CHAPTER 59

Testing Code with Unit Tests, Test-Driven Development, and Code Contracts

IN THIS CHAPTER

- Testing Code with Unit Tests
- Introducing Test-Driven Development
- Understanding Code Contracts

When you purchase new software, you expect that the software works. I'm pretty sure you don't like applications you buy to cause unexpected crashes or errors due to apparently unhandled situations. The same is for users purchasing your software or for colleagues in your company performing their daily work through your applications, so you need to pay particular attention to check if and how your code works. Although implementing error handling routines is fundamental, another important moment in application development is testing code. Testing allows checking for code blocks' correct behavior under different situations, and it should be the most deep possible. In big development teams, testers play an important role, so they need to have good tools for successfully completing their work. In this chapter you learn about the Visual Studio tools for testing code, starting from unit tests until the new Code Contracts library, passing through the Test-Driven Development approach. You also see how such tooling can be successfully used even if you are a single developer.

Testing Code with Unit Tests

Unit tests enable testing code portions outside the application context to check if they work correctly so that testing is basically abstracted from the application. Typically you create a test project, where there are classes and methods that encapsulate and isolate the original code so that you can test it in a kind of isolated sandbox without editing the

source project. Visual Studio 2010 is the ideal environment for performing unit tests, so this section explains how you can accomplish this important task.

Creating Unit Tests

First, you need some code to test. Imagine you have a `Rectangle` class that exposes methods for math calculations on a rectangle's perimeter and area. Create a new class library, name it **UnitTestDemo**; then rename Class1.vb as **Rectangle.vb** and write the following code:

```
Class Rectangle


    Shared Function CalculatePerimeter(ByVal sideA As Double,
                                       ByVal sideB As Double) As Double
        Return (sideA * 2) + (sideB * 2)
    End Function

    Shared Function CalculateArea(ByVal sideA As Double,
                                  ByVal sideB As Double) As Double
        Return sideA * sideB
    End Function
End Class
```

Imagine you want to test both methods to check if they work correctly but inside an isolated environment, abstracted from the original project. In the code editor select both methods; then right-click and select **Create Unit Tests**. This launches the **Create Unit Tests** dialog, where you can select objects to add to the test project. Expand the `UnitTestDemo` namespace and select both methods, as demonstrated in Figure 59.1.

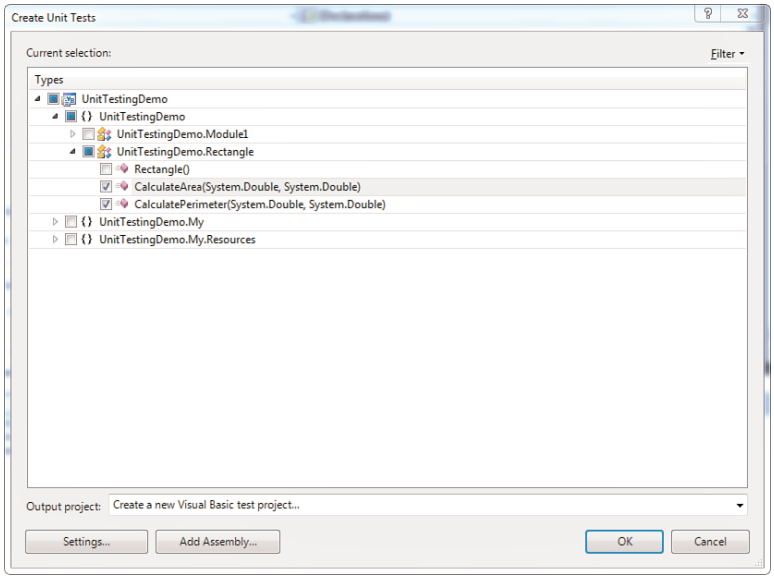

FIGURE 59.1 Choosing methods to be added to unit tests.

When you click OK, you will be asked to specify a new test project name. For this example leave unchanged the default selection and go ahead. Visual Studio will also raise a warning message because in this case we are creating a unit test for a class marked as `Friend`, asking if you want to mark the test project with the `InternalsVisibleTo` attribute. You can click Yes in order to decorate the project with such an attribute. When Visual Studio finishes generating the new project, you notice a `RectangleTest` test class whose content is reported in Listing 59.1.

LISTING 59.1 The Newly Generated Test Class

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports UnitTestingDemo


'''<summary>
'''This is a test class for RectangleTest and is intended
'''to contain all RectangleTest Unit Tests
'''</summary>
<TestClass()> _
Public Class RectangleTest

    Private testContextInstance As TestContext

    '''<summary>
    '''Gets or sets the test context which provides
    '''information about and functionality for the current test run.
    '''</summary>
    Public Property TestContext() As TestContext
        Get
            Return testContextInstance
        End Get
        Set(ByVal value As TestContext)
            testContextInstance = Value
        End Set
    End Property

#Region "Additional test attributes"
    '
    'You can use the following additional attributes as you write your tests:
    '
    'Use ClassInitialize to run code before running the first test in the class
    '<ClassInitialize()>  _
    'Public Shared Sub MyClassInitialize(ByVal testContext As TestContext)
    'End Sub
    '
    'Use ClassCleanup to run code after all tests in a class have run
```

```
    '<ClassCleanup()>  _
    'Public Shared Sub MyClassCleanup()
    'End Sub
    '
    'Use TestInitialize to run code before running each test
    '<TestInitialize()>  _
    'Public Sub MyTestInitialize()
    'End Sub
    '
    'Use TestCleanup to run code after each test has run
    '<TestCleanup()>  _
    'Public Sub MyTestCleanup()
    'End Sub
    '
#End Region
    '''<summary>
    '''A test for CalculateArea
    '''</summary>
    <TestMethod()> _
        Public Sub CalculateAreaTest()
        Dim sideA As Double = 0.0! ' TODO: Initialize to an appropriate value
        Dim sideB As Double = 0.0! ' TODO: Initialize to an appropriate value
        Dim expected As Double ' TODO: Initialize to an appropriate value
        Dim actual As Double
        actual = Rectangle.CalculateArea(sideA, sideB)
        Assert.AreEqual(expected, actual)
        Assert.Inconclusive("Verify the correctness of this test method.")
    End Sub

