

Microsoft[®] Virtual Labs

**Application Validation through
Unit Testing**

Microsoft[®]

Table of Contents

Application Validation through Unit Testing.....	1
Exercise 1 Generate Unit Tests.....	2
Exercise 2 Author Unit Tests.....	6
Exercise 3 Test non-public data.....	8
Exercise 4 Create Data-Driven Unit Tests.....	11
Exercise 5 Viewing Test Results	16
Exercise 6 Examining Code Coverage of Tests.....	18

Application Validation through Unit Testing

Objectives

The objective of this lab is to understand how to take advantage of the Unit Testing tools available in Visual Studio 2008.

In this lab, you will learn how to:

- Generate unit tests from existing code.
- Author unit tests based off pre-existing assemblies
- Use data to drive the unit tests
- View the test results
- Enable code coverage and examine how well the unit tests are exercising the assembly under test.

Estimated Time to Complete This Lab

30 Minutes

Computers used in this Lab



VSTSRTM08

The password for the Administrator account on all computers in this lab is:


P2ssw0rd

Exercise 1

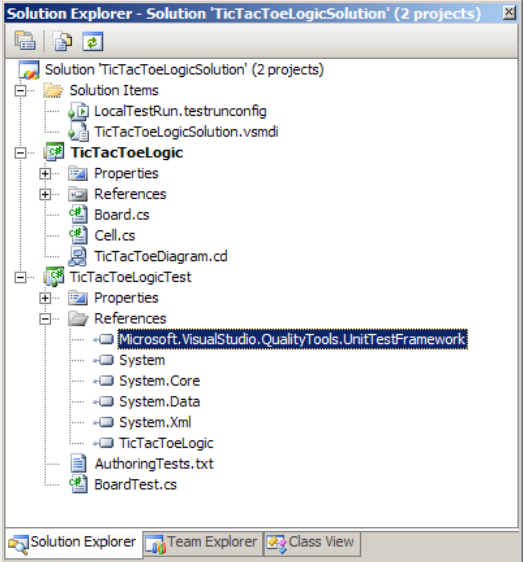
Generate Unit Tests

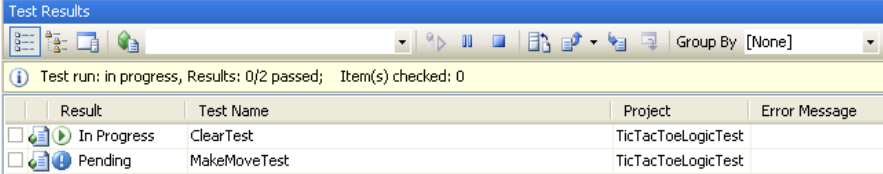
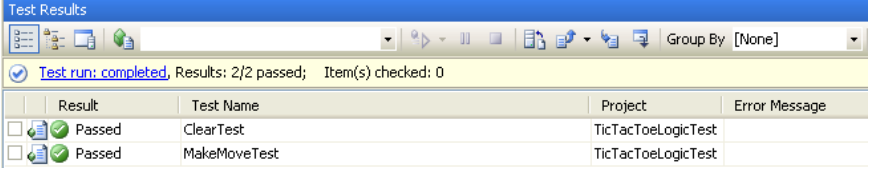
Scenario

In this exercise, you will learn how to generate unit tests from the source code of the class you want to test. In addition, you'll learn how to author the unit tests manually.

Tasks	Detailed Steps
<p>Complete the following tasks on:</p> <p> VSTSRTM08</p> <p>1. Setup</p>	<p>a. Open Microsoft Visual Studio 2008</p> <p>b. Select File Open Project/Solution.</p> <p>c. Navigate to C:\MSLabs\VSTS 2008\Work\200\UnitTesting\TicTacToeLogicSolution \ and open the TicTacToeLogicSolution.sln solution.</p>
<p>2. Generate Tests for the Board Class</p>	<p>a. Open the file Board.cs. In that file you will find the methods Clear() and MakeMove(int, char), these are the methods we will be generating tests for.</p> <p>b. Right click anywhere inside the Clear() method and select Create Unit Tests...</p> <div data-bbox="553 863 1365 1304" data-label="Image"> </div> <p>c. A dialog will appear which will prompt you to select the methods that tests should be generated for and where the tests should be placed. Since the cursor was in the Clear() method when Create Unit Tests... was selected, the Clear() method is already checked in the dialog.</p>

Tasks	Detailed Steps
	<div data-bbox="574 191 1341 762" data-label="Image"> </div> <p data-bbox="505 825 1380 930"> d. Select Clear and MakeMove for test generation. e. Verify that you have selected Create a new Visual C# test project... for the Output project. </p> <div data-bbox="505 947 1438 1209" data-label="Text" style="background-color: #f0f0f0; padding: 5px;"> <p><i>Note: To modify configuration settings for test generation, select the Settings... button. This allows you to change the default naming options for test files, classes, and methods that will be generated. It also allows you to specify options for how you want the test methods to be stubbed out.</i></p> <p><i>By default, the items that are displayed in the window are all available classes and members that are in the project. If you were to select Filter, you can change the criteria of what is displayed to show only public items and also show base class items that are available for test generation.</i></p> </div> <p data-bbox="505 1220 1438 1356"> f. Click OK to continue with the test generation. g. Since the option to create a new test project was selected, a dialog box will pop up prompting you to name your new project. Type TicTacToeLogicTest as the project name and press Create. </p> <div data-bbox="605 1371 1344 1604" data-label="Image"> </div> <p data-bbox="505 1619 1401 1896"> h. Select Project TicTacToeLogicTest Properties to open the Project designer. On the Application tab, change the Default Namespace property to VstsTraining.Test. i. Select File Save Selected Items and then File Close. j. Notice the reference to Microsoft.VisualStudio.QualityTools.UnitTestFramework and TicTacToeLogic have already been added. Your solution should now look something like the following: </p>

