WPF DEBUGGING AND PERFORMANCE

BY **ALESSANDRO DEL SOLE**

SUCCINCTLY E-BOOK SERIES

Syncfusion®

WPF Debugging and Performance Succinctly

By

Alessandro Del Sole

Foreword by Daniel Jebaraj

Syncfusion®

Deliver innovation with ease®

Technical Reviewer: James McCaffrey
Copy Editor: John Elderkin
Acquisitions Coordinator: Hillary Bowling, online marketing coordinator, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author has been carefully chosen from a pool of talented experts who share our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on numerous topics. The books will always be free. Any updates we publish will also be free.

About the Author

Alessandro Del Sole has been a Microsoft Most Valuable Professional (MVP) since 2008. Awarded MVP of the Year in 2009, 2010, 2011, 2012, and 2014, he is internationally known as a Visual Studio expert and a Visual Basic and .NET authority. Alessandro has authored many printed books and e-books on programming with Visual Studio, including *Visual Studio 2017 Succinctly*, *Visual Studio Code Succinctly*, *Visual Basic 2015 Unleashed*, and *Roslyn Succinctly*. He has written numerous technical articles about .NET, Visual Studio, and other Microsoft technologies in Italian and English for many developer portals, including MSDN Magazine and the former Visual Basic Developer Center from Microsoft. He is a frequent speaker at Italian conferences, and he has released a number of Windows Store apps. He has also produced a number of instructional videos in both English and Italian. Alessandro works as a senior .NET developer, trainer, and consultant. You can follow him on Twitter at @progalex.

Introduction

Windows Presentation Foundation (WPF) has been, for many years, the premiere Microsoft platform for building modern Windows desktop applications. Over the years, both the platform and the development tools have co-evolved to offer the powerful and optimized development experience of the latest versions of Visual Studio and Blend, especially with design-time tools, controls, and the XAML editor.

WPF allows you to build applications based on modern user interfaces, which might include multimedia, graphics, documents, and various data. All of these features require system resources, which means two things are important: optimizing your code to consume system resources in the best possible way and providing optimized performance, including perceived performance. The latter gives users the perception that the application is quick and responsive, even though it must access many resources behind the scenes (e.g., disk I/O, long-running operations over data or over a network). Moreover, designing the user interface and writing code is just a part of the job. You will also spend time debugging your code and analyzing the user interface. Because of its nature, in WPF you might also need to debug XAML data bindings and analyze the behavior of the user interface when rendering graphics, multimedia, or long lists of data-bound items.

In this e-book, you will learn how to debug a WPF application by leveraging all the powerful tools in Visual Studio 2015 and 2017, including the most recent additions that allow you to investigate the behavior of the user interface at runtime. Also, you will learn how to analyze and improve an application's performance in order to provide your customers with the best possible experience and thereby make them happy. If you are not familiar with designing and coding WPF applications, make sure you read the free book *WPF Succinctly* before starting this one.

The only software requirement is Visual Studio 2015 or 2017. You can download the Community edition for free. For Visual Studio 2015, I also strongly recommend that you install the latest update (currently Update 3), which includes new tools for XAML debugging that will be discussed in Chapter 5. Update 3 is already included in the current downloads. If you work with Visual Studio 2017, make sure you've installed the Windows desktop development with .NET workload. This is required to enable WPF development and all the tools and steps described in this book.

Note: In this e-book I will use Visual Studio 2015 to reach a broader audience, but everything discussed here applies to both Visual Studio 2015 and 2017, and to both C# and Visual Basic, except where explicitly noted.

The companion source code for this e-book is available on GitHub at: https://github.com/AlessandroDelSole/WpfDebuggingSuccinctly.

Chapter 1 Debugging WPF Applications

Debugging is one of the most important tasks in the application development lifecycle, representing the process of investigating for errors and of analyzing an application's execution flow over an object's state. Of course, this is true for all development environments and platforms, not just WPF. In the case of WPF, you use Visual Studio and its powerful debugger and instrumentation in order to improve your code quality. This short chapter explains what you will need to debug a WPF application and a number of concepts also follow in the next chapters.

Note: Debugging a WPF application involves using tools that are also available to other development platforms, such as Windows Forms and ASP.NET. If you already have experience with debugging in Visual Studio, this chapter (and the next two chapters) will explain concepts and techniques with which you might already be familiar. However, I'll describe new debugging features in Visual Studio 2015 and 2017; I will also show how to fully leverage tools and functionalities—so keep an eye on the first three chapters.

Debugging in Visual Studio

Debugging an application in Visual Studio means starting your project with an instance of the debugger attached, which you perform simply by pressing F5 or Start on the standard toolbar. When you do this, Visual Studio compiles your solution and starts the resulting output (an .exe file in case of WPF) by attaching the debugger.

Tip: You can press Ctrl+F5 to start the application without the debugger attached.

Before you press F5, you must make sure the Debug configuration has been selected for compilation. This can be done by selecting Debug in either the Solution Configuration combo box in the toolbar or in the Configuration Manager window (see Figure 1) that you can reach by selecting **Build**, **Configuration Manager**.

