

CHAPTER 58

Advanced Analysis Tools

Writing code is just one part of the developer life. There are so many other aspects to consider in producing high-quality applications. For example, if you produce reusable class libraries, you must ensure that your code is compliant with Common Language Specification, and this requires deep analysis. Another key aspect is performance. If you produce a great application with the most desired functionalities but with poor performance, perhaps your customer will prefer a faster and less-consuming application even if it has a minor number of features. Continuing from its predecessors, Visual Studio 2010 offers a number of integrated tools for analyzing code and performance to improve your applications' quality. It is worth mentioning that most of the previously existing tools have been significantly enhanced due to the WPF-based architecture of the IDE. In this chapter you learn how to take advantage of Visual Studio 2010's integrated analysis tools for writing better applications.

VISUAL STUDIO SUPPORTED EDITIONS

Analysis tools are available only in some Visual Studio 2010 editions. To complete tasks explained in this chapter, you need at least Visual Studio 2010 Premium or the Visual Studio 2010 Ultimate that is required for IntelliTrace.

IN THIS CHAPTER

- ▶ Introducing Analysis Tools
- ▶ Performing Code Analysis
- ▶ Calculating Code Metrics
- ▶ Profiling Applications
- ▶ IntelliTrace, the Historical Debugger
- ▶ Generating Dependency Graphs

Introducing Analysis Tools

Visual Studio 2010 offers the following analysis tools, which can help you produce high-quality applications:

- ▶ Code Analysis, which analyzes code for compliance with Microsoft coding rules
- ▶ Code Metrics, which returns statistic results and analyzes code maintainability according to specific indexes;
- ▶ Profiler, which analyzes application performance and suggests solutions for solving problems
- ▶ IntelliTrace, formerly known as Historical Debugger, which allows keeping track of every single event and exceptions happening during the entire application lifetime

In this chapter you learn to take advantage of the listed tools for improving quality in your code.

Performing Code Analysis

Earlier in this book you learned about Common Language Specification, learning that it is a set of common rules and guidelines about writing code that can be shared across different .NET languages favoring interoperability and that is considered well designed for the .NET Framework. In some cases it can be hard ensuring that all your code is CLS-compliant, especially when you have large projects with tons of lines of code. To help you write better and CLS-compliant code, Microsoft produced a code analysis tool named FxCop that analyzes compiled assemblies for non-CLS-compliant code and that reports suggestions and solutions for solving errors. Although free, FxCop is an external tool and is bound to developers using Visual Studio editions such as Express or Professional.

DOWNLOADING FXCOP

If you do not have Visual Studio Ultimate but want to try the code analysis features, you can check out FxCop, which is available on the MSDN Code Gallery at <http://code.msdn.microsoft.com/codeanalysis>. Generally all concepts described in this section are available in FxCop, too.

Fortunately, the Ultimate edition offers an integrated version of the code analysis tool that you can invoke on your project or solution directly within the IDE; moreover you can customize the code analysis process by setting specific rules. Before examining available rules, it is a good idea to create a simple project for demonstrating how code analysis works, so create a new class library project in Visual Basic 2010. When the code editor is ready, rename the Class1 file to **HelperClass** and write the code shown in Listing 58.1, which attempts defining a CLS-compliant class but that makes several violations to the Microsoft rules, for which I give an explanation for solving later in this section.

LISTING 58.1 Writing a Simple non-CLS-Compliant Class Library

```

<CLSCompliant(True)>
Public Class HelperClass

    Private CustomField As String

    Public Property customResult As String

    'Just a demo function
    Public Function doubleSum(ByVal FirstValue As Double,
                             ByVal SecondValue As Double) As Double
        Return FirstValue + SecondValue * 2
    End Function
End Class

```

For rules, it is worth mentioning that Microsoft divides guidelines in writing code into the rules summarized in Table 58.1.

TABLE 58.1 Microsoft Code Analysis Rules

Rule name	Description
Microsoft.Design	Determines if assemblies contain well-designed objects or if the assembly definition is CLS-compliant
Microsoft.Globalization	Determines if globalization techniques are well implemented
Microsoft.Interoperability	Determines if the code makes correct usage of COM interoperability
Microsoft.Maintainability	Checks for code maintainability according to Microsoft rules
Microsoft.Mobility	Checks for timer and processes correct implementation
Microsoft.Naming	Determines if all identifiers match the CLS rules (such as public/private members, method parameters and so on)
Microsoft.Performance	Checks for unused or inappropriate code for compile time and runtime performances from the CLR perspective
Microsoft.Portability	Determines if the code is portable for invoked API functions
Microsoft.Reliability	Provides rules for a better interaction with the Garbage Collector
Microsoft.Security	Provides security-related rules sending error messages if types and members are not considered secure
Microsoft.Usage	Determines if a code block correctly invokes other code

Performing code analysis does not require all the mentioned rules to be checked. You can specify only a subset of preferred rules or specify the complete set. To specify the rules sets involved in the code analysis, follow these steps:

1. Open **My Project** and click on the **Code Analysis** tab. Figure 58.1 shows how the designer looks

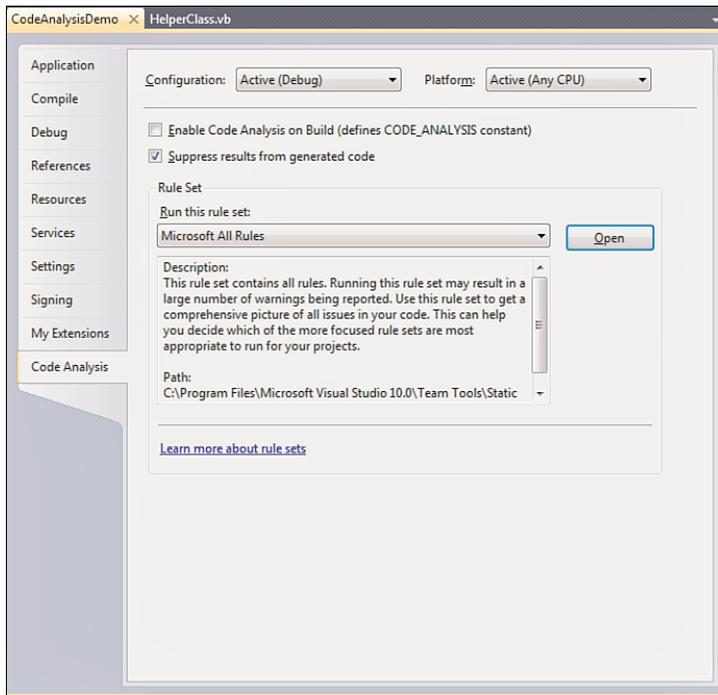


FIGURE 58.1 The Code Analysis designer.

2. Expand the **Run This Rule Set** combo box. You get a list of available rule sets with an accurate description for each set. By default, the offered set of rules is **Microsoft Minimum Recommended Rules**. Replace it by selecting **Microsoft All Rules** that includes all sets listed in Table 58.1 and that is the most accurate. To get detailed information on each rule set, simply click **Open**. Figure 58.2 shows how you can browse rules available in the selected set, getting summary information for each rule, and specifying how you want to get help on violations (for example, online or offline)
3. Select the **Analyze, Run Code Analysis** command and wait for a few seconds until the building and code analysis process is completed. When ready, Visual Studio shows a report listing all violations to coding rules encountered in the project. The report is shown in Figure 58.3.