Tasks	Detailed Steps
	
<p>3. Add test logic</p>	<p>a. Open the newly created BoardTest.cs file and take a moment to examine it. One thing that should be pointed out is that the methods ClearTest() and MakeMoveTest() have been created and some basic test code has already been created. Note that both tests have an <code>Assert.Inconclusive</code> method call in them. This method is used instead of marking the tests as failures and should only be used when a test really hasn't been implemented yet.</p> <p>b. Navigate to the MakeMoveTest() method and let us implement that one first. Modify the method using the following example as a guideline:</p> <pre data-bbox="506 1050 1399 1495"> /// <summary> ///A test for MakeMove (int, char) ///</summary> [TestMethod()] public void MakeMoveTest() { Board target = new Board(); int cellNumber = 3; char cellValue = 'X'; target.MakeMove(cellNumber, cellValue); } </pre> <p><i>Note: Since the expected behavior of MakeMove(int, char) is to throw an exception if a move is not valid, this test is considered to be a success if no exception is thrown.</i></p> <p>c. Navigate up to the ClearTest() method. This test is a little more tricky to implement since there is no way to actually view the Cells property, but since it should be impossible for both an X and an O to occupy the same cell, if we put an X in a cell, call Clear() and then put an O in that same cell and an exception is not thrown, it can be assumed that the cells were cleared out appropriately. Modify ClearTest() as follows:</p> <pre data-bbox="506 1768 1399 1890"> /// <summary> ///A test for Clear () ///</summary> [TestMethod()] </pre>


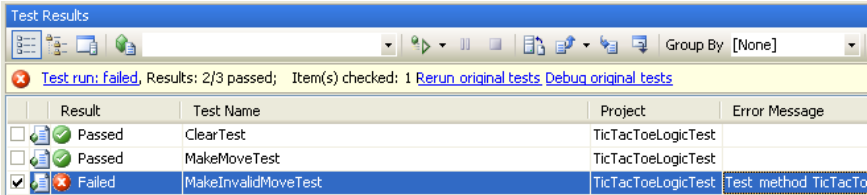
Tasks	Detailed Steps
	<pre data-bbox="506 197 954 508"> public void ClearTest() { Board target = new Board(); target.MakeMove(5, '0'); target.Clear(); target.MakeMove(5, 'X'); } </pre> <p data-bbox="506 512 1396 571">d. In Solution Explorer, right-click on the TicTacToeLogicTest and set it as the Startup Project, then press Ctrl-F5 to run the tests.</p> <p data-bbox="506 588 1396 646">e. The Test Results window will display and show something like the following while the tests are running</p>  <p data-bbox="506 890 1396 919">f. After the tests have completed, the window should show something similar to:</p>  <p data-bbox="506 1155 1396 1213">g. Exercise 5 will cover this window in more detail later on, but for now, it should be pointed out that at least both tests are passing.</p>

Exercise 2

Author Unit Tests

Scenario

In this exercise, you will be creating unit tests manually to further exercise the code. Authoring unit tests comes in handy when either you don't have access to the source code in order to generate unit tests, or when the generated unit tests don't quite exercise the code completely.

Tasks	Detailed Steps
<p>Complete the following tasks on:</p> <p> VSTSRTM08</p> <p>1. Create a test that expects an exception</p>	<p>a. Open the BoardTest.cs file.</p> <p>b. In the previous exercise, we made the statement that the MakeMove method would throw an exception if an error occurred while trying to make the specific move. Making statements is nice, but it would be better if we had a test to back up that statement. Let's do that now by defining MakeInvalidMoveTest().</p> <p>c. Paste the following code at the bottom of the file, creating a new method named MakeInvalidMoveTest(). Here we will test the theory that an exception is thrown if both an 'X' and 'O' are placed in the same location on the board.</p> <pre data-bbox="506 871 1398 1157"> [TestMethod()] public void MakeInvalidMoveTest() { Board target = new Board(); target.MakeMove(8, 'X'); target.MakeMove(8, 'O'); } </pre> <p><i>Note: This method performs the logic we want, but now we must tell the Microsoft testing framework that this is a test method. To do that, notice that we must decorate the method with the [TestMethod()] attribute.</i></p> <p>d. Press Ctrl-F5 to run the suite of tests again.</p> <p>e. This time a failure has occurred and the results should look like:</p>  <p>f. This test is failing because an exception is occurring, but we wanted this exception to occur, so we need to tell the test an exception is ok. This is done by adding the [ExpectedException] attribute to the method. The test method should now look like</p> <pre data-bbox="506 1776 1398 1896"> [TestMethod()] [ExpectedException(typeof(System.ApplicationException))] public void MakeInvalidMoveTest() { </pre>