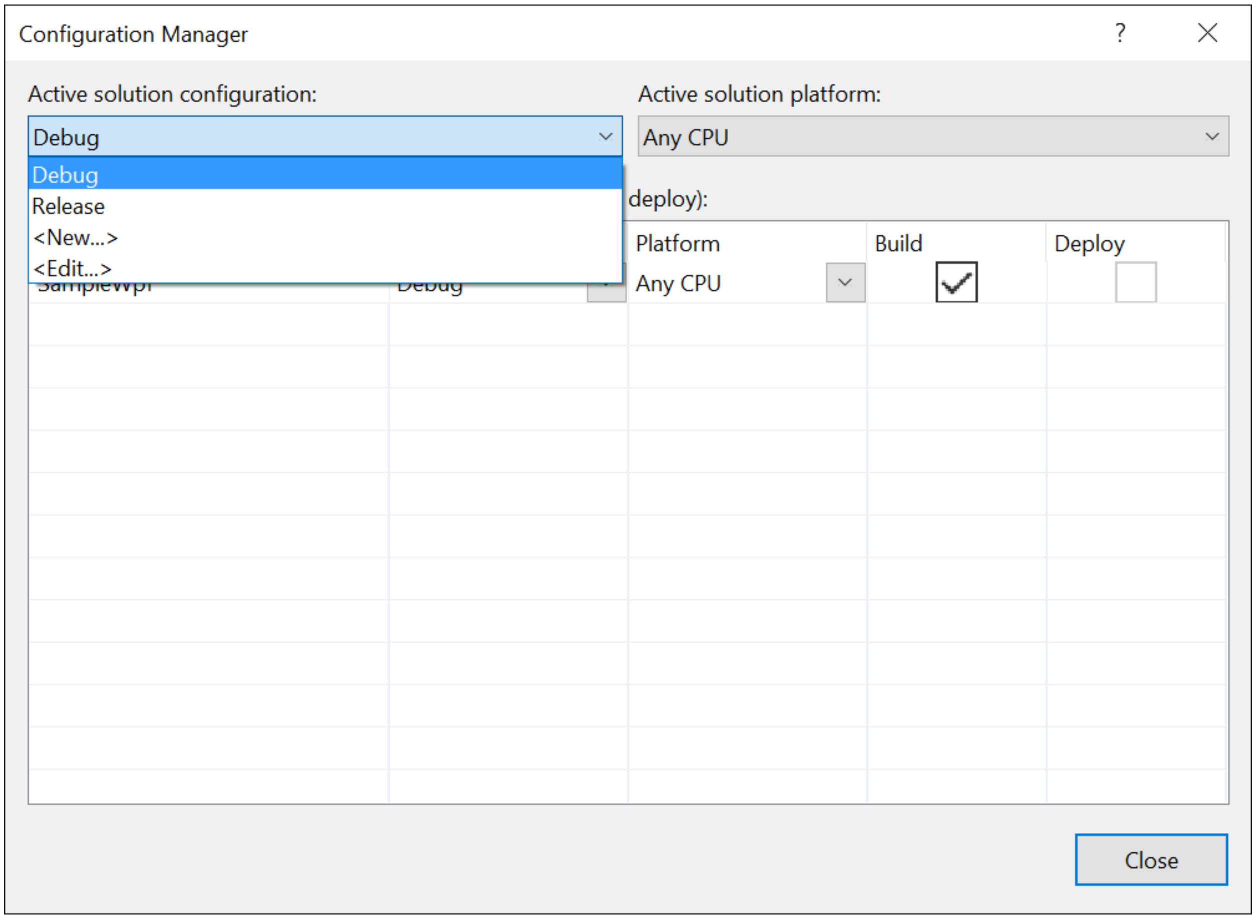
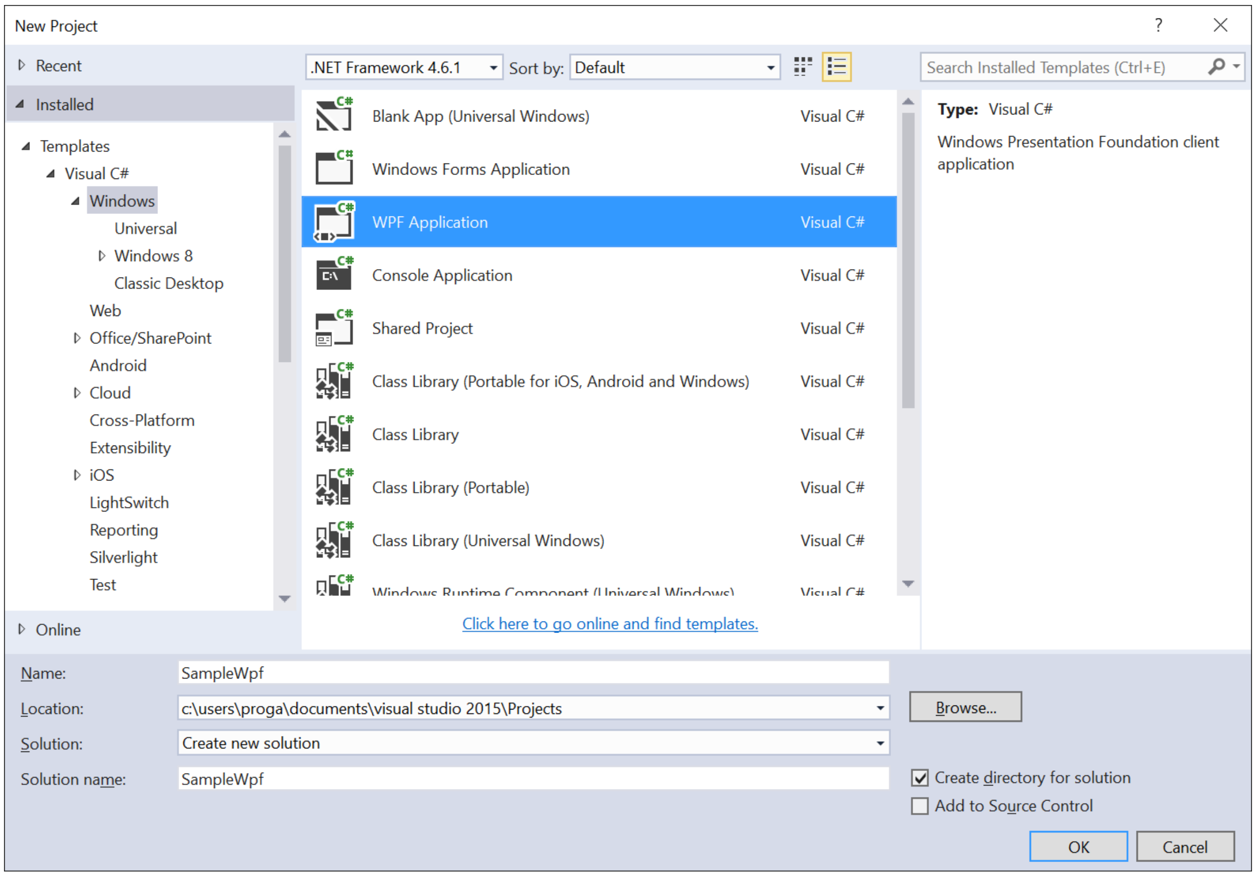