Each violation message includes the violation ID and a description that can help you fix the error. In most cases violations are interpreted by the IDE as warnings, but in the code

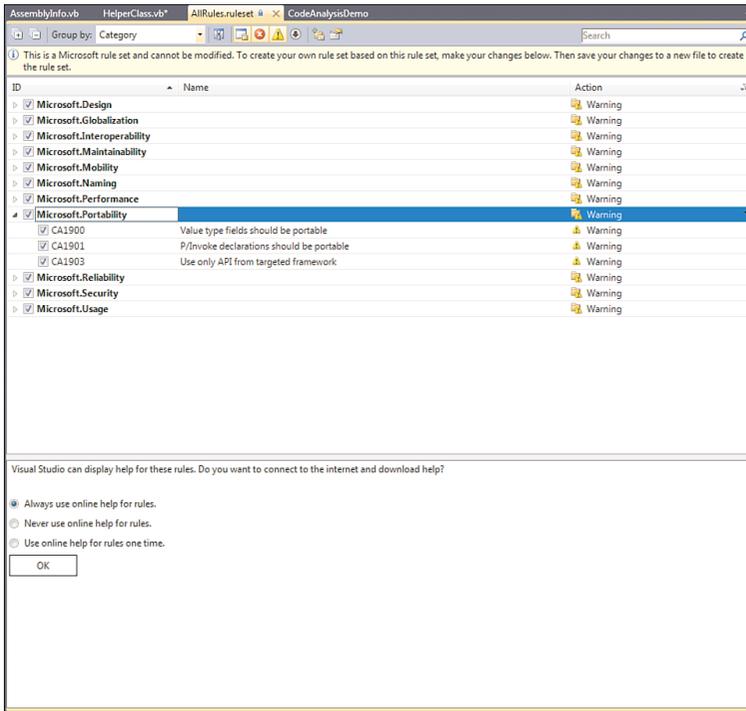


FIGURE 58.2 Browsing rule sets.

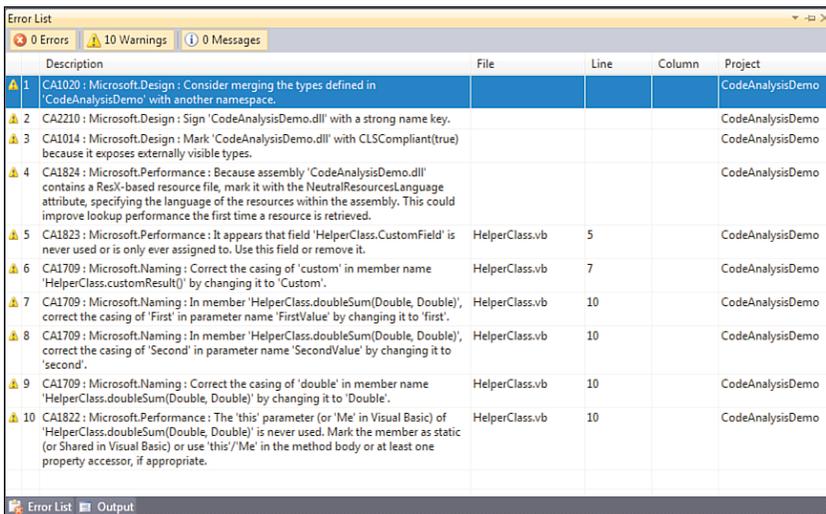


FIGURE 58.3 Report from the code analysis tool.

analysis designer you can modify this behavior by setting what violations must be notified as breaking errors. Also notice how assembly-level violations do not include the line of code to be fixed, whereas code-level violations do. In this case you simply double-click the error message to be immediately redirected to the line of code to be fixed.

GETTING ERROR HELP

There are hundreds of Microsoft rules, so summarizing all of them in this book is not possible. You can get detailed information on each violation (and on how you solve the violation for each) by simply right-clicking the error message and selecting **Show Error Help** from the pop-up menu.

At this point we can begin fixing violations. It is worth mentioning that there can be situations in which violations cannot be fixed due to particular scenarios. For example, the first violation in our example (CA1020) indicates that we should merge the current type into an existing namespace, because a well-formed namespace contains at least five members. Due to our demo scenario, we can ignore this violation that is nonbreaking. The next error message (CA2210) indicates that the assembly must be signed with a strong name. I described strong names in Chapter 53, "Understanding the Global Assembly Cache," so follow those instructions to add a strong name file to the library. I named the strong name file as `TestCode.pfx` providing the `TestCode` password. The next violation (CA1014) requires the assembly to be marked as CLS-compliant. To accomplish this, click the **Show All Files** button in Solution Explorer, expand **My Project** and add the following line to the `AssemblyInfo.vb` file:

```
<Assembly: CLSCompliant(True)>
```

The next violation (CA1824) indicates that a neutral-language resource should be supplied to the assembly. Because you have the `AssemblyInfo.vb` file already open, write the following line:

```
<Assembly: NeutralResourcesLanguageAttribute("en-US")>
```

You could also set this property via the Assembly Information dialog from the Application tab in My Project. With this step, all assembly level violations were fixed. Now it's time to solve code-level violations. The CA1823 violation suggests that there is a field named `CustomField` that is never used and that, because of this, should be removed to improve performance. Now, remove the following line of code:

```
Private CustomField As String
```

The next step is to solve three CA1709 violations that are all about members naming. We need to replace the first letter of the `doubleSum` method with the uppercase and the first letter of both arguments with lowercase. This is translated in code as follows:

```
Public Function DoubleSum(ByVal firstValue As Double,
                          ByVal secondValue As Double) As Double
```

NAMING CONVENTIONS

We discussed naming conventions in Chapter 7, “Class Fundamentals,” with regard to methods, method arguments, and properties, so refer to that chapter for details.

There is also another naming convention violation to be fixed on the `customResult` property that must be replaced with `CustomResult`. The last required fix is on performance (CA1822 violation). The code analysis tool determines that the `DoubleSum` method never invokes the class constructor; therefore, it suggests to mark the method as `Shared` or to invoke the constructor. In this particular situation the method can be marked as `Shared` like this:

```
Public Shared Function DoubleSum(ByVal firstValue As Double,
                                ByVal secondValue As Double) As Double
```

In this particular case we do not need an instance method. For your convenience all edits are available in Listing 58.2 (except for assembly level edits).

LISTING 58.2 Fixing Errors Reported by the Code Analysis Tool

```
<CLSCompliant(True)>
Public Class HelperClass

    'Private CustomField As String

    Public Property CustomResult As String

    Public Shared Function Sum(ByVal firstValue As Double,
                              ByVal secondValue As Double) As Double
        Return firstValue + secondValue
    End Function
End Class
```

If you now run the code analysis tool again, you notice that only the CA1020 design rule is still reported (the one on merging types into an existing namespace that we decided not to fix due to our particular scenario). The code analysis tool is a helpful instrument, especially if you are a libraries developer. Microsoft has a Code Analysis Team with a blog where you can find interesting information: <http://blogs.msdn.com/fxcop>. Also remember that you can easily retrieve detailed information on violation errors and fixes directly from within Visual Studio.

Calculating Code Metrics

Code metrics is an interesting tool that analyzes a project or a solution providing results about the ease of maintainability according to specific indexes. You can invoke the tool from the Analyze menu by selecting the **Calculate Code Metrics** command or by right-clicking the project name in Solution Explorer and then selecting the same-named command. The tool calculates code metrics according to the indexes summarized in Table 58.2.

TABLE 58.2 Code Metrics Analyzed Indexes

Index	Description
Maintainability index	A percentage value indicating ease of maintainability for the selected project or solution. A higher value indicates that the project is well structured and easily maintainable.
Cyclomatic complexity	A percentage value indicating complexity of loops, nested loops, and nested conditional blocks, such as nested For . . Next loops, Do . . Loop loops, or If . . End If nested blocks. A higher value indicates that you should consider refactoring your code to decrease loop complexity because this leads to difficult maintainability.
Depth of inheritance	Indicates the inheritance level for classes in the project. The result shows the report for the class with the highest inheritance level. A higher value indicates that it might be difficult finding problems in a complex inheritance hierarchy.
Class coupling	Calculates how many references to classes there are from method parameters and return values, local variables, and other implementations. A higher value indicates that code is difficult to reuse, and you should consider revisiting your code for better maintainability.
Lines of code	Just a statistic value. It returns the number of IL code affected by the analysis.