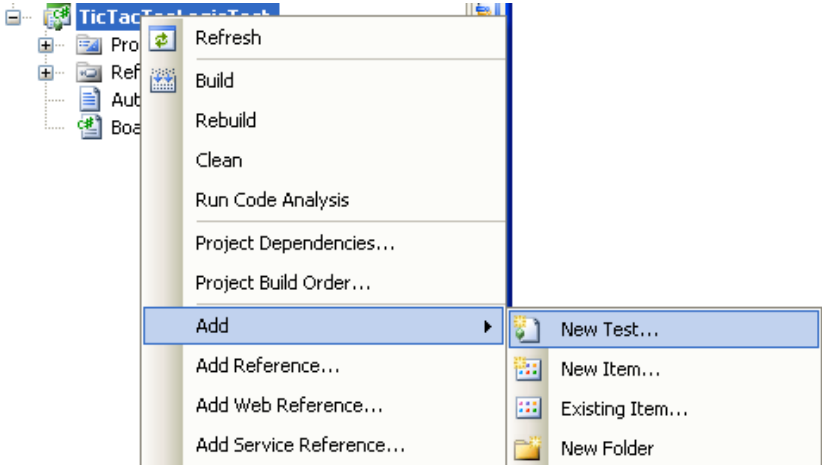
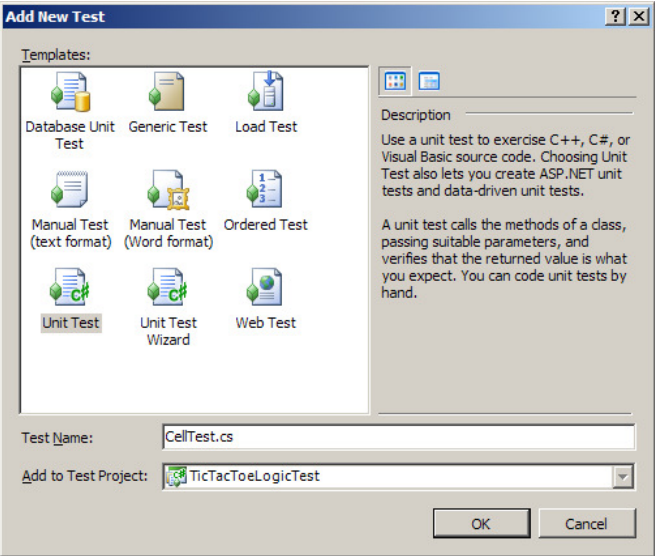
Tasks	Detailed Steps
	<pre>Board target = new Board(); target.MakeMove(8, 'X'); target.MakeMove(8, 'O'); }</pre> <p>g. Press Ctrl-F5 again to rerun the tests and verify that all three tests are now passing in the Test Results window.</p>
<p>2. Explore other test attributes</p>	<p>a. Open the BoardTest.cs file.</p> <p><i>Note: One thing that was common in all three test methods we created was the creation of the new Board class. It would be nice if code common to all tests could be placed in one central location and be guaranteed to be run before each test. The testing framework provided by Microsoft allows for just such a thing in a method decorated with the [TestInitialize()] attribute.</i></p> <p>b. Expand the Additional test attributes code section in the BoardTest.cs file.</p> <p>c. Navigate to the commented section of code that defines the MyTestInitialize() method and uncomment the whole method.</p> <p>d. Add a line of code to the method, assigning a new Board object to the target variable. The code should then look like the following example:</p> <pre>[TestInitialize()] public void MyTestInitialize() { target = new Board(); }</pre> <p>e. At the top of the BoardTest class, create a property of type Board and name it Board. The code should look similar to:</p> <pre>internal Board target { get { return _target; } set { _target = value; } } private Board _target;</pre> <p>f. In the three methods created earlier, remove any construction of the Board class inside of the test methods because the construction will now be done for us in MyTestInitialize before each test is run.</p> <p>g. Press Ctrl-F5 to launch the tests and verify that all three pass.</p> <p><i>Note: Examine the other commented methods inside of the Additional test attributes section. Methods marked with the [TestCleanup()] attribute run at the end of every test whether an exception is thrown or not. [ClassInitialize()] and [ClassCleanup()] are similar to their corresponding [Test...] attributes except these run when the test class is created and destroyed.</i></p>

Exercise 3

Test non-public data

Scenario

In this exercise, you will be creating unit tests that will exercise non-public data. While the debate is still on as to whether this is a good practice or not, Microsoft has provided the ability to easily perform such tasks, which do deserve mentioning. Utilizing this tool will allow you to test non-public classes as well as non-public methods on all classes.

Tasks	Detailed Steps
<p>Complete the following tasks on:</p> <p> VSTSRM08</p> <p>1. Create a new unit test class</p>	<p>a. In Solution Explorer, right click on the TicTacToeLogicTest project and select Add New Test.</p>  <p>b. The Add New Test dialog pops up and under Templates: select Unit Test</p> 