Figure 1: The Configuration Manager Dialog

Without these symbols, the debugger will not be able to collect the necessary information from your code, and the resulting experience will be very poor. Symbols are not generated when you select the Release configuration, and this makes sense because it means you have finished debugging and you are ready to distribute your application. Now, let's prepare an example.

Preparing a sample application

In order to understand how both the debugger and the integrated debugging tools in Visual Studio work, let's work with a sample application. And because the focus is on the tooling, not the code, we do not need a complex project. First, in Visual Studio, we select **File**, **New Project**. In the New Project dialog, select the **WPF Application** template (see Figure 2), name the new project SampleWpf, and click **OK**.


New Project
Recent
Installed
Templates
Visual C#
Windows
Universal
Windows 8
Classic Desktop
Web
Office/SharePoint
Android
Cloud
Cross-Platform
Extensibility
iOS
LightSwitch
Reporting
Silverlight
Test
Online
Blank App (Universal Windows) Visual C#
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Shared Project Visual C#
Class Library (Portable for iOS, Android and Windows) Visual C#
Class Library Visual C#
Class Library (Portable) Visual C#
Class Library (Universal Windows) Visual C#
Windows Runtime Component (Universal Windows) Visual C#
Click here to go online and find templates.
Search Installed Templates (Ctrl+E)
Browse...
OK
Cancel

.NET Framework 4.6.1 Sort by: Default

Type: Visual C#

Windows Presentation Foundation client application

Name: SampleWpf

Location: c:\users\proga\documents\visual studio 2015\Projects

Solution: Create new solution

Solution name: SampleWpf

☑ Create directory for solution

☐ Add to Source Control

Figure 2: Creating a WPF Project

The goal of the sample application is to open a text file, which allows the user to select the file name. This much is enough to demonstrate many debugging features. Of course, more specific examples will be created when necessary. Based on this, the XAML code for the user interface is shown in Code Listing 1, while the code-behind is shown in Code Listing 2.