To understand how it works, in Visual Studio 2010 open the **SnippetCompilerVSPackage** sample project described in the previous chapter; then run the Code Metrics tool by selecting the **Analyze, Calculate Code Metrics for Solution** command. After a few seconds you get the report shown in Figure 58.4.

As you can see from the report, the project has a maintainability index of 88 that is quite good. Generally values from 80 to 100 are the best range for maintainability; Visual Studio shows a green symbol if the index is good or a red one if the maintainability index is too poor. You can expand the **SnippetCompilerVsPackage** item to see how the global result is subdivided for each class and also for each class member. The global Cyclomatic Complexity index is 11, which is a small number for our kind of project. Depth of Inheritance index is 3, which is a small value, meaning that there is one or more class inheriting from another class that inherits from another one (the third one is

Hierarchy	Maintainability I...	Cyclomatic Com...	Depth of Inherita...	Class Coupling	Lines of Code
SnippetCompilerVSPackage (Debug)	88	11	3	28	25
AlessandroDeSole.SnippetCompilerVSPackage	88	11	3	28	25
GuidList	96	2	1	1	2
MyToolWindow	83	3	3	6	8
Content.Get() As Object	86	1		1	2
Content.Set(Object)	94	1		2	1
New()	73	1		3	5
PlgCmdList	100	1	1	0	1
SnippetCompilerVSPackagePackage	71	5	2	24	14

FIGURE 58.4 Calculating code metrics for the specified project.

System.Object); this is an absolutely acceptable value in this particular scenario. The Class Coupling index is a little too high. It is determined by the SnippetCompilerVsPackagePackage class, meaning that this class has a lot of references to other classes. Particularly, if you expand the class you notice that the problem is the Initialize method that makes calls to a lot of objects. Obviously, a high index doesn't necessarily indicate problems. In this code example a high value is acceptable, because all invocations are required to make the Visual Studio package work, but in a reusable class library a high value needs attention and code refactoring.

EXPORTING TO EXCEL

If you need to elaborate the code metrics results, you can export the analysis report to Microsoft Excel. This can be accomplished with the Open List in Excel button on the Code Metrics Result tool window.

Profiling Applications

Performance is a fundamental aspect in development. An application with slow performance can discourage customers, even if it has the coolest user interface or features. Visual Studio offers an integrated profiling tool that has been deeply enhanced in the 2010 edition. The new profiler takes advantage of the WPF-based user interface of Visual Studio also offering better organization for the collected information layout. To understand how the profiler works, we need a test project. Now, create a new console application in Visual Basic. The best way for understanding how the profiler can improve your productivity is considering the simple example of strings concatenation both using String and StringBuilder objects. In the main module of the console application type this simple code:

```
Module Module1

    Sub Main()
        ConcatenationDemo()
    End Sub
End Module
```

```

    Console.ReadLine()
End Sub

Sub ConcatenationDemo()

    Dim testString As String = String.Empty

    For i = 0 To 10000
        testString += "I love VB 2010"
    Next
End Sub

End Module

```

The preceding code simply creates a big concatenation of String objects. To analyze performance, we now need to start the profiler. Select the **Analyze, Launch Performance Wizard** command. This launches the step-by-step procedure for setting up the profiler running against your project. Figure 58.5 shows the first dialog in the wizard, where you can select the most appropriate profiling technique according to your needs.

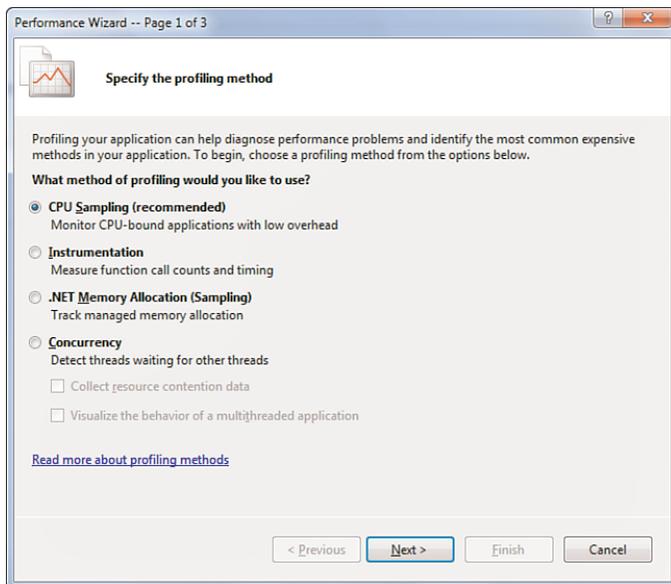


FIGURE 58.5 Selecting the profiling technique.

There are several available modes, all described in Table 58.3.

TABLE 58.3 Available Profiling Modes

Mode	Description
CPU Sampling	Analyzes performances at predetermined intervals for monitoring CPU usage. This is the recommended mode for applications that use few resources; it collects less information but it consumes less system resources.
Instrumentation	Collects complete information on the application performance injecting specific testing code. It is suggested for long-running applications and processes and consumes more system resources.
.NET Memory Allocation	Analyzes memory allocation performance.
Concurrency	Analyzes how multithreaded application consume resources and their performance.

Leave unchanged the default CPU-sampling option and click **Next**. The second dialog allows selecting the executable to be analyzed. As you can see from Figure 58.6, you can choose one of the executables resulting from the current solution or an external executable. Select the current executable (the default option) and continue.

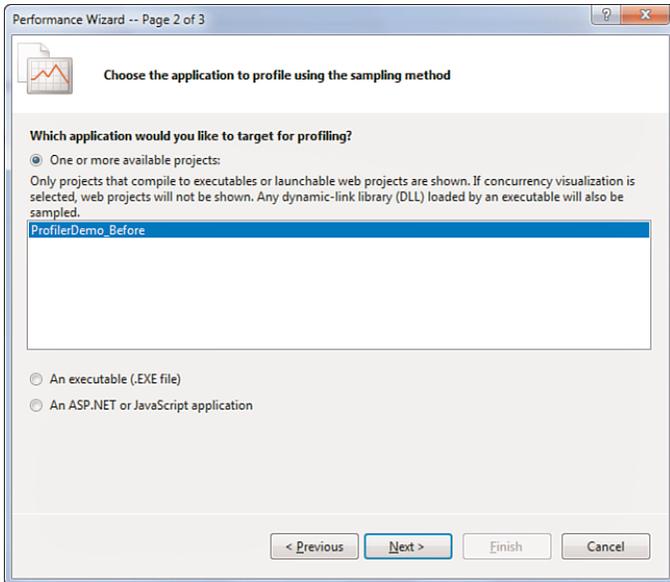


FIGURE 58.6 Selecting the executable to be analyzed.

The last dialog in this wizard is just a summary. Uncheck the one check box available so that the Profiler will not be launched when you click **Finish** (see Figure 58.7).