Tasks	Detailed Steps
	<ul style="list-style-type: none"> c. For Test Name type in CellTest.cs and press OK. d. After CellTest.cs is opened, type using VstsTraining; at the top of the file.
<p>2. Create and use a private accessor for Cell class</p>	<ul style="list-style-type: none"> a. Open Cell.cs in the TicTacToeLogic project and right-click anywhere in the editor to bring up the following menu <div data-bbox="609 367 1120 808" style="border: 1px solid gray; padding: 5px; margin: 10px 0;"> </div> b. Select Create Private Accessor TicTacToeLogicTest. This will create a private accessor for the Cell class and place it in the TicTacToeLogicTest assembly. c. Open up Solution Explorer and notice that a new file has been added to the test project called TicTacToeLogic.accessor. This file contains the necessary reflection logic in order to exercise the non-public code. d. Navigate to TestMethod1 (within the CellTest.cs file) which was created for you when you created the CellTest class, and change the signature and body to look like the following code block. <pre data-bbox="519 1150 1396 1501" style="background-color: #f0f0f0; padding: 10px; margin: 10px 0;"> [TestMethod] public void ValidateXValueIsAllowed() { VstsTraining.Cell_Accessor accessor = new VstsTraining.Cell_Accessor(); accessor.Value = 'X'; Assert.AreEqual<char>('X', (char)accessor.Value); } </pre> e. Press Ctrl-F5 in order to execute the tests.
<p>3. Regenerate the private accessor</p>	<ul style="list-style-type: none"> a. One thing to point out is that since the VstsTraining_CellAccessor class is generated from the original Cell class, any modifications made to the Cell class that you would like to test will require you to manually re-generate the VstsTraining_CellAccessor class. b. To re-generate the private accessor for Cell, open Cell.cs. c. Right-click anywhere in the file and select Create Private Accessor TicTacToeLogicTest. <p data-bbox="503 1806 1429 1900" style="background-color: #f0f0f0; padding: 5px;"><i>Note: The private accessor has now been updated to reflect the latest version of the Cell class. One thing to point out is that if any non-public items that are under test have had their names changed since the last time the private accessor was re-</i></p>

Application Validation through Unit Testing


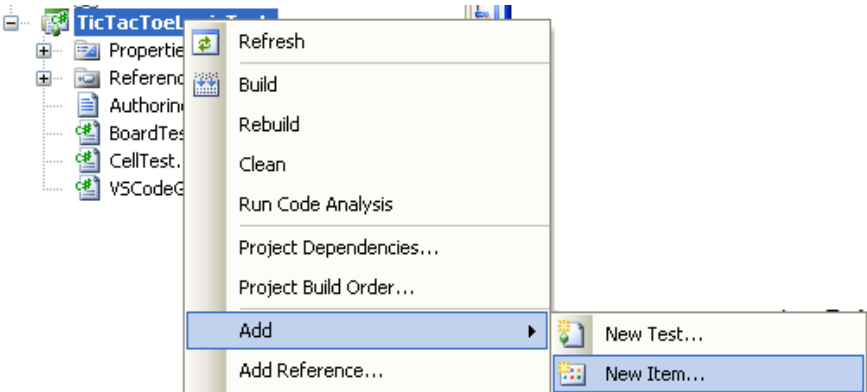
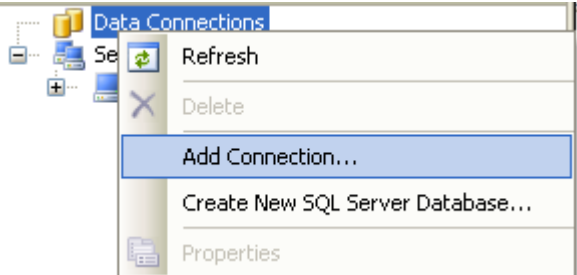
Tasks	Detailed Steps
	<i>generated, there might be compiler errors in the test code now.</i>

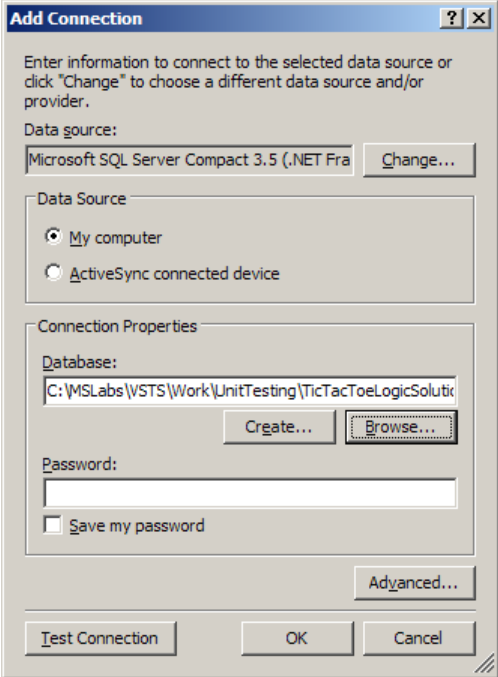
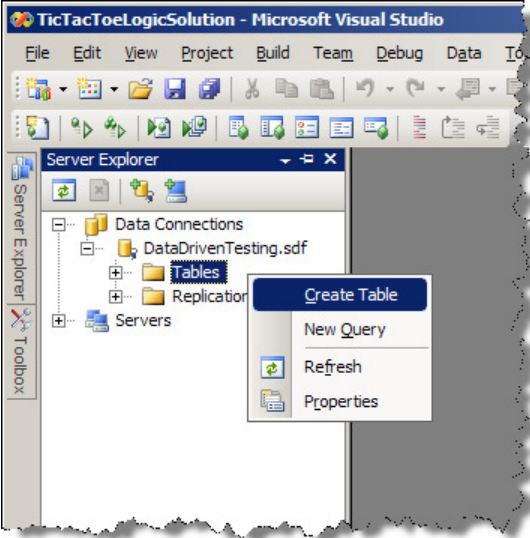
Exercise 4