    '''<summary>
    '''A test for CalculatePerimeter
    '''</summary>
    <TestMethod()> _
        Public Sub CalculatePerimeterTest()
        Dim sideA As Double = 0.0! ' TODO: Initialize to an appropriate value
        Dim sideB As Double = 0.0! ' TODO: Initialize to an appropriate value
        Dim expected As Double ' TODO: Initialize to an appropriate value
        Dim actual As Double
        actual = Rectangle.CalculatePerimeter(sideA, sideB)
        Assert.AreEqual(expected, actual)
        Assert.Inconclusive("Verify the correctness of this test method.")
    End Sub
End Class
```

For code in Listing 59.1, there are some aspects to consider:

- The `TextContext` object, known as context, represents the isolated box where unit tests are executed.
- A test class is decorated with the `TestClass` attribute while test methods are decorated with the `TestMethod` attribute.
- By default Visual Studio provides result comparisons via the `Assert.AreEqual` method that checks for parameters' equality, but you are not limited to this particular operation. You can choose which of the `Assert` class static methods is the most appropriate for your needs.
- Test methods cannot be shared while they must be public, accept no parameter, and return no value. This is the reason why values must be initialized within method bodies. By default Visual Studio assigns zero or null values that you have to replace with valid ones.

Also notice that the `Assert.Inconclusive` statements are placed as a way for communicating that the method implementation has not been completed yet, and they will be commented before running tests. The idea of unit testing is comparing the expected result with the actual result of an action. With that said, first comment the `Assert.Inconclusive` statements; then replace the methods' code as follows:

```
<TestMethod()> _
    Public Sub CalculateAreaTest()
    Dim sideA As Double = 10 ' TODO: Initialize to an appropriate value
    Dim sideB As Double = 20 ' TODO: Initialize to an appropriate value
    Dim expected As Double = 2000 ' TODO: Initialize to an appropriate value
    Dim actual As Double
    actual = Rectangle.CalculateArea(sideA, sideB)
    Assert.AreEqual(expected, actual)
    'Assert.Inconclusive("Verify the correctness of this test method.")
End Sub

<TestMethod()> _
    Public Sub CalculatePerimeterTest()
    Dim sideA As Double = 10 ' TODO: Initialize to an appropriate value
    Dim sideB As Double = 20 ' TODO: Initialize to an appropriate value
    Dim expected As Double = 60 ' TODO: Initialize to an appropriate value
    Dim actual As Double
    actual = Rectangle.CalculatePerimeter(sideA, sideB)
    Assert.AreEqual(expected, actual)
    'Assert.Inconclusive("Verify the correctness of this test method.")
End Sub
```

Notice that the first method voluntarily causes an error, to demonstrate how unit testing works. The expected value is in fact greater than it should be. At this point you are ready to run both unit tests.

Running Unit Tests

When you create unit tests, Visual Studio automatically shows and anchors the Test Tools toolbar that contains buttons for executing, debugging, and managing unit tests. You can choose to run a single unit test or multiple ones. For the current example, click the **Run All Tests in Solution** button. When all unit tests complete, the Test Results tool window shows a report about test success or failures. Figure 59.2 shows how such a window looks for the current example.

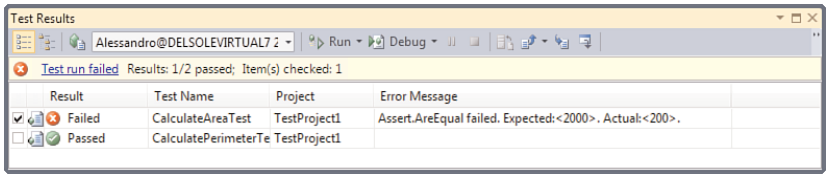

FIGURE 59.2 Viewing unit test results.

Notice how the `CalculateAreaTest` method failed while `CalculatePerimeterTest` succeeded. The Error Message column provides details on the occurred error so that you can fix it. In this case the equality check failed because the method returned a result different from the expected one. You can also get detailed failure information, by clicking the **Test Run Failed** hyperlink. Figure 59.3 shows the failure summary, where you can get information on the test name, server, and timestamp.

NOTE ON FIXING ERRORS

In a typical real-life scenario, you will not edit the expected result to make a unit test work, whereas you will instead fix errors in the code. The example proposed in this chapter is a demo scenario, and its purpose is explaining how unit test works. This is the reason why here you are about to fix the expected result, but in real-world applications the expected result will remain unvaried.

In the `CalculateAreaTest` method, replace the expected declaration as follows:

```
Dim expected As Double = 200
```

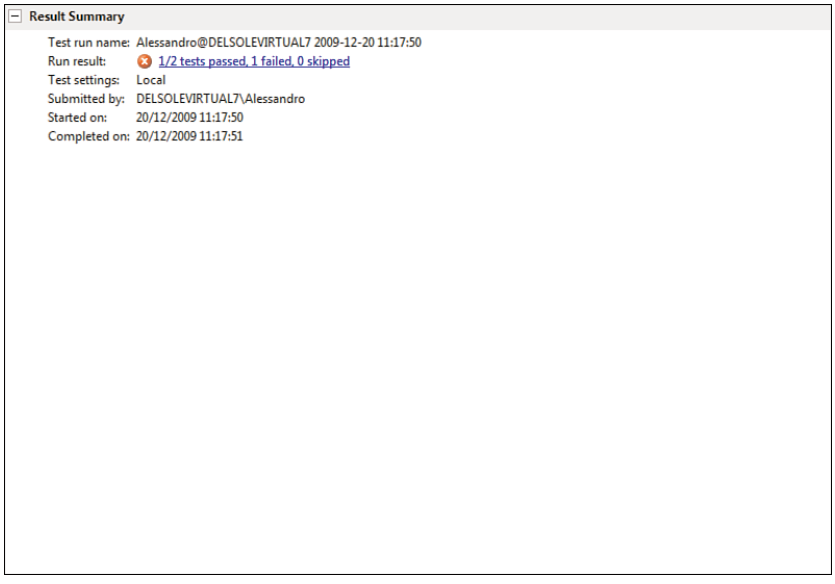

FIGURE 59.3 Viewing the test failure summary.

Now run again both unit tests. At this point both tests pass because in both cases expected value and actual value are equal, as demonstrated in Figure 59.4.

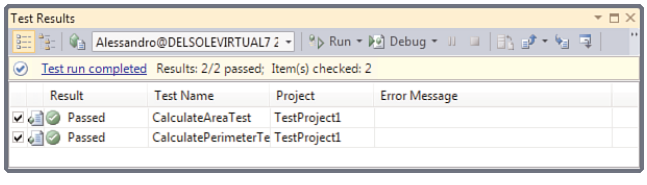

FIGURE 59.4 Both unit tests passed.

Enabling Code Coverage

Visual Studio enables getting information on the amount of code that was subject to the actual test. This can be accomplished by enabling a feature known as Code coverage. To enable it, follow these steps:

1. Select the **Test**, **Edit Test Settings**, **Local** command. This enables editing the current test configuration;
2. When the Test Settings dialog appears, select **Data and Diagnostics**; then flag the **Code Coverage** option in the bottom-right part, as shown in Figure 59.5;

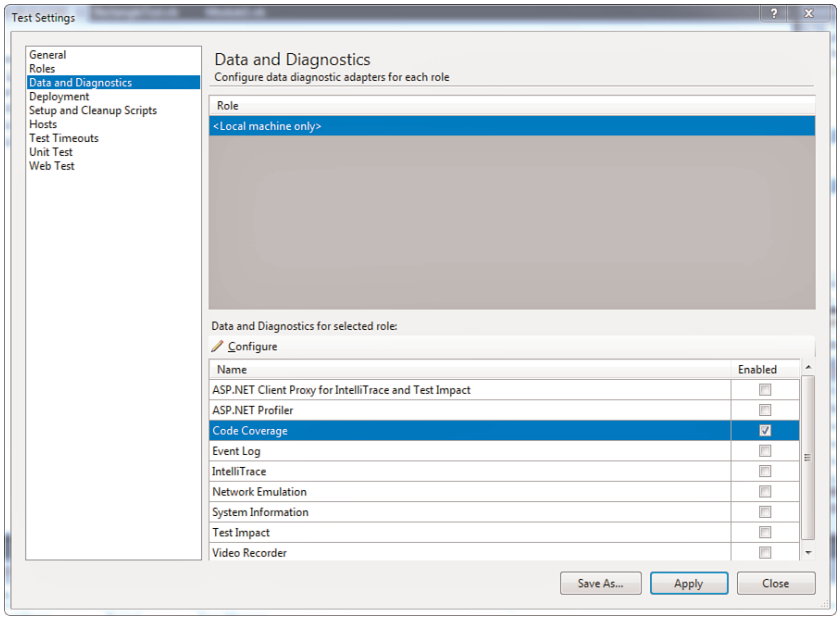

FIGURE 59.5 Enabling code coverage.

3. Click **Configure**. When the Code Coverage **D**etails dialog appears, select the test assembly, which in this case is **TestProject1.dll**.

Now rerun all unit tests. When completed, the Code Coverage Results dialog shows information on collected results. Figure 59.6 shows the results.

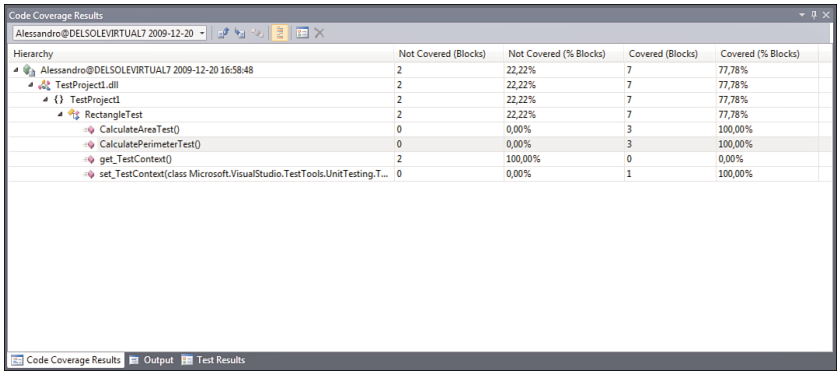

FIGURE 59.6 Examining code coverage results.

You can expand the results to get information on code coverage percentage for single members. For example, both test methods have a 100% coverage percentage against a total 77.78% for the entire project. If you switch to the test code file you notice that Visual Studio automatically highlights lines of code that were subject to test.

Unit Tests and IntelliTrace

In Chapter 58, "Advanced Analysis Tools," you discovered IntelliTrace, and you learned how the historical debugger can be helpful in debugging code due to its in-depth analysis features. IntelliTrace can also be used for unit test failures to get even more detailed information on occurred exceptions. For this, you first need to enable IntelliTrace in the testing environment, so select the **Test, Edit Test Settings, Local** command. When the settings dialog appears, select IntelliTrace, as shown in Figure 59.7.

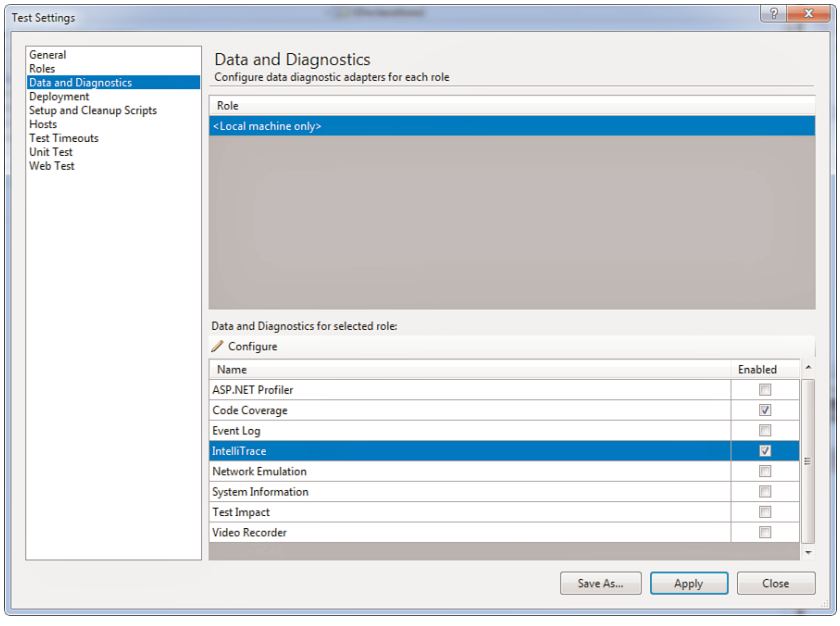

FIGURE 59.7 Enabling IntelliTrace for unit tests.

Now provide the following bad value in the `CalculateAreaTest` method so that unit test will fail:

```
Dim expected As Double = 2000
```

Run the unit test again, at this point you get a test failure. Simply double-click the error message, and you get detailed information on the exception, as shown in Figure 59.8.

Now click on the log file hyperlink shown below the Collected Files item. This launches the IntelliTrace log analysis tool (as shown in Figure 59.9), where you can analyze collected information according to what you already learned about this in Chapter 58.

Notice how you can get detailed information on the stack trace related to the failed assertion in the test code.

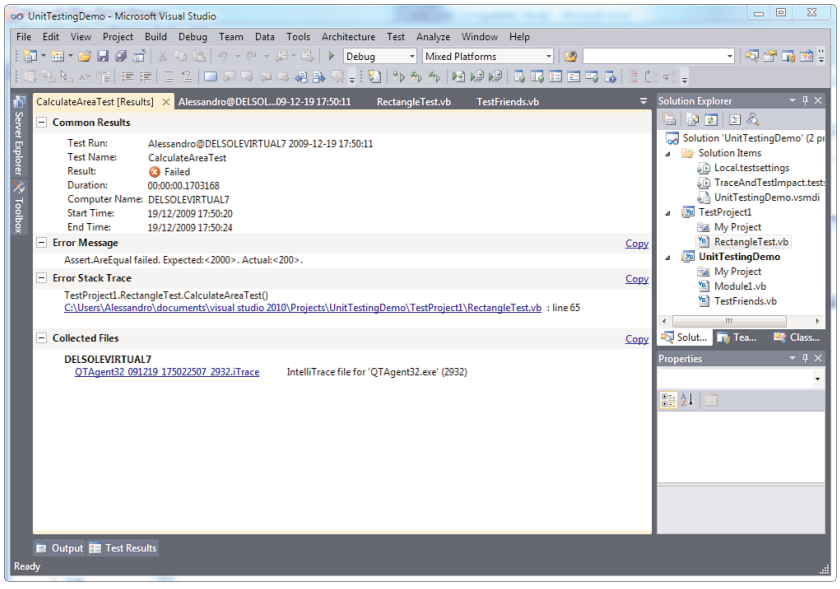

FIGURE 59.8 Detailed information provided by IntelliTrace about the failed unit test.

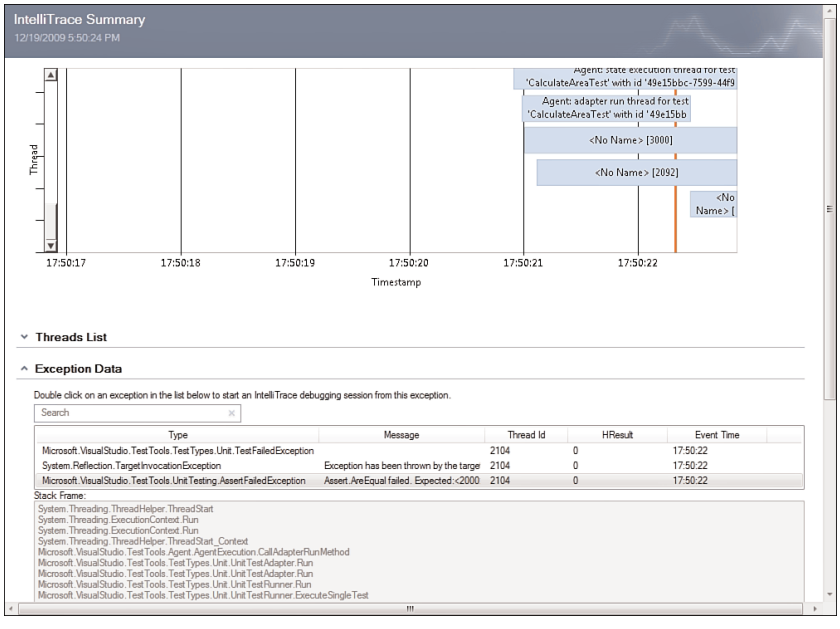

FIGURE 59.9 Analyzing IntelliTrace log for unit test failures.

Introducing Test-Driven Development

Test-Driven Development (also known as TDD or Test-Driven Design) is a programming approach in which developers create applications by first writing unit tests and then writing the actual code after the unit test passes. This particular approach helps writing better code, because you ensure that it will work via unit tests, but it is also a life philosophy so that you need to have a change of mind when approaching TDD. Basically TDD is structured into three main moments:

- **Red:** The developer generates a new unit test from scratch, so that it will typically fail. This is the reason why it's called Red.
- **Green:** The developer focuses on writing code that makes the unit test work and pass. As you saw in the previous section, passing unit tests return a green result.
- **Refactor:** This is the moment in which the developer reorganizes code, moving it from the unit test to the actual code system in the application project, making the code clearer and fixing it where necessary.

This chapter is not intended to be a deep discussion on TDD, whereas it is instead intended to be a guide to Visual Studio 2010 instrumentation for Test-Driven Development. Particularly you see how the Generate from **U**sage new feature discussed in Chapter 18, "'Generate from Usage' Coding Techniques," is the main help you have in TDD with Visual Basic 2010. Before going on, it is a good idea to enable test options that enable double-clicking a test result failure in the Test **R**esults dialog to be redirected to the code that threw errors. Follow these steps:

1. Go to **Tools**, **Options** and select the **Test Tools**, **Test Execution** subfolder.
2. Enable the **Double Clicking Failed or Inconclusive Unit Test Result Displays the Point of Failure in Test** item. Figure 59.10 shows how to accomplish this.
3. Click **OK** to close the dialog.

At this point you can create a test project where you can launch your unit tests.

Creating a Test Project

The first step in the Test-Driven Development approach is creating a Test Project related to the actual application project. Follow these steps:

1. Create a new class library named Rectangle and remove the Class1.vb default file.
2. Add a new test project to the solution. To accomplish this, select **File**, **Add**, **New Project**, and then in the Test Projects folder, select the **Test Documents** sub node, finally select the **Test Project** template. Figure 59.11 shows how to find and select the template in the New Project dialog.
3. Add a new class to the test project and name it **RectangleTest.vb**. This is basically the place where you write unit tests.

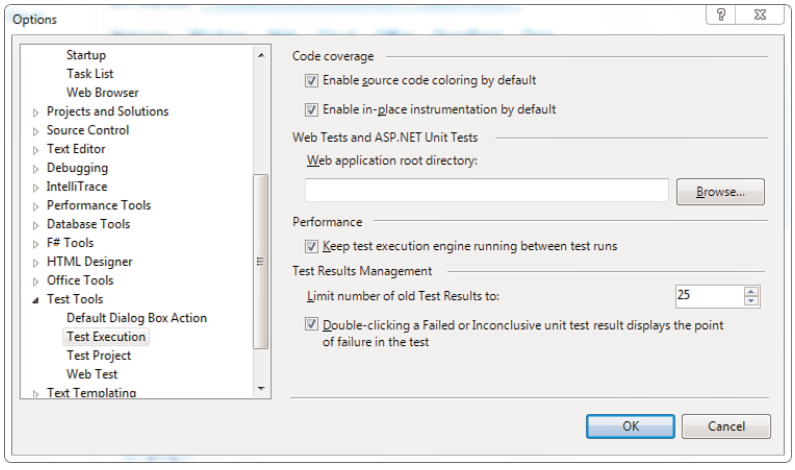

FIGURE 59.10 Setting test execution options.

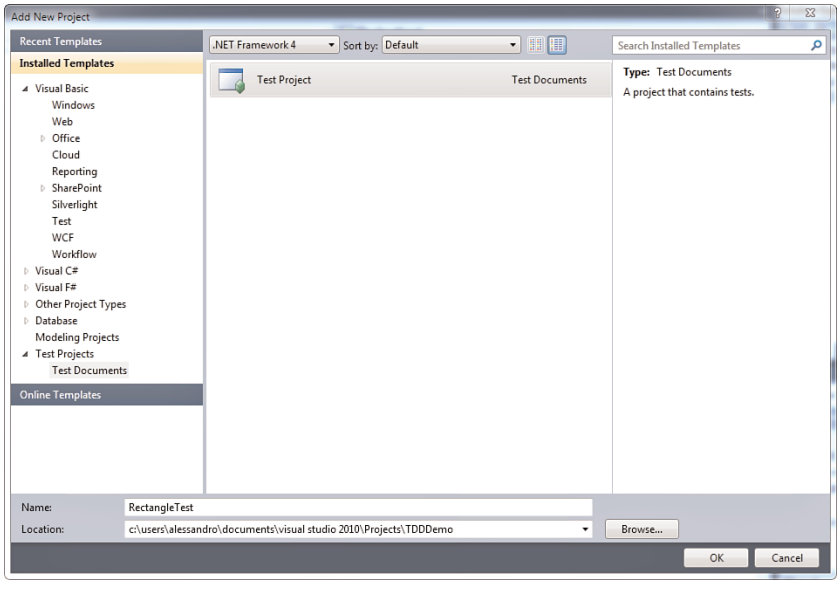

FIGURE 59.11 Creating a new test project.

The new class is the place where you write and run unit tests. Now imagine you want to test a `Rectangle` class exposing `Width` and `Height` properties and methods for math calculations such as the perimeter. The class will be exposed by the actual project at the end of the TDD approach, at the moment you need to test class and methods in the test project. To accomplish this, the first thing is marking the test class with the `Microsoft.VisualStudio.TestTools.UnitTesting.TestClass` attribute. Fortunately the

namespace is automatically imported by Visual Basic when you generate a test project, so you do not need to write this import manually. This is therefore how the new class looks:

```
<TestClass()>
Public Class RectangleTest
End Class
```

Basically the `TestClass` attribute makes a class recognizable by Visual Studio as a place for unit tests, which the next subsection covers.

Creating Unit Tests

As you can recap from the "Testing Code with Unit Testing" section, unit tests are methods allowing tests against small, isolated portions of code. To be recognized as unit tests, such methods must be decorated with the `Microsoft.VisualStudio.TestTools.UnitTesting.TestMethod` attribute. Continuing our example and having the requirement of implementing a method for calculating the perimeter for rectangles, this is how the method stub appears:

```
<TestMethod()>
Sub CalculatePerimeter()
End Sub
```

Now go into the method body and write the following line:

```
Dim rect As New Rectangle
```

The `Rectangle` type is not defined yet, so Visual Studio underlines the declaration by throwing an error. Click on the error options pop-up button and click Generate New Type, as shown in Figure 59.12.

This launches the Generate New Type dialog, where you can select the Rectangle project in the Location combo box. See Figure 59.13 for details.

WHY GENERATE A TYPE IN THE PRODUCTION PROJECT?

You might ask why the preceding example showed how to add the new type to the actual project instead of the test one. In a demo scenario like this, adding a new type to the test project would be useful, but in real life you might have dozens of new types to add and then moving all types from a project to another, including code edits, can be less productive. The illustrated approach keeps the benefits of TDD offering a way of implementing types directly in the project that will actually use them.

At this point you will need to replace the following declaration:

```
Dim rect As New Rectangle
```

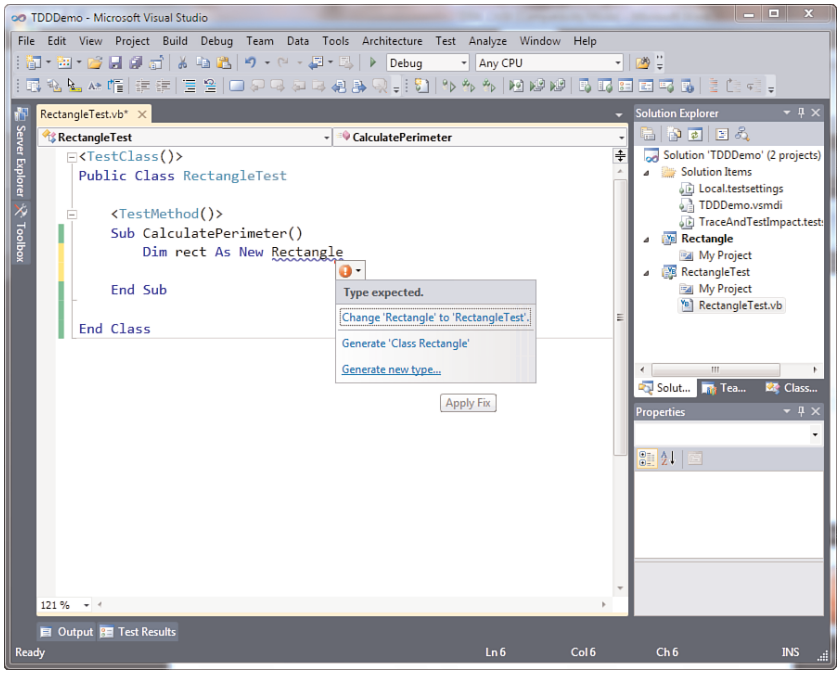

FIGURE 59.12 Choosing a correction option.

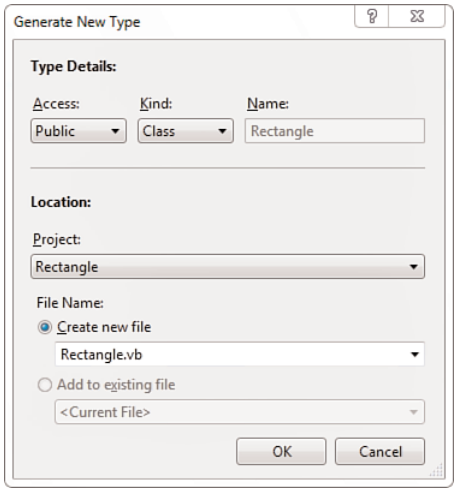

FIGURE 59.13 Generating a new `Rectangle` type.

with this one, including the namespace:

```
Dim rect As New Rectangle.Rectangle
```

Now type the following line of code:

```
rect.Width = 150
```

The `Width` property is not exposed yet by the `Rectangle` class, so Visual Studio underlines it as an error. As for the class generation, click the error correction options and select the **Generate Property Stub for 'Width'** choice, as shown in Figure 59.14.

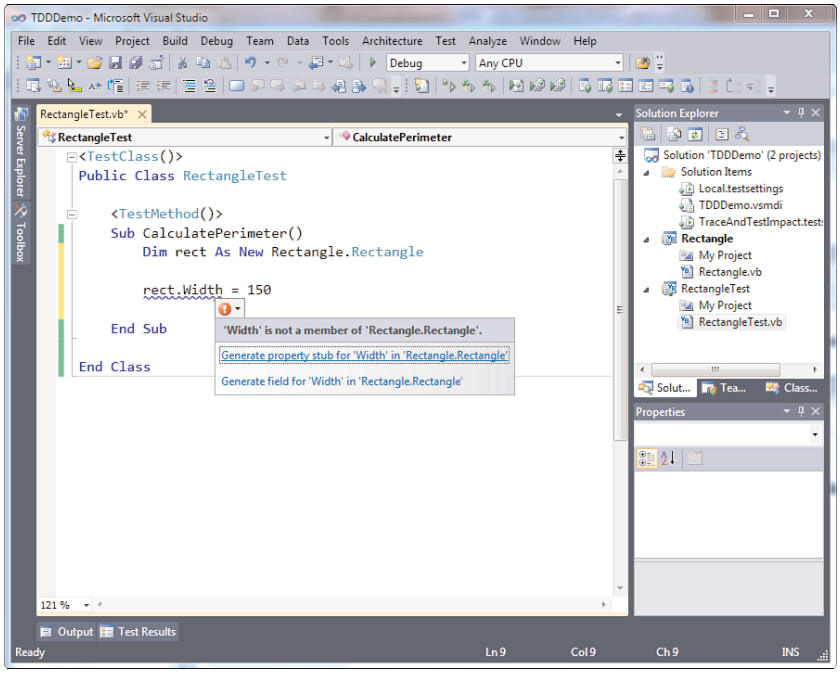

FIGURE 59.14 Generating a property stub.

This adds a property to the `Rectangle` class. Now write the following line of code and repeat the steps previously shown:

```
rect.Height = 100
```

Now complete the method body by writing the following lines:

```
Dim expected = 500
```

```
Dim result = rect.CalculatePerimeter
Assert.AreEqual(expected, result)
```

Basically you have an expected result (notice I'm using type inference) and an actual result returned by the `CalculatePerimeter` method. This method does not exist yet, so use the **Generate from Usage Feature** to add a new method stub to the `Rectangle` class. Now run the unit test and it will fail as expected, being in the Red moment of TDD, as demonstrated in Figure 59.15.

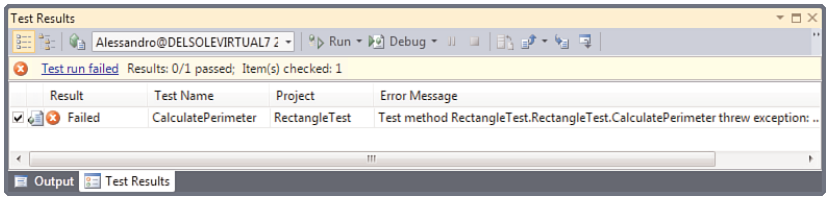

FIGURE 59.15 Running the new unit test fails due to an exception.

Basically the unit test fails because the current method definition for `CalculatePerimeter` is the following:

```
Function CalculatePerimeter() As Object
    Throw New NotImplementedException
End Function
```

So edit the method as follows, to make it return a more appropriate type and perform the required calculation:

```
Function CalculatePerimeter() As Integer
    Return (Width * 2) + (Height * 2)
End Function
```

Now run again the unit test and it will pass. You have thus successfully completed the **Green** phase of TDD, and now you can now move to the final Refactor step.

Refactoring Code

When your unit tests all pass, it is time to reorganize code. For example, if you take a look at the `Rectangle` class, you notice that the **Generate from Usage Feature** generated objects of type `Integer`, and this is also the reason why the `CalculatePerimeter` method has been forced to return `Integer`. Although correct, the most appropriate type for math calculations is `Double`. Moreover, you might want to consider writing a more readable code in the method body. After these considerations, the `Rectangle` class could be reorganized as follows:

```
Public Class Rectangle
```

```
    Public Property Width As Double
    Public Property Height As Double

    Public Function CalculatePerimeter() As Double

        Dim sumOfWidth As Double = Me.Width * 2
        Dim sumOfHeight As Double = Me.Height * 2
        Dim perimeter As Double = sumOfHeight + sumOfWidth