Code Listing 1

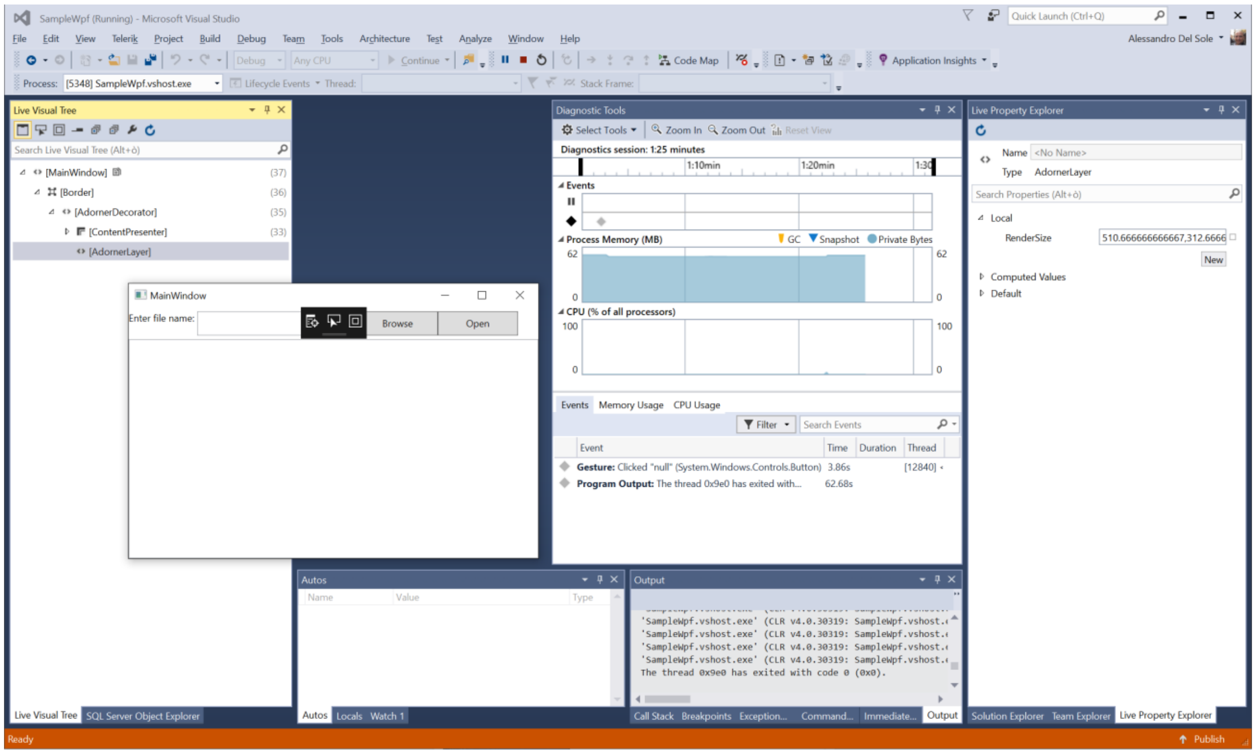
```
<Window x:Class="SampleWpf.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:local="clr-namespace:SampleWpf"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="40"/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <StackPanel Orientation="Horizontal" VerticalAlignment="Center">
            <TextBlock Text="Enter file name: "/>
            <TextBox x:Name="FileNameBox" Width="200" />
            <Button Width="100" Height="30" Content="Browse"
x:Name="BrowseButton" Click="BrowseButton_Click"/>
            <Button Width="100" Height="30" Content="Open"
x:Name="OpenButton" Click="OpenButton_Click"/>
        </StackPanel>
        <TextBox x:Name="ContentBox" Grid.Row="1" />
    </Grid>
</Window>
```

Code Listing 2

```
using Microsoft.Win32;
using System.IO;
using System.Windows;

namespace SampleWpf
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml.
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void OpenButton_Click(object sender, RoutedEventArgs e)
        {
            if (!string.IsNullOrEmpty(this.FileNameBox.Text))
            {
                this.ContentBox.Text = OpenFile(this.FileNameBox.Text);
            }
        }

        private string OpenFile(string fileName)
        {
            return File.ReadAllText(fileName);
        }

        private void BrowseButton_Click(object sender, RoutedEventArgs e)
        {
            var openDialog = new OpenFileDialog();
            openDialog.Title = "Select a .txt file";
            openDialog.Filter = "Text files (.txt)|*.txt";
            if (openDialog.ShowDialog()==true)
            {
                this.FileNameBox.Text = openDialog.FileName;
            }
        }
    }
}
```

Quick overview of debugging tools

When you have completed writing the sample code, press F5. This will start the application for debugging. At this point, a number of debugging and performance-analysis tools for WPF automatically appear (see Figure 3).



Event	Time	Duration	Thread
Gesture: Clicked "null" (System.Windows.Controls.Button)	3.86s		[12840]
Program Output: The thread 0x9e0 has exited with...	62.68s		

```
'SampleWpf.vshost.exe' (CLR v4.0.30319: SampleWpf.vshost.
'SampleWpf.vshost.exe' (CLR v4.0.30319: SampleWpf.vshost.
'SampleWpf.vshost.exe' (CLR v4.0.30319: SampleWpf.vshost.
'SampleWpf.vshost.exe' (CLR v4.0.30319: SampleWpf.vshost.
The thread 0x9e0 has exited with code 0 (0x0).
```

Figure 3: Debugging and Performance Tools

The black rectangle with three small buttons that overlays the main window of the application was first introduced in Visual Studio 2015 Update 2, and it allows for quick investigation into the user interface's behavior at runtime (this will be discussed further in Chapter 5). This rectangle can be minimized when not needed. The Visual Studio IDE shows a number of tool windows:

- Live Visual Tree and Live Property Explorer, which allow for investigation of the behavior of the user interface at runtime.
- Diagnostic Tools, which allows for performance analysis and will be discussed later in Chapter 7 Analyzing the Application Performances.
- Autos, Local, and Watch 1, which allow for variable and expression evaluations.

- Breakpoints, which provides information about breakpoints.
- Output, which shows all the information the debugger catches. This includes details about the entire lifecycle, not just errors or other issues.

These tool windows will be discussed thoroughly in Chapter 3 Working with Debug Windows (except for Live Visual Tree and Live Property Explorer, which will be discussed in Chapter 5 XAML Debugging). Of course, there are other useful tool windows and commands you can invoke from the Debug menu. Starting in the next chapter, we will start debugging the sample application in order to gain a deeper knowledge about the power of both the debugger and the Visual Studio IDE.

Chapter summary

Debugging WPF applications involves the debugger that ships with Visual Studio and many integrated tools, such as commands and tool windows. Visual Studio 2015 Update 2 has also introduced new tools that make it easier to investigate the behavior of the user interface at runtime. You must be careful about enabling the Debug configuration, which allows the IDE to generate the proper symbols and makes the debugger capable of collecting all the necessary information about your code.

Chapter 2 Stepping Through Code

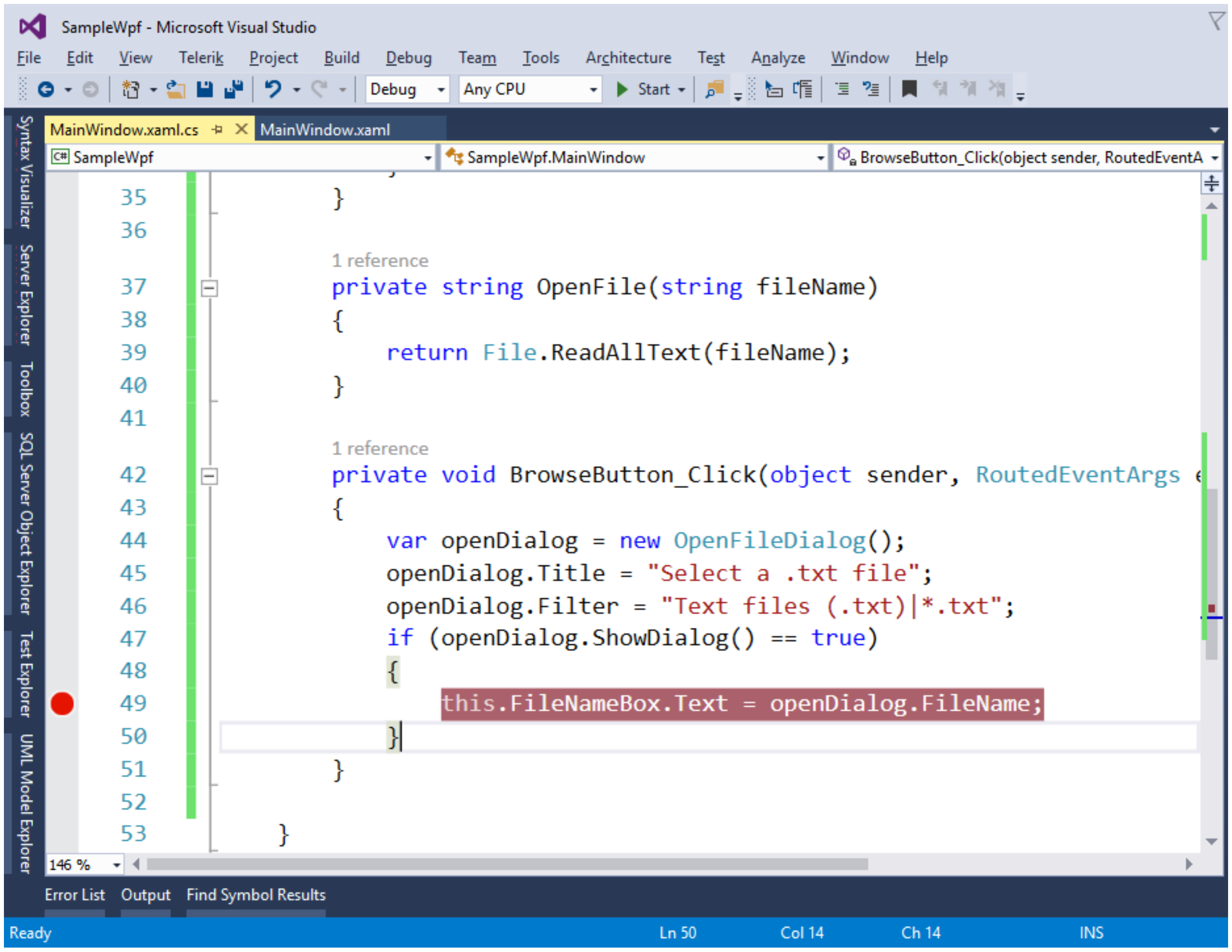
The debugger is one of the most powerful tools in the Visual Studio development experience. For instance, it allows us to execute single lines of code or small sets of lines and investigate the behavior of code along with referenced variables. Debugging code starts in the code editor, so this chapter will address how to work with breakpoints, data tips, and special commands that execute a limited set of lines of code per time. What you will learn in this chapter applies to any kind of application you build with Visual Studio, but here we will work with a WPF application.