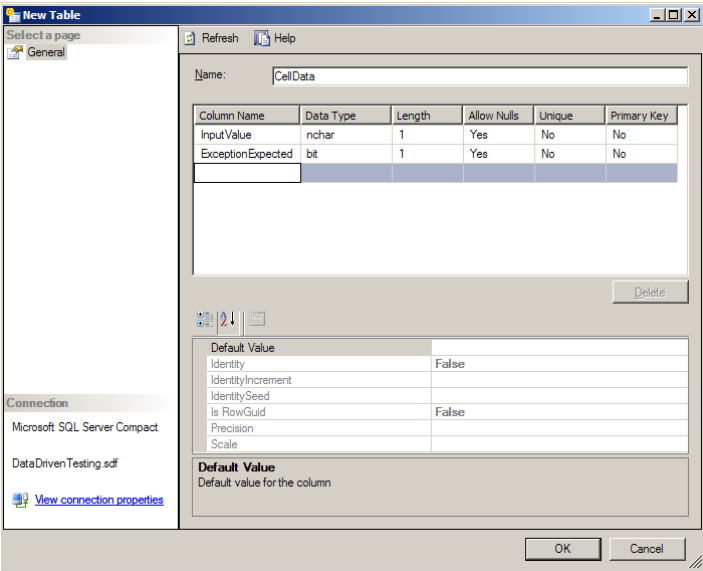
Create Data-Driven Unit Tests

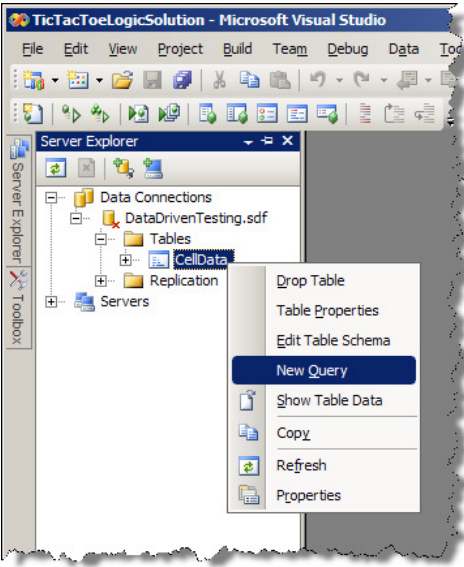
Scenario

In this exercise, you will create unit tests that retrieve their test values and expected answers from a database. Data-driven testing is convenient when there is an API that needs to be tested with multiple input values. Instead of writing multiple tests that just have different data and the same logic, the data can be pulled from a database and the logic will only be written once. In this example, we will be setting up the data-driven testing all through code, but this can also be performed from the test view by selecting an existing test and pressing **F4** to get to the test properties.

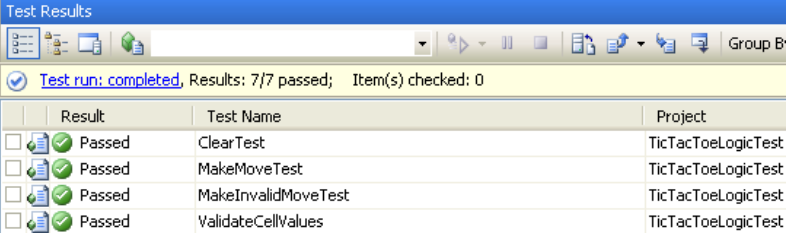
Tasks	Detailed Steps
<p>Complete the following tasks on:</p> <p> VSTSRTM08</p> <p>1. Create the database</p>	<p>a. Make sure that Microsoft SQL Server Compact 3.5 is installed.</p> <p>b. Right-click on the TicTacToeLogicTest project and select Add New Item.</p>  <p>c. Select Local Database from the Visual Studio installed templates section.</p> <p>d. Name the database <code>DataDrivenTesting.sdf</code> and press Add.</p> <p>e. Open the Server Explorer by selecting View Server Explorer.</p> <p>f. Right-click on Data Connections within Server Explorer and select Add Connection...</p>  <p>g. You may initially see the "Choose Data Source" dialog, and if so, you should select Microsoft SQL Server Compact 3.5, and press Continue to bring up the "Add Connection" dialog. Otherwise, you will immediately see the "Add Connection" dialog, and if so, click on the Change... button select Microsoft SQL Server Compact 3.5, and press OK.</p> <p>h. Press the Browse... button and select the <code>C:\MSLabs\VSTS 2008\Work\200\UnitTesting\TicTacToeLogicSolution\</code></p>

Tasks	Detailed Steps
	<p>TicTacToeLogicTest\DataDrivenTesting.sdf file.</p> <p>i. The connection window should now look like the one below.</p>  <p>j. Press OK to finalize the connection.</p> <p>k. After the database is created, expand the newly created DataDrivenTesting database tree in Server Explorer. Right click on the Tables folder and select Create Table.</p>  <p>l. Set the Name to <u>CellData</u>.</p> <p>m. Create two columns</p> <ul style="list-style-type: none"> • ColumnName: InputValue, Data Type: nchar, Length: 1, Allow Nulls: Yes • ColumnName: ExceptionExpected, Data Type: bit, Allow Nulls: Yes