        Return perimeter
    End Function
End Class
```

In this way you have working code that uses more appropriate types and that is more readable.

Understanding Code Contracts

Code Contracts is a new library in the .NET Framework 4.0 offered by the System.Diagnostics.Contract namespace and enables checking, both at runtime and compile time, if the code is respecting specified requirements. This is something that you will often hear about as Contracts by design. The idea is that code needs to respect specified contracts to be considered valid. There are different kinds of contracts, known as preconditions (what the application expects), post-conditions (what the application needs to guarantee), and object invariants (what the application needs to maintain). We cover all of them in next subsections. At the moment it is important to understand some other concepts. Code contracts are a useful way for testing code behavior and this can be accomplished both at runtime (*runtime checking*) and compile time (*static checking*). Runtime checking needs the code to be executed and is useful when you cannot predict some code values; for example the application needs the user to enter some values that will be then validated; at this point you can take advantage of contracts for implementing validation rules. Static checking can be useful if you have hard-coded variable values and you need to check if they are contract-compliant. Both checking methods can be used together and can be set in the Visual Studio IDE as explained in the next sections.

Setting Up the Environment

The Code Contracts library is part of the .NET Framework 4.0, so you do not need to install anything more to use it in code; there are some components that you need to install separately to access contracts settings in the Visual Studio IDE. So, before going on reading this chapter, go to the following address: http://msdn.microsoft.com/en-us/devlabs/dd491992.

aspx. From the DevLab site download the Code Contracts tools in the most appropriate version for you. (For example the VSTS Edition is intended to work with Visual Studio 2010 Ultimate.) When installed, you can access design-time settings for code contracts.

Setting Contracts Properties

You set code contracts properties by first opening My Project and then clicking the **Code Contracts** tab, as shown in Figure 59.16.

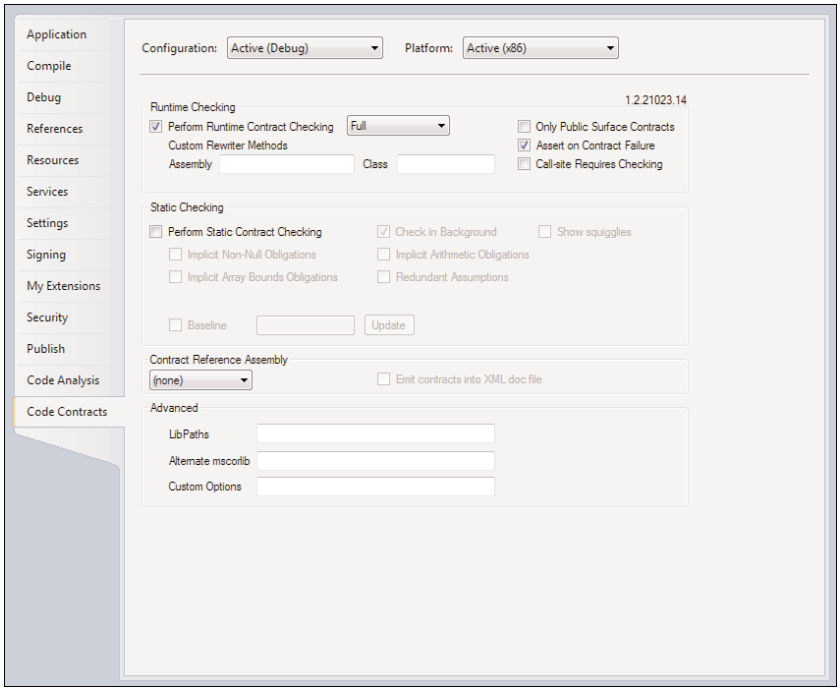

FIGURE 59.16 Accessing code contracts settings.

Notice how you can enable general settings, such as runtime checking and static checking, and specific settings for both profiles. Particularly, for runtime checking you should leave the default Full selection if you have both preconditions and post-conditions. If you want to enable static checking, too, for compile time contracts checking, by default the Check in Background option is also selected. This enables the background compiler to check for contracts violations and send error messages to the Errors tool window. If this option is unselected, eventual error messages will be listed at the end of the build process.

TIP

If you remove the flag on Assert on Contract Failure, instead of an error dialog showing details about the violation, the control will be returned to the Visual Studio code editor that will break at the line of code that violated the contract.

The next examples show both preconditions and post-conditions, so leave unchanged the default settings. Before getting hands on the code, you should read a little about tools that enable Visual Studio to integrate and work with contracts.

Tools for Code Contracts

When you use code contracts, the first requirement is the `System.Diagnostics.Contracts` namespace, exposed by the Mscorlib.dll assembly. By the way, this is not enough to make your code take advantage of contracts. Although you never see this, Visual Studio invokes behind the scenes some command-line tools. This subsection provides basic information on these tools and on their purpose.

The Binary Rewriter

As you know, when you compile a .NET executable, the file is made of metadata and Intermediate Language. When you use contracts, especially for runtime checking, the Intermediate Language within an executable needs to be modified to recognize contracts. The edits are performed by the CCrewrite.exe tool that injects the appropriate code for contracts in the appropriate place into your assembly.

The Static Checker

The static checker is represented by the CCCheck.exe tool and provides Visual Studio the capability of performing static analyses and checks for contracts violations without the need of executing code; a typical scenario is the compilation process.

Now that you have basic knowledge of the code contracts system, it is time to write code and understand how contracts work.

Preconditions

You add preconditions contracts to tell the compiler that the code can be executed only if it respects the specified contract. Generally preconditions are useful replacements for custom parameters validation rules. For example, consider the following simple method that multiplies two numbers:

```
Function Multiply(ByVal first As Double,
                  ByVal second As Double) As Double

    If first < 0 Or second < 0 Then
        Throw New ArgumentNullException
    Else
        Return first * second
    End If
End Function
```

Inside the method body, the code checks for valid parameters; otherwise, it throws an exception. This is common, but code contracts provide a good way, too. The preceding method could be rewritten with code contracts as follows:

```
Function Multiply(ByVal first As Double,
```

```
            ByVal second As Double) As Double

    Contract.Requires(first > 0)
    Contract.Requires(second > 0)
    Return first * second
End Function
```

So you just invoke the `Contract.Requires` method for evaluating a Boolean condition that will be accepted only when evaluated as `True`. Now consider the following `Rectangle` class:

```
Class Rectangle


    Property Width As Double
    Property Height As Double

    Function CalculatePerimeter() As Double
        Dim result = (Width * 2) + (Height * 2)