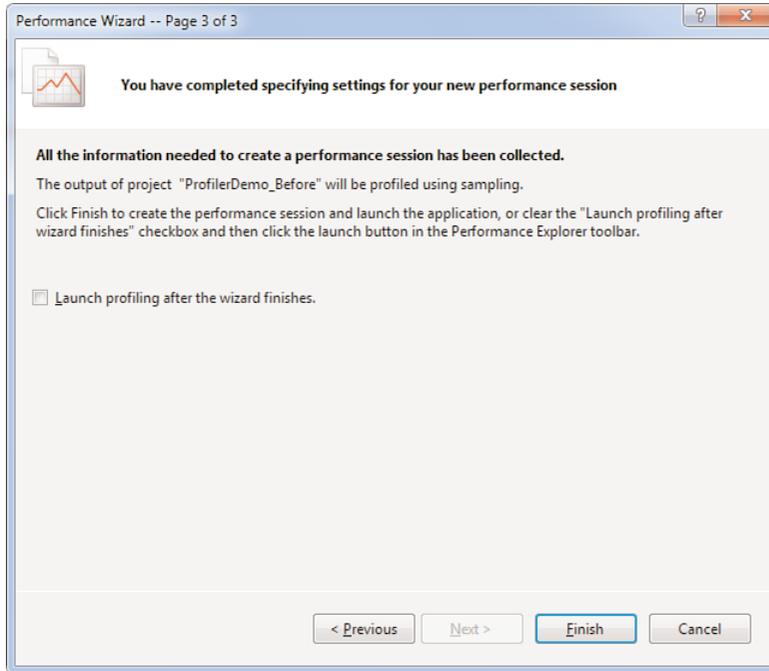


FIGURE 58.7 Completed set up of the Profiler.

You might want to leave the flag on the checkbox to automatically launch the Profiler if the default settings are okay for you, but in this particular scenario we need to make a couple of manual adjustments. When the wizard shuts down, the Performance Explorer tool window appears in the IDE. Because the sample application does not actually stress the CPU very intensively, we need to set a smaller value for the CPU Sampling intervals. To accomplish this, follow these steps:

1. In the Performance Explorer window, right-click the **ProfilerDemo_Before** item.
2. Select the **Properties** command from the popup menu.
3. In the ProfilerDemo_Before Property Pages dialog, select the Sampling node on the left and then replace the default value in the Sampling Interval text box with 50000. This will let the profiler collect data at smaller intervals of CPU clock cycles (see Figure 58.8 for details). This small value is appropriate for small pieces of code like the current example, but you could leave unchanged the default value in real-world applications.

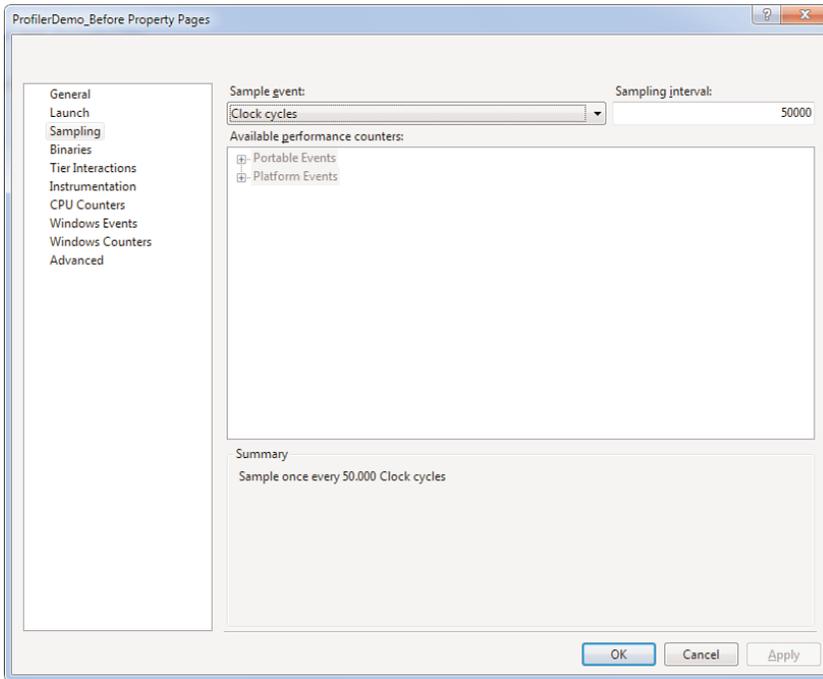


FIGURE 58.8 Setting CPU Sampling interval.

4. Click **OK** to close the dialog and then right-click the **ProfilerDemo_Before** item in Performance Explorer, finally click **Start Profiling** from the popup menu. This will launch the Profiler.

If it is the first time you have run the Profiler, Visual Studio requires your permission for upgrading driver credentials to let the Profiler access system resources for monitoring purposes (see Figure 58.9).

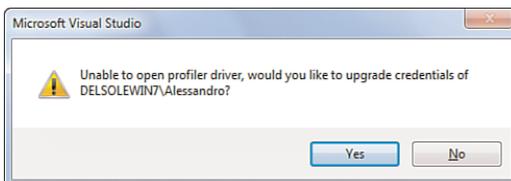


FIGURE 58.9 Upgrading driver profiler credentials.

Visual Studio now runs your application with an instance of the Profiler attached. During all the application lifetime Visual Studio will look like Figure 58.10, showing a work-in-progress message.

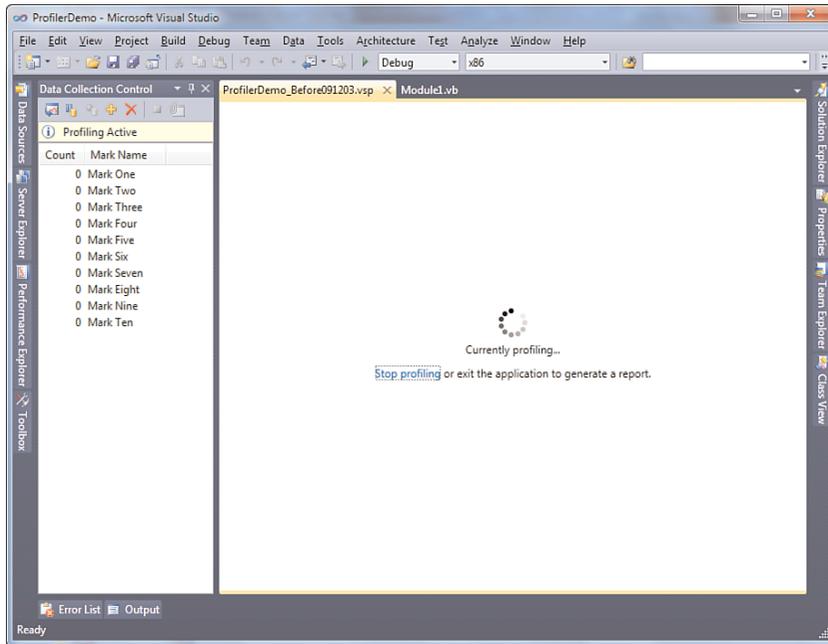


FIGURE 58.10 The profiler is currently running.

The application will be analyzed unless you terminate it. The profiler monitors performances at specific intervals to collect detailed information. When ready, simply close the application. This also detaches the profiler instance. At this point the profiler generates a report about the analysis. Figure 58.11 shows how the report appears.

If you are familiar with the profiler in Visual Studio 2008, you notice some differences but also some improvements in the new reporting system. On the top of the report there is a graph showing the CPU usage at monitored intervals. In the center of the screen there is the Hot Path summary, which shows the most resources consuming function calls and their hierarchy. The analyzed executable has a global impact of 100%, which is a high value and that can be negative for performances. Notice how the `ConcatenationDemo` method makes invocations to `String.Concat`. This is the function doing most individual work, as evidenced in the Hot Path view and in the Functions Doing Most Individual Work graph. It has a 98.65% impact meaning that there is some work to do to improve performance. By default, the profiler report shows results for your code but not for code invoked at a lower level by and within the CLR. Fortunately you can also analyze deeper function calls by clicking the **Show All Code** link in the upper right. Figure 58.12 shows how behind the scenes the sample code invokes the `System.String.wstrcpy` system function that also has a high impact.