Tasks	Detailed Steps
	<p>n. Click OK to save the table as <u>CellData</u>.</p> 
<p>2. Create a data-driven test</p>	<p>a. Open CellTest.cs.</p> <p>b. Navigate to the ValidateXValueIsAllowed method. Instead of creating more tests that will just test other valid and invalid values for the Cell.Value property, we will instead allow our database table to drive this one test with various values.</p> <p>c. Add the DataSource attribute to the ValidateXValueIsAllowed method. This not only tells the test to be a data-driven test, but it also tells the test where to retrieve its data from. The first parameter tells the test what type of data adapter to use, the second parameter is the connection string, the third parameter specifies what table to use from the database for the test, and the fourth tells how the data should be retrieved and used.</p> <pre>[DataSource("System.Data.SqlClient", "data source= DataDirectory \\DataDrivenTesting.sdf", "CellData", DataAccessMethod.Sequential), DeploymentItem("TicTacToeLogicTest\\DataDrivenTesting.sdf")]</pre> <p>d. Rename the method from ValidateXValueIsAllowed to ValidateCellValues. This needs to be performed since we will now be testing more than just the 'X' as the value.</p> <p>e. Change the body of the ValidateCellValues method to look like the code below.</p> <pre>[DataSource("System.Data.SqlClient", "data source= DataDirectory \\DataDrivenTesting.sdf", "CellData", DataAccessMethod.Sequential), DeploymentItem("TicTacToeLogicTest\\DataDrivenTesting.sdf"), TestMethod] public void ValidateCellValues() { VstsTraining.Cell_Accessor accessor = new VstsTraining.Cell_Accessor(); char inputValue = Convert.ToChar(TestContext.DataRow["InputValue"]);</pre>

Tasks	Detailed Steps								
	<pre> try { accessor.Value = inputValue; Assert.AreEqual<char>(inputValue, (char)accessor.Value); } catch (Exception) { if (!Convert.ToBoolean(TestContext.DataRow["ExceptionExpected"])) { Assert.Fail("An exception was thrown when one was not expected."); } } } </pre>								
<p>3. Populate the database and execute the test</p>	<p>a. Navigate to Server Explorer and expand the tree view of the DataDrivenTesting.sdf node until you see the CellData table.</p> <p>b. Right click on the CellData table and select Show Table Data</p>  <p>c. Populate the table with the following data</p> <table border="1" data-bbox="561 1591 1003 1759"> <thead> <tr> <th>InputValue</th> <th>ExceptionExpected</th> </tr> </thead> <tbody> <tr> <td>X</td> <td>False</td> </tr> <tr> <td>O</td> <td>False</td> </tr> <tr> <td>Y</td> <td>True</td> </tr> </tbody> </table> <p>d. Press Ctrl-F5 to execute the tests. All tests should pass and the Test Results window should look similar to Figure below</p>	InputValue	ExceptionExpected	X	False	O	False	Y	True
InputValue	ExceptionExpected								
X	False								
O	False								
Y	True								

Application Validation through Unit Testing


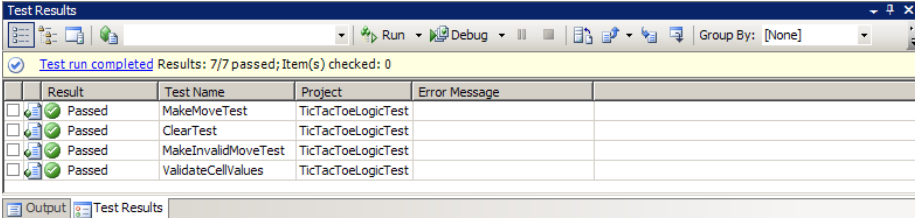
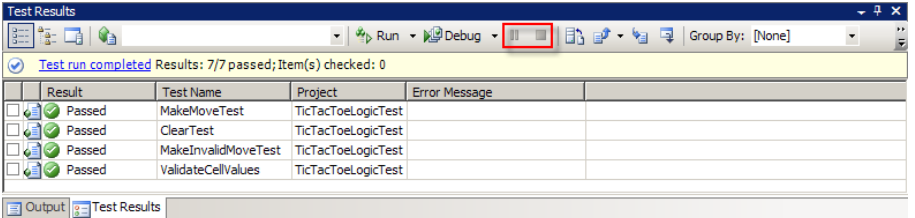
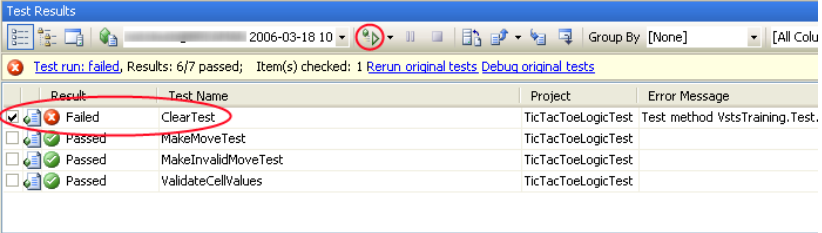
Tasks	Detailed Steps															
	 <p>The screenshot shows a 'Test Results' window with a table of test results. The table has four columns: 'Result', 'Test Name', and 'Project'. All four tests listed are 'Passed' and belong to the 'TicTacToeLogicTest' project.</p> <table border="1"><thead><tr><th>Result</th><th>Test Name</th><th>Project</th></tr></thead><tbody><tr><td>Passed</td><td>ClearTest</td><td>TicTacToeLogicTest</td></tr><tr><td>Passed</td><td>MakeMoveTest</td><td>TicTacToeLogicTest</td></tr><tr><td>Passed</td><td>MakeInvalidMoveTest</td><td>TicTacToeLogicTest</td></tr><tr><td>Passed</td><td>ValidateCellValues</td><td>TicTacToeLogicTest</td></tr></tbody></table>	Result	Test Name	Project	Passed	ClearTest	TicTacToeLogicTest	Passed	MakeMoveTest	TicTacToeLogicTest	Passed	MakeInvalidMoveTest	TicTacToeLogicTest	Passed	ValidateCellValues	TicTacToeLogicTest
Result	Test Name	Project														
Passed	ClearTest	TicTacToeLogicTest														
Passed	MakeMoveTest	TicTacToeLogicTest														
Passed	MakeInvalidMoveTest	TicTacToeLogicTest														
Passed	ValidateCellValues	TicTacToeLogicTest														

Exercise 5

Viewing Test Results

Scenario

Thus far we have done quite a bit of testing, but we really haven't taken the time to view the results except for taking a look at the **Test Results** window briefly. This exercise will examine the **Test Results** window at a deeper level which will allow you to really examine what is going on for each test that is executed.