        Return result
    End Function

    Sub New(ByVal width As Double, ByVal height As Double)
        Me.Width = width
        Me.Height = height
    End Sub
End Class
```

The `CalculatePerimeter` instance method takes no arguments and performs calculations on instance properties but does not check for valid values. With regard to this you can take advantage of the `Contract.Requires` method that specifies a condition allowing the code to be considered valid if the condition is evaluated as `True`. For example, consider the following reimplementation of the method:

```
Function CalculatePerimeter() As Double
    Contract.Requires(Me.Height > 0)
    Contract.Requires(Of ArgumentOutOfRangeException) _
             (Me.Width > 0)

    Dim result = (Width * 2) + (Height * 2)
    Return result
End Function
```

In this case the contract requires that the `Height` property is greater than zero; otherwise a runtime error is thrown. You instead use `Contract.Requires(Of T)` when you want to throw a specific exception when the contract is violated. The above example throws an

`ArgumentOutOfRangeException` if the `Width` property is less than zero. For example consider the following code that creates an instance of `Rectangle` but violates the contract:

```
Dim r As New Rectangle(0, 80)
Console.WriteLine(r.CalculatePerimeter)
```

When you run this code, the runtime throws an `ArgumentOutOfRageException`, as shown in Figure 59.17, due to an invalid `Width` value.

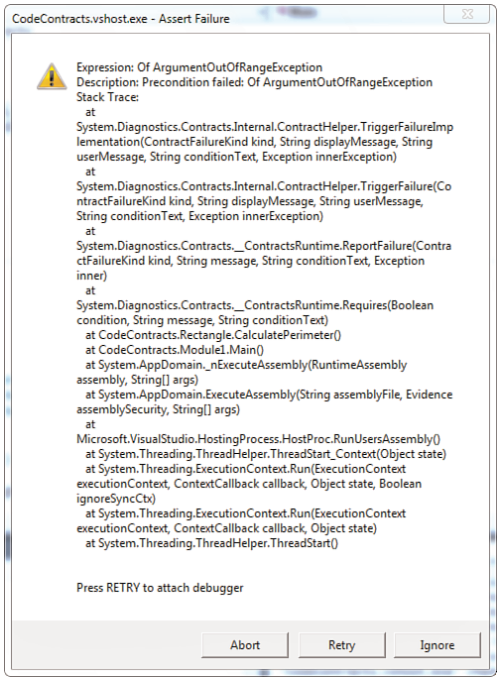

FIGURE 59.17 The exception thrown when the code violates a precondition.

Preconditions are thus useful when you want to validate code elements before they are invoked or executed. The next section discusses post-conditions instead.

Post-Conditions

A post-condition is a contract that is checked after code is executed and is basically used to check the result of some code execution. Continuing with the previous example, you might want to check that the `CalculatePerimeter` method produces a value greater than

zero before returning the result. This kind of post-condition is accomplished via the `Contracts.Ensures` method, as demonstrated in the following snippet:

```
Function CalculatePerimeter() As Double
    Contract.Ensures(Contract.Result(Of Double)() > 0)
    Dim result = (Width * 2) + (Height * 2)

    Return result
End Function
```

Also notice how `Ensures` invokes `Contract.Result(Of T)`. This is basically the representation of the code result, and `T` is nothing but the expected type, which in this case is `Double`. This line of code must be placed before the code is executed and the compiler can link the actual result with the contract evaluation.

EXCEPTIONAL POST-CONDITIONS

The `Contract` class also provides an `EnsuresOnThrow(Of TException)` method that checks for the condition only when the specified exception is thrown. Generally this approach is discouraged, and you should use it only when you have complete understanding of what kind of exceptions your method could encounter.

Old Values

You can refer to values as they existed at the beginning of a method by using the `Contract.OldValue(Of T)` method. For example, the following code ensures that a hypothetical value variable has been updated:

```
Contract.Ensures(value) = Contract.OldValue(value) + 1)
```

Invariants

Invariants are special contracts that ensure an object is considered valid during all its lifetime. Invariant contracts are provided inside one method decorated with the `ContractInvariantMethod` attribute that affects all members in the enclosing class. Only one invariant method can be declared inside a class, and typically it should be marked as `Protected` to avoid risk of calls from clients. The method is by convention named `ObjectInvariant` (although not mandatory) and is used instead of preconditions and post-conditions. The following code snippet provides an example:

```
<ContractInvariantMethod()>
Protected Sub ObjectInvariant()
    Contract.Invariant(Me.Width > 0)
    Contract.Invariant(Me.Height > 0)
End Sub
```

Simply this code establishes that during the entire lifetime of the `Rectangle` object, both `Width` and `Height` properties must be greater than zero so that they can be considered in a valid state.

Assertions and Assumptions

The `Contract` class provides an `Assert` method that is used for verifying a condition at a particular point in the program execution. Typically you use it as follows:

```
Contract.Assert(Width > 0)
```

There is also another method named `Assume`, which works exactly like `Assert` but is used when static verification is not sufficient to prove the condition you are attempting to check.

Contract Events

The `Contract` class exposes a `ContractFailed` event that is raised when a condition is violated and that you can handle to get detailed information. The following sample event handler shows how to collect violation information:

```
Private Sub Contract_ContractFailed(ByVal sender As Object,
                                    ByVal e As ContractFailedEventArgs)
    Console.WriteLine("The following contract failed: {0}, {1}",
                      e.Condition, e.FailureKind.ToString)
End Sub
```

The `ContractFailedEventArgs.Condition` property is a string storing the condition while the `ContractFailedEventArgs.FailureKind` is an enumeration offering the failure kind (for example, `Precondition`, `Invariant`, and so on).

Summary

This chapter illustrated the Visual Studio instrumentation and libraries about testing applications. The first discussion was about unit testing, a technique used for checking if code blocks work outside the application context, inside a sandbox. For this you saw how test projects work and how to enable code coverage. The last topic in the unit test discussion was about IntelliTrace, useful for unit tests debugging. The second discussion of the chapter was about Test-Driven Development, a programming approach in which developers write their software by starting by writing unit tests. Particularly you saw Visual Studio 2010's support for TDD, with special regard of the Generate from **U**sage feature. The final part of this chapter was dedicated to Code Contracts, a new library in the .NET Framework 4 that enables writing code in the *contract-by-design* fashion and that is supported by the `System.Diagnostics.Contracts` namespace and the `Contract` class.