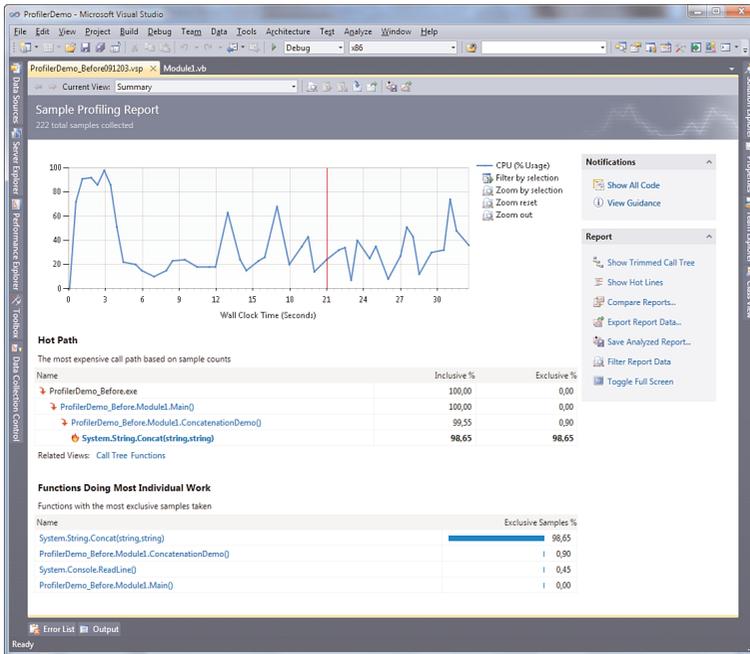


FIGURE 58.11 The analysis report produced by the profiler.

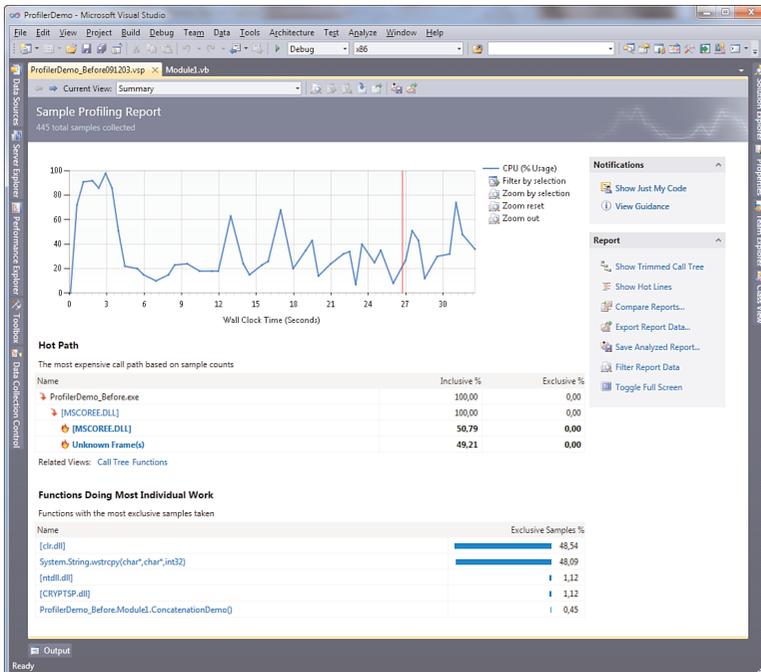


FIGURE 58.12 Showing all code function calls.

The **Current View** combo box enables viewing lots of other useful information, such as processes involved in profiling, function details, lines of code most consuming, and marks. For marks, you can check out intervals and related CPU usage percentage, as shown in Figure 58.13.

Mark ID	Mark Name	Timestamp	Processor_Totals\% Pr...
0	Start of Program	0.00	0
1	AutoMark	67.09	0
2	AutoMark	583.60	72
3	AutoMark	1.106,25	91
4	AutoMark	1.823,32	92
5	AutoMark	2.339,78	86
6	AutoMark	2.897,26	96
7	AutoMark	3.403,48	86
8	AutoMark	3.915,26	51
9	AutoMark	4.440,31	22
10	AutoMark	5.440,35	20
11	AutoMark	6.940,37	15
12	AutoMark	6.940,59	10
13	AutoMark	7.940,45	15
14	AutoMark	8.440,48	23
15	AutoMark	9.444,51	24
16	AutoMark	10.444,58	18
17	AutoMark	11.444,60	18
18	AutoMark	11.944,63	18
19	AutoMark	12.944,69	63
20	AutoMark	13.947,07	24
21	AutoMark	14.447,78	15
22	AutoMark	15.447,84	29
23	AutoMark	15.949,81	26
24	AutoMark	16.950,87	68
25	AutoMark	17.950,91	20
26	AutoMark	18.972,89	35
27	AutoMark	19.483,97	43
28	AutoMark	19.984,01	14
29	AutoMark	20.984,09	24
30	AutoMark	21.984,11	32
31	AutoMark	22.485,10	34
32	AutoMark	22.985,49	7
33	AutoMark	23.485,21	40
34	AutoMark	24.491,60	25
35	AutoMark	24.997,22	35
36	AutoMark	25.997,25	8
37	AutoMark	26.998,78	27
38	AutoMark	27.515,71	51
39	AutoMark	28.016,04	43

FIGURE 58.13 Checking intervals and CPU usage.

The report is saved as a .vsp file that can be useful for later comparisons. You can also export the report to Microsoft Excel or as Xml data (**Export Report Data** command). There is other good news. If you check out the Error List window, you notice a warning message saying that the invocation to `String.Concat` has an impact of 100 and that you should consider using a `StringBuilder` for string concatenations. This means that in most cases Visual Studio can detect the problem and provide the appropriate suggestions to fix it. Following this suggestion, replace the application code as follows:

```
Module Module1

    Sub Main()
        ConcatenationDemo()
        Console.ReadLine()
    End Sub
```

```

Sub ConcatenationDemo()

    Dim testString As New Text.StringBuilder

    For i = 0 To 10000
        testString.Append("I love VB 2010")
    Next

End Sub

End Module
    
```

Now follow the same steps described before about setting the value for the CPU Sampling clock cycles to 50000 for the current project. Then run the Profiler again via the **Analyze, Profiler, Start Profiling** command. After a few seconds, Visual Studio 2010 generates a new report, represented in Figure 58.14.

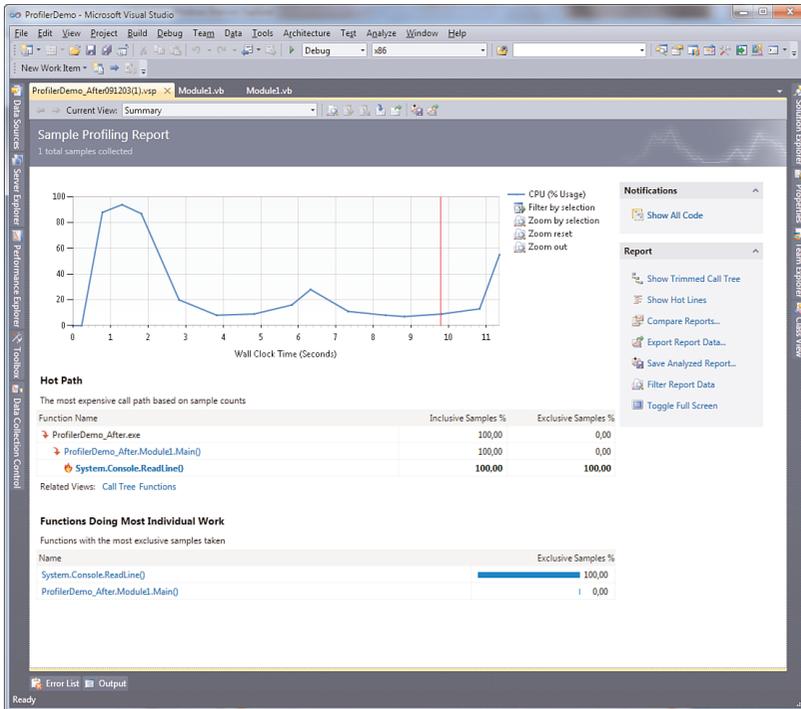


FIGURE 58.14 The new report after code improvements.