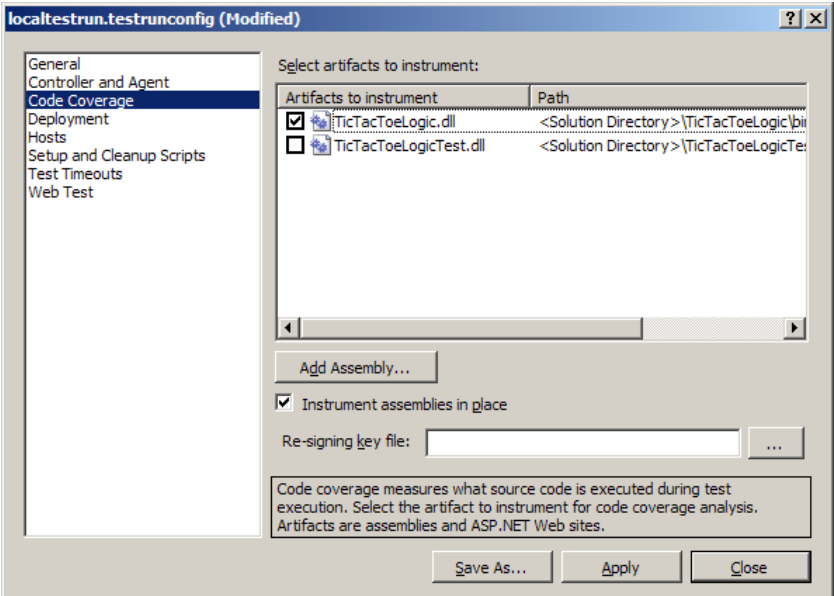
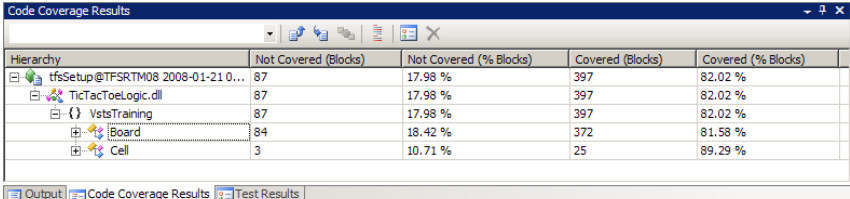
Tasks	Detailed Steps
<p>Complete the following tasks on:</p> <p> VSTSRTM08</p> <p>1. Getting to know the Test Results window</p>	<p>a. From the tests run in the previous exercise, take a look at the Test Results window. This window is the portal to all the information you will need when tests are being run and after the test run has completed.</p>  <p>b. From this window, tests that are currently running can be paused, resumed, or even stopped. This functionality is only enabled when tests are actually running, which is why the buttons are currently grayed out.</p>  <p>c. The Test Results window also allows you to view previous test runs. These are accessible via the dropdown list located right next to the play/pause/stop buttons</p> <p>d. When tests are run, if for some reason a test fails, the Test Results window will automatically have that test checked, making it easy for you to fix the code and then just re-run that particular test. To run only that test, instead of pressing Ctrl-F5 (or F5) to run the tests, you would just press the Play button. This button is enabled in Figure below because a test in the window is selected.</p> 
<p>2. Get test details for a particular test</p>	<p>a. Double click on the ClearTest entry in the Test Results window. This will bring up test details for that particular test method. From this window you can see,</p>

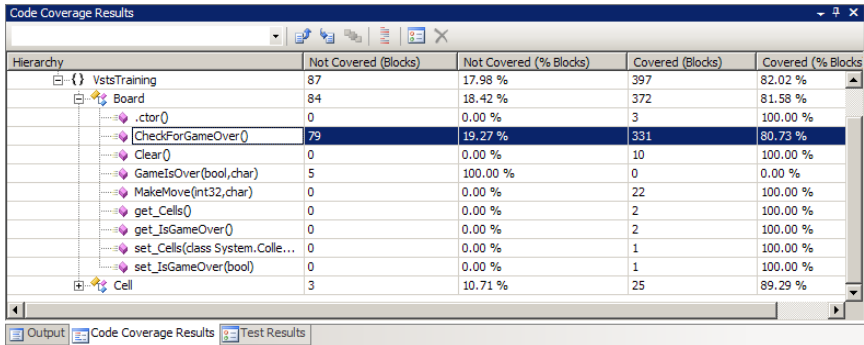
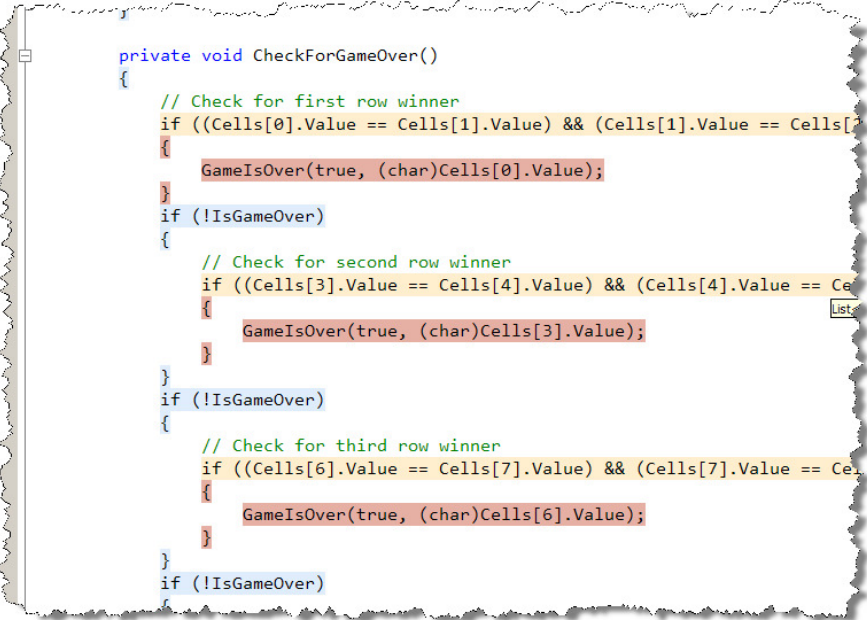
Exercise 6

