirement-based test suites.
- Criteria-based test selection with query-based test suites.
- Excel-like interface with the grid for easy test case creation.
- Reusable test steps and test data with shared steps and shared parameters.
- Sharable test plans, test suites and test cases for reviewing with stakeholders.
- Browser-based test execution on any platform.
- Real-time charts for tracking test activity.

Test Hub provides an easy yet comprehensive way to test the user stories you plan to release in a sprint. Test Hub is available on-premises with TFS, as well as in the cloud with Visual Studio Online. Get started with a free 90-day trial right away at visualstudio.com. To see Test Hub in action, watch the demo at aka.ms/WebTCMDemo.

Manoj Bableshwar *is a program manager with the Visual Studio Online team at Microsoft. His team ships Manual Testing Tools to Visual Studio Online.*

Thanks to the following Microsoft technical expert for reviewing this article: Ravi Shanker
Ravi Shanker Ravi works as a Principal Program Manager with the Visual Studio Testing Tools team.

MSDN Magazine Blog

14 Top Features of Visual Basic 14: The Q&A
Wednesday, Jan 7
Big Start to the New Year at MSDN Magazine
Friday, Jan 2

More MSDN Magazine Blog entries >

Current Issue

Browse All MSDN Magazines

Subscribe to MSDN Flash newsletter

Receive the MSDN Flash e-mail newsletter every other week, with news and information personalized to your interests and areas of focus.