Notice how the invocations to strings concatenations method completely disappeared from the performance negative impacts. Also notice how the CPU usage is better than before. This means that our edits improved the application performance, as also demonstrated by the Error List window that is now empty. The most resource-consuming function is now `Console.ReadLine`, which just waits for you to press a key and therefore can be completely ignored. To get a better idea of what actually happened you can compare the two generated reports. Right-click one of the available reports in the Performance Explorer tool window and select **Compare Performance Report**. In the dialog browse for the second report and click **OK**. Figure 58.15 shows the result of the comparison.

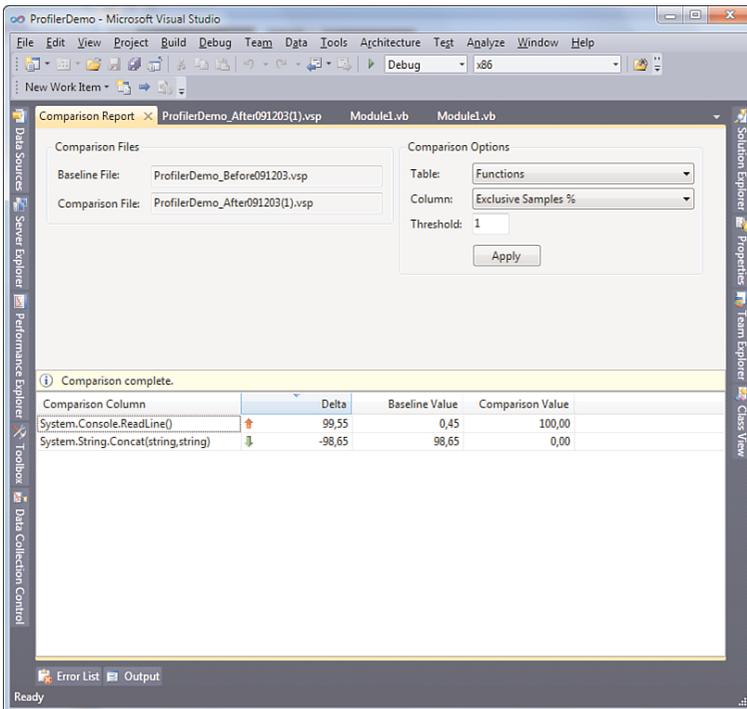


FIGURE 58.15 Comparing two performance reports.

The Delta column indicates the difference between the old value and the new one. The Baseline column indicates the value in the old analysis, whereas Comparison indicates the value in the new analysis. Referring to this specific example, it means that the `String.Concat` function passed from a bad state to a good one. The profiler is a good friend in your developer life, and the suggestion is to profile applications every time you can. The Visual Studio 2010 profiler also offers improvements for profiling multithreaded applications, which is useful against parallel programming techniques.

Profiling External Executables

The Visual Studio profiler is not limited to solutions opened in the IDE, but it can be successfully used for profiling standalone executables. Basically you are allowed profiling also Win32 executables over managed ones. To demonstrate how this works, close the solution opened in Visual Studio (if any) ensuring that nothing is available in Solution Explorer. Now start the Performance Wizard following the steps shown in the previous section until you get the result shown in Figure 58.5. The next step is selecting the **An Executable (.EXE file)** option. When selected this, you need to specify an executable to analyze. Just for demo purposes, select the executable generated in the previous example. Figure 58.16 shows how to accomplish this.

After completing this wizard, Visual Studio launches the specified executable with an instance of the profiler attached. When completed, Visual Studio also generates a report, as shown in the previous section. In this particular case you get nothing but the same profiling results, being substantially the same application. In all other cases you get specific analysis results for the selected executable. The Error List continues to show warning or error messages related to the application performance providing the appropriate suggestions when available. Of course, improving performance requires having the source code for the executable analyzed.

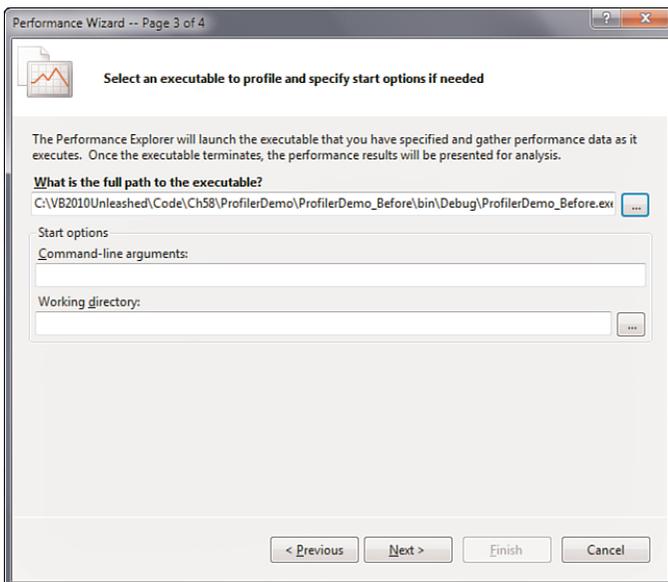


FIGURE 58.16 Selecting a standalone executable.

IntelliTrace, the Historical Debugger

EDITION NOTE

The IntelliTrace debugger is available only with the Visual Studio 2010 Ultimate edition.

One of the most important new tools in Visual Studio 2010 is *IntelliTrace*, formerly known as the historical debugger. This tool can improve your debugging experience because it can record and navigate every event occurring during the application lifetime, such as events and failures including information on specific threads. IntelliTrace is fully integrated with the code editor and with the rest of the IDE functionalities, such as Call Stack and Locals tool windows so that it can provide a complete debugging environment. The tool is capable of recording (to a file, too) and debugging the following happenings:

- ▶ Application events, such as user interface events or application exceptions
- ▶ Playback debugging, which allows deep debugging over specific events occurred before and after a particular code block
- ▶ Unit test failures
- ▶ Load test failures and build acceptances test
- ▶ Manual tests

In this section you learn how to use IntelliTrace to debug application events, exceptions, and unit test failures. Before showing IntelliTrace in action, it is a good idea to set its options in Visual Studio.

IntelliTrace Options

There are a lot of available options for customizing IntelliTrace's behavior. You set its options via the usual **Tools, Options** command and then selecting the IntelliTrace item. Figure 58.17 demonstrates this.

IntelliTrace is enabled by default. The default behavior is that IntelliTrace will collect events only, but for this discussion select **IntelliTrace Events and Call Information** to get complete information on the code calls, too. This kind of monitoring is the most expensive in terms of system resources and should be used only when necessary but it offers a high-level view of what happens during the application lifetime. You have deep control over application events that IntelliTrace can keep track of. Select the **IntelliTrace Events** item. You see a list of .NET elements, such as ADO.NET, Data Binding, and Threading; for each of them you can specify particular events you want to be traced. For example the Gesture events collection enables specifying user interface events such as button clicks, as demonstrated in Figure 58.18.

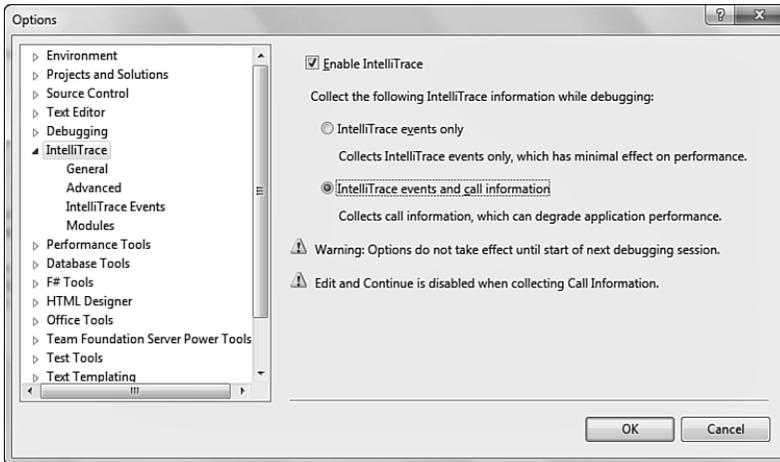


FIGURE 58.17 Setting general options for IntelliTrace.