Introducing breakpoints and data tips

A breakpoint causes the execution of the application to halt at the point where it is placed. When the application breaks because of a breakpoint, Visual Studio enters into break mode. In break mode, you can investigate the value of local variables, you can execute code in steps, you can evaluate expressions with debugging windows, and you can resume the execution when ready. Placing one or more breakpoints in your code is easy. In fact, you can either press F9 on the line of code you want to debug or you can click the leftmost column in the code editor window. You can also select Debug, Toggle Breakpoint. When you add a breakpoint, the line of code is highlighted in red. For example, in the sample application created in Chapter 1, add a breakpoint on the following line inside the **`BrowseButton_Click`** event handler:

```
this.FileNameBox.Text = openDialog.FileName;
```

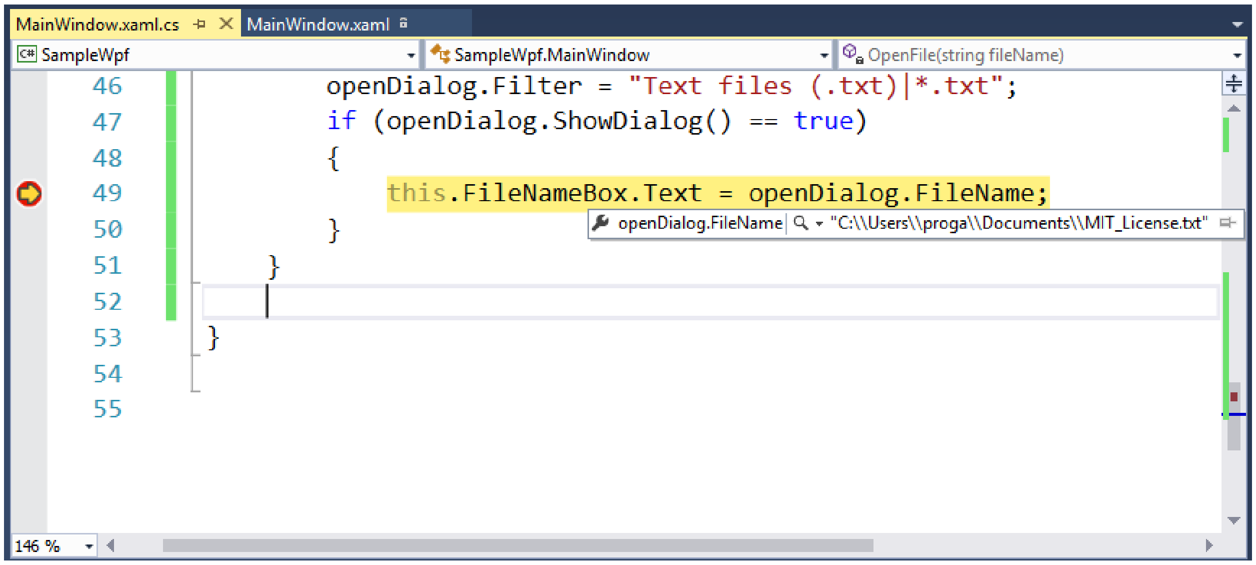
The line will be highlighted in red, as shown in Figure 4.



```
        }

        1 reference
        private string OpenFile(string fileName)
        {
            return File.ReadAllText(fileName);
        }

        1 reference
        private void BrowseButton_Click(object sender, RoutedEventArgs e
        {
            var openDialog = new OpenFileDialog();
            openDialog.Title = "Select a .txt file";
            openDialog.Filter = "Text files (.txt)|*.txt";
            if (openDialog.ShowDialog() == true)
            {
                this.FileNameBox.Text = openDialog.FileName;
            }
        }

    }
```

Figure 4: Adding Breakpoints

Notice how the scroll bar in the code editor reports small red squares to help you find breakpoints in your code. Now, press F5 to start debugging the application. When running, click **Browse** and select any file on your PC. When Visual Studio encounters the breakpoint, it breaks the execution by entering the break mode, and it highlights in yellow the line that is currently debugged before that line is executed (see Figure 5). If you hover over the **`openDialog.FileName`** object, you will see a small tooltip that shows its current value (see Figure 5). If you do the same over the **`this.FileNameBox.Text`** property, the tooltip will show that it has no value yet, which is correct because that line of code has not been executed. Such tooltips go under the name of **Data Tips**. Now you have many options: you can step through the code with one of the debugging commands (described shortly), such as Step Into (F11), or you can resume the application execution with F5.



```
            openDialog.Filter = "Text files (.txt)|*.txt";
            if (openDialog.ShowDialog() == true)
            {
                this.FileNameBox.Text = openDialog.FileName;
            }
        }

}
```

openDialog.FileName "C:\\Users\\proga\\Documents\\MIT_License.txt"

Figure 5: Investigating Variables' Value with Data Tips

Run-to-click in Visual Studio 2017

Before Visual Studio 2017, you had to introduce temporary breakpoints to continue the execution from a breakpoint to a certain point in your code. Visual Studio 2017 took a step forward and introduced a new feature called **Run to Click**. When the debugger enters in break mode and you hover over a line of code, a green glyph appears near the line. This glyph represents the Run to Click button. If you click it, your code will be executed to that line, without the need of temporary breakpoints. Actually, the line of Run to Click is highlighted and not executed, exactly as would happen if a breakpoint was set on that line. You can find a more detailed explanation in my *Visual Studio 2017 Succinctly* e-book.