Examining Code Coverage of Tests

Scenario

The previous exercises have all focused on how to test the code, but in this exercise we will find exactly what code gets tested with the tests, and more importantly, what code is not being tested at all. While 100% code coverage doesn't guarantee that the code is bug free, at least it shows that you are hitting all code paths.

Tasks	Detailed Steps																														
<p>Complete the following tasks on:</p> <p> VSTSRTM08</p> <p>1. Enable code coverage</p>	<p>a. Open Solution Explorer and double-click on the localtestrun.testrunconfig file.</p> <p>b. Select Code Coverage from the listbox and select the TicTacToeLogic.dll artifact to instrument. There is no need to run code coverage on the test assembly.</p>  <p>c. Click Apply and Close.</p> <p>d. Press Ctrl-F5 to run the tests again. This test run will collect code coverage information since it is now enabled.</p>																														
<p>2. View code coverage results</p>	<p>a. From the Test menu, select Windows Code Coverage Results.</p> <p>b. Expand the tree view in the Code Coverage Results window so that both the Board and Cell class can be seen.</p>  <table border="1" data-bbox="539 1654 1383 1780"> <thead> <tr> <th>Hierarchy</th> <th>Not Covered (Blocks)</th> <th>Not Covered (% Blocks)</th> <th>Covered (Blocks)</th> <th>Covered (% Blocks)</th> </tr> </thead> <tbody> <tr> <td>tfsSetup@TFSRTM08 2008-01-21 0...</td> <td>87</td> <td>17.98 %</td> <td>397</td> <td>82.02 %</td> </tr> <tr> <td> TicTacToeLogic.dll</td> <td>87</td> <td>17.98 %</td> <td>397</td> <td>82.02 %</td> </tr> <tr> <td> VstsTraining</td> <td>87</td> <td>17.98 %</td> <td>397</td> <td>82.02 %</td> </tr> <tr> <td> Board</td> <td>84</td> <td>18.42 %</td> <td>372</td> <td>81.58 %</td> </tr> <tr> <td> Cell</td> <td>3</td> <td>10.71 %</td> <td>25</td> <td>89.29 %</td> </tr> </tbody> </table> <p>c. Well, most of the code was covered in the Cell class, but as you can quickly see, 84 blocks were not covered in the Board class.</p>	Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)	tfsSetup@TFSRTM08 2008-01-21 0...	87	17.98 %	397	82.02 %	TicTacToeLogic.dll	87	17.98 %	397	82.02 %	VstsTraining	87	17.98 %	397	82.02 %	Board	84	18.42 %	372	81.58 %	Cell	3	10.71 %	25	89.29 %
Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)																											
tfsSetup@TFSRTM08 2008-01-21 0...	87	17.98 %	397	82.02 %																											
TicTacToeLogic.dll	87	17.98 %	397	82.02 %																											
VstsTraining	87	17.98 %	397	82.02 %																											
Board	84	18.42 %	372	81.58 %																											
Cell	3	10.71 %	25	89.29 %																											

Tasks	Detailed Steps
	<p>d. Finish expanding the Board class to figure out where the code is that was not covered.</p>  <p>e. In order to figure out what lines of code were never tested, double-click on CheckForGameOver. This will open the Board.cs file within the TicTacToeLogic project, and shows the lines colored with red lines being code paths that were never run, and green paths showing what code was executed. If your screen doesn't show the colors, close the Board.cs file and double-click CheckForGameOver again.</p>  <pre> private void CheckForGameOver() { // Check for first row winner if ((Cells[0].Value == Cells[1].Value) && (Cells[1].Value == Cells[2].Value)) { GameIsOver(true, (char)Cells[0].Value); } if (!IsGameOver) { // Check for second row winner if ((Cells[3].Value == Cells[4].Value) && (Cells[4].Value == Cells[5].Value)) { GameIsOver(true, (char)Cells[3].Value); } } if (!IsGameOver) { // Check for third row winner if ((Cells[6].Value == Cells[7].Value) && (Cells[7].Value == Cells[8].Value)) { GameIsOver(true, (char)Cells[6].Value); } } if (!IsGameOver) { // Check for column winners } } </pre>