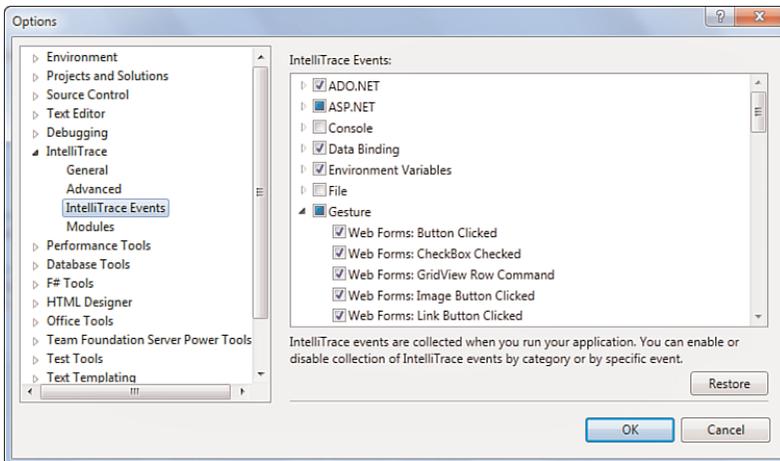


FIGURE 58.18 Selecting events to be tracked by IntelliTrace.

The Advanced and Modules items enable respectively specifying locations and size for logs and which .NET modules must be tracked other than the current application. For this demonstration leave the default settings unchanged. At this point we need a sample application to see IntelliTrace in action.

Creating a Sample Application

The goal of this section is to illustrate how IntelliTrace can save your time by tracking events and exceptions with detailed information that can help you to understand what the problem is. According to this, creating a client WPF application can be a good example. To demonstrate both application events and exceptions, we can place a button

that invokes a method attempting to open a file that does not exist. After creating a new WPF project named (IntelliTraceDemoApp) with Visual Basic, in the XAML code editor, write the code shown in Listing 58.3.

LISTING 58.3 Setting Up the User Interface for the Sample Application

```
<Window x:Class="MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MainWindow" Height="350" Width="525">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Button Name="OpenFileButton" Width="100" Height="50"
      Content="Open file" Grid.Row="0"/>
    <TextBox IsReadOnly="True" Name="FileTextBox"
      Grid.Row="1"/>
  </Grid>
</Window>
```

On the Visual Basic code-behind side, write the code shown in Listing 58.4, which simply handles the `Button.Click` event and tries to open a text file.

LISTING 58.4 Writing the Failing Visual Basic Code Before Running IntelliTrace

Class MainWindow

```
Private Sub OpenFileButton_Click(ByVal sender As System.Object,
  ByVal e As System.Windows.RoutedEventArgs) _
  Handles OpenFileButton.Click

  Me.FileTextBox.Text = OpenFile()
End Sub

'Just for demo purposes
'Consider dialogs implementation in real life apps
Private Function OpenFile() As String

  'Attempting to open a fake file
  Return My.Computer.FileSystem.
```

```
ReadAllText("C:\Alessandro.txt")
```

```
End Function
```

```
End Class
```

Now that we have a sample project, which will voluntarily fail at runtime, it is time to see how to catch problems via IntelliTrace.

Tracking Application Events and Exceptions with IntelliTrace

Run the demo application and click the button to cause an exception. The application breaks because the specified file is not found. At this point the IntelliTrace tool window appears inside Visual Studio 2010, showing a list of events that you can see in Figure 58.19.

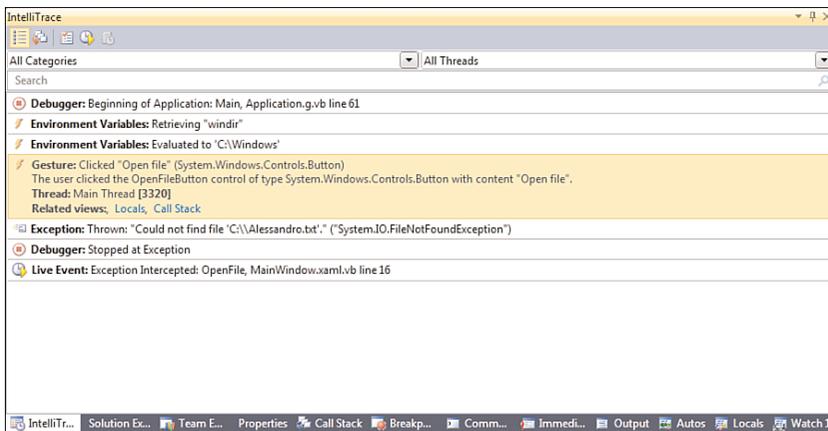


FIGURE 58.19 IntelliTrace in action, showing occurred events.

FILTERING RESULTS

You can filter information by category and by thread. Use the upper combo box to respectively select which kind of events category you want to check out (for example, Console, ASP.NET, ADO.NET, and so on) and which specific thread you want to get tracking information for.

As you can see, IntelliTrace is not just a debugger showing errors or places where errors occurred. It is a powerful tool capable of tracking every single event occurring during the entire application lifetime—meaning that you could even use IntelliTrace to just keep track of events without errors. In the current example notice how the first tracked event is the application startup. There are other events tracked: Click the **Gesture** event to get information on such a happening. IntelliTrace informs you that the user clicked the

OpenFileButton control, which is of type `System.Windows.Controls.Button` and whose content is `Open File`, also showing the thread description and ID. This is a predictable event, but as you can imagine you can keep track of all events, for example when data-binding is performed on a data-bound control or when the user selects an item from a `ListBox` or when the application attempts to make a particular action. This can produce a huge amount of information that you can analyze later. To get information on the exception thrown, simply click the exception item. Visual Studio shows the full error message and the line of code that caused the exception; when you click such an item, the line of code will be automatically highlighted in the code editor for your convenience. IntelliTrace also offers the ability of checking the entire tree of function calls (just in case you enable call information). In a few words, this feature keeps track of every single function call made by the runtime during the application lifetime. To show this, click the **Show Calls View** button in the IntelliTrace toolbar. Figure 58.20 shows an example, pointing to the place where the user clicked the button.

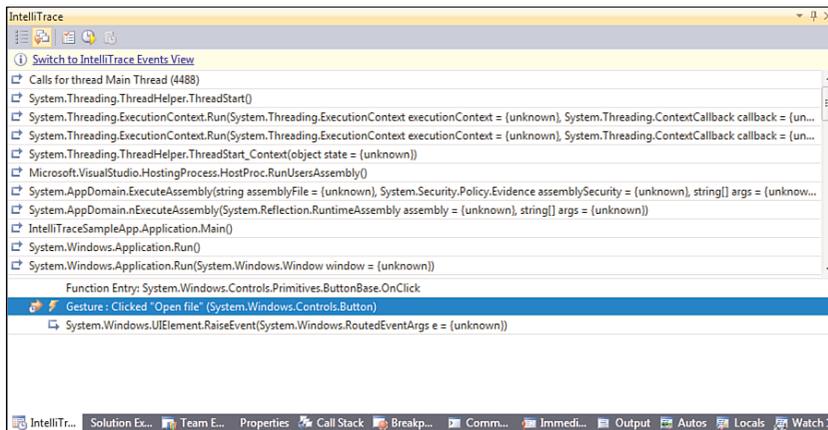


FIGURE 58.20 Showing functions call tree with IntelliTrace.