Intentionally and unintentionally breaking the application execution

Breakpoints allow you to halt the application execution intentionally at a certain point, so that you can investigate the behavior of your code. You can also intentionally break the application execution in code, using the **`System.Diagnostics.Debugger`** class and its **`Break`** method as follows:

```
System.Diagnostics.Debugger.Break();
```

When this line of code is encountered, the debugger enters the break mode. Of course, there are situations when the application execution breaks unintentionally—this is typically the case with unhandled runtime errors.

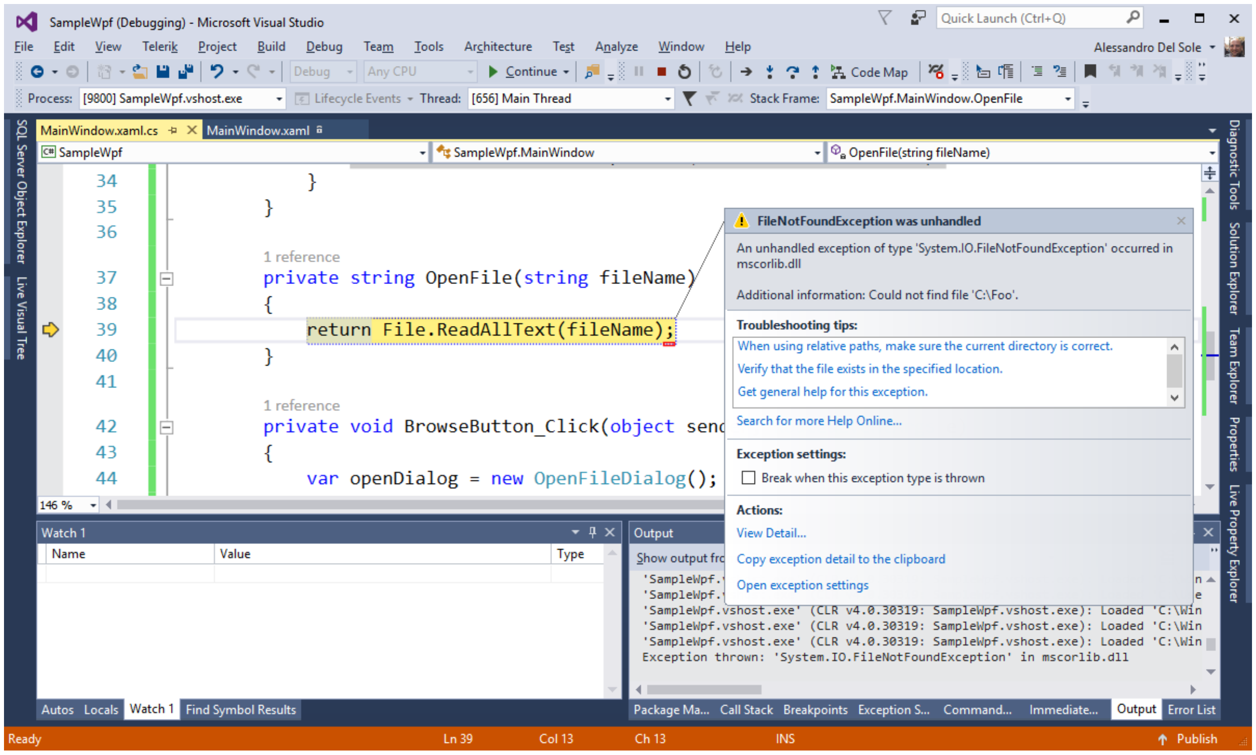
Tip: The `Debugger` class provides a communication channel with the debugger. The `IsAttached` property returns true if an instance of the debugger is attached to the process; the `Log` method allows sending a message to the debugger; the `Launch` method allows launching and attaching the debugger to the application process.

Removing breakpoints

Removing breakpoints is as easy as adding them. You can remove a breakpoint using F9, or by clicking the leftmost column in the code editor, or by selecting Debug, Toggle Breakpoint on the breakpoint's current line. You can remove all the breakpoints by pressing Ctrl+Alt+F9 or by selecting Debug, Delete All Breakpoints.

Understanding runtime errors

Runtime errors are unpredictable. They happen while the application is running, and they are often caused by programming errors that cannot be detected at compile time. As an example, suppose you give users the option to specify a file name, but then the file is not found on disk. In real-world scenarios, you would certainly implement `try..catch` blocks, predict as many errors as possible, and handle those errors the proper way. However, while debugging, you might encounter an unhandled error. For instance, run the sample application again, but instead of selecting a file name with the Browse button, enter a file name that does not exist in the text box, then click **Open**. At this point, the application execution will break because of an unhandled error, and Visual Studio will enter the break mode, highlighting in yellow the line of code that caused the exception (see Figure 6).


SampleWpf (Debugging) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Telerik Project Build Debug Team Tools Architecture Test Analyze Window Help
Alessandro Del Sole
Debug
Any CPU
Continue
Code Map
Process: [9800] SampleWpf.vshost.exe
Lifecycle Events
Thread: [656] Main Thread
Stack Frame: SampleWpf.MainWindow.OpenFile
MainWindow.xaml.cs
MainWindow.xaml
SampleWpf
SampleWpf.MainWindow
OpenFile(string fileName)
146 %
Watch 1
Name
Value
Type
Autos Locals Watch 1 Find Symbol Results
Output
Show output fro
Package Ma... Call Stack Breakpoints Exception S... Command... Immediate... Output Error List
SQL Server Object Explorer
Live Visual Tree
Diagnostic Tools
Solution Explorer
Team Explorer
Properties
Live Property Explorer
Ready
Ln 39
Col 13
Ch 13
INS
Publish

```
            }
        }

        1 reference
        private string OpenFile(string fileName)
        {
            return File.ReadAllText(fileName);
        }

        1 reference
        private void BrowseButton_Click(object send
        {
            var openDialog = new OpenFileDialog();
```

FileNotFoundException was unhandled

An unhandled exception of type 'System.IO.FileNotFoundException' occurred in mscorlib.dll

Additional information: Could not find file 'C:\Foo'.

Troubleshooting tips:

When using relative paths, make sure the current directory is correct.
Verify that the file exists in the specified location.
Get general help for this exception.

Search for more Help Online...

Exception settings:

☐ Break when this exception type is thrown

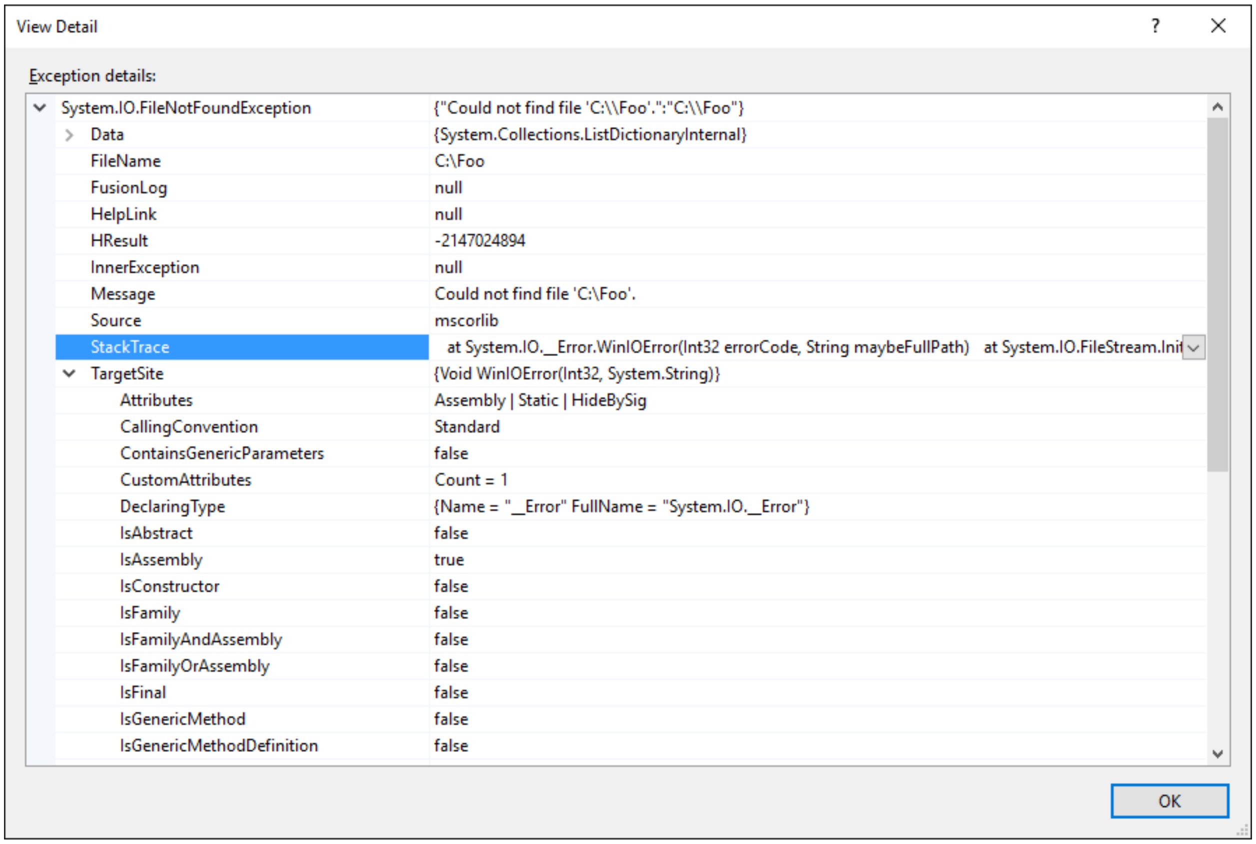
Actions:

View Detail...
Copy exception detail to the clipboard
Open exception settings

```
'SampleWpf.
'SampleWpf.
'SampleWpf.vshost.exe' (CLR v4.0.30319: SampleWpf.vshost.exe): Loaded 'C:\Win
'SampleWpf.vshost.exe' (CLR v4.0.30319: SampleWpf.vshost.exe): Loaded 'C:\Win
'SampleWpf.vshost.exe' (CLR v4.0.30319: SampleWpf.vshost.exe): Loaded 'C:\Win
Exception thrown: 'System.IO.FileNotFoundException' in mscorlib.dll
```

Figure 6: Runtime Error Causes Application