The sample figure shows the first function call that is a call for starting the main thread. The next functions are calls that the CLR makes to run the application and initialize it; the list is complete, also showing calls made during the rest of the application lifetime until it broke because of the exception. You can easily browse calls using the vertical scrollbar on the right.

Analyzing IntelliTrace Logs

When the application is running with the debugger attached, IntelliTrace records everything happening. (This is the reason why you might notice a small performance decrease when you select the events and call recording option.) Such recordings are saved to log files available in the `C:\ProgramData\Microsoft Visual Studio\10.0\TraceDebugging` folder and can be analyzed directly from within Visual Studio. To accomplish this, follow these steps:

1. In Windows Explorer, open the `C:\ProgramData\Microsoft Visual Studio\10.0\TraceDebugging` folder.
2. Double-click the last log file related to your application. Notice that log file names all begin with the application name but you can open the most recent according to the date modified value.

Visual Studio opens the log file showing a result similar to what you see in Figure 58.21.

Logs contain lots of information. For example, you can select a particular thread in the upper graph and check for the related threads list below. This is useful for understanding at what time a specific thread was tracked. Moreover information on exceptions will also be shown. You can also check about system information and modules involved in the tracking process (such information is placed at the bottom of the page).

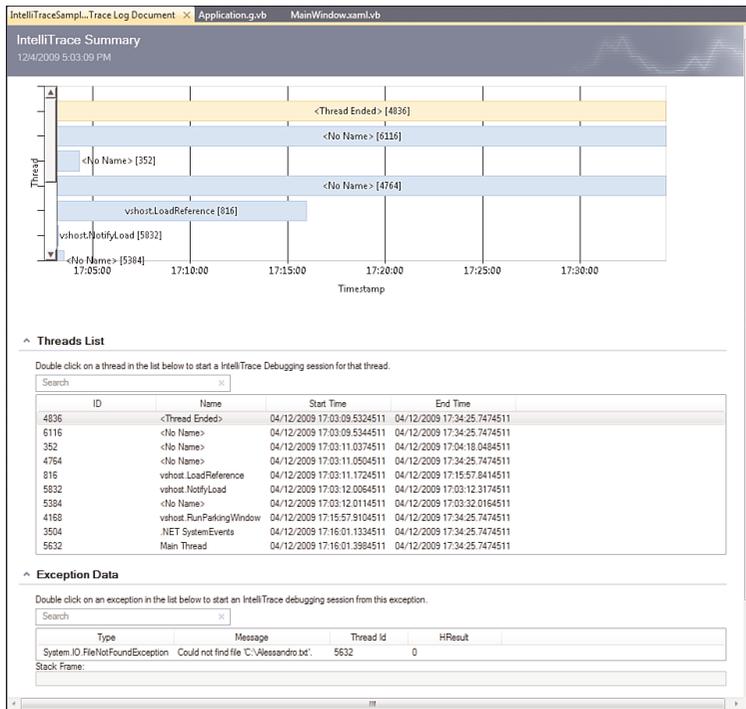


FIGURE 58.21 Analyzing IntelliTrace logs.

Using IntelliTrace for Unit Tests

IntelliTrace is not limited to tracking the application lifetime but can also be used for unit tests for understanding what is behind failures. This particular scenario is discussed in Chapter 59, “Testing Code with Unit Tests, Test-Driven Development, and Code Contracts,” which provides an overview of unit testing and test-driven development.

Generating Dependency Graphs

Another interesting addition, new to VS 2010, is the Dependency Graph generation. Basically this feature enables generating a WPF-based, graphical, browsable view of dependencies between objects in your projects. Dependency graphs can be generated at assembly level (including all types), namespace level (including only types from a given namespace), or at class level. To demonstrate this feature, create a new console project and add the following items:

- ▶ An Entity Data Model mapping the Customers, Orders and Order_Details tables from the Northwind database (see Chapter 27, “Introducing the ADO.NET Entity Framework,” for a recap).
- ▶ A new class named `Helper`, whose purpose is just offering a simple method returning a collection of order details based on the order identifier. The code for this class is shown in Listing 58.5.

LISTING 58.5 Demo Class to Be Mapped into the Graph

```
Imports System.Data.Objects

Public Class Helper

    Public Shared Function GetOrderDetails(ByRef context As NorthwindEntities,
                                          ByVal orderID As Integer) As _
        ObjectQuery(Of Order_Detail)

        Dim result = From det In context.Order_Details
                    Where det.Order.OrderID = orderID
                    Select det

        Return CType(result, Global.System.Data.Objects.
                    ObjectQuery(Of Global.DependencyGraphDemo.Order_Detail))

    End Function
End Class
```

Now select the **Architecture, Generate Dependency Graph, By Assembly** command. After a few seconds the new graph will be available inside Visual Studio 2010. You can then expand items and check for their dependencies with other objects, as represented in Figure 58.22.

To understand items mapping you can check out the legend. To show complete dependencies information, right-click the graph and select the Show Advanced Selection command. This launches the Selection tool window where you can select one or more objects to be investigated and additional objects to be analyzed such as public or private properties. Figure 58.21 shows public properties dependencies from all classes in the project. For example, the `NorthwindEntities` class has dependencies with the `Helper.GetOrderDetails` method because this one receives an argument of type `NorthwindEntities`. The dependency graphs are flexible and can be exported to XPS documents or to images (right-click the graph for additional options).

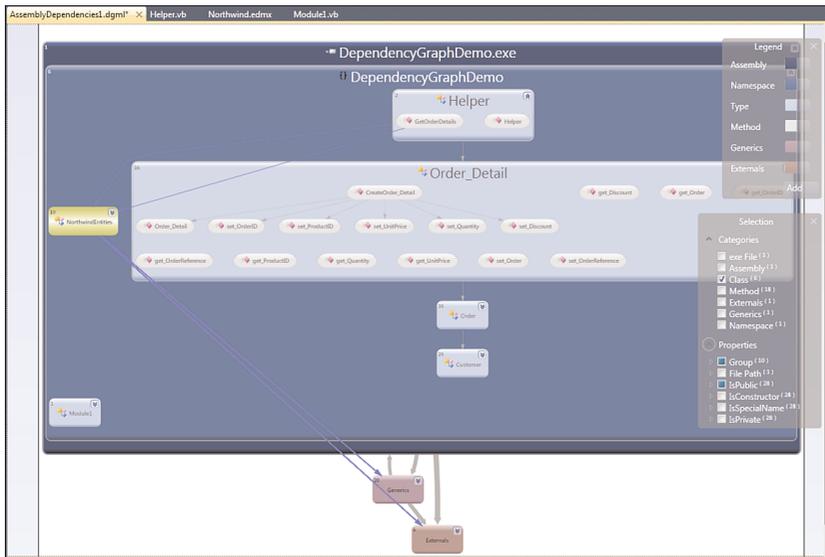


FIGURE 58.22 The newly generated assembly-level dependency graph.

Summary

This chapter covered some important analysis tools available in Visual Studio 2010 Premium and Ultimate that are necessary for improving applications' quality. You discovered how to analyze code for compliance with Microsoft coding rules, required especially when you produce reusable class libraries. Then you saw how to check for code maintainability using the Code Metrics tool. In the second part of the chapter you got information on the integrated Profiler that simplifies checking for performance issues through IDE suggestions; you also saw different usages of the Profiler, including standalone executables. The chapter also focused on the most interesting addition in Visual Studio 2010, the IntelliTrace debugger. Explanations provided information on using this tool for understanding how to keep track of all application events and exceptions during the entire application lifetime, showing also how to analyze summary logs within Visual Studio. Finally, you got an overview of another new addition, the Dependency Graph generation that provides a graphical view of dependencies at assembly, namespace, and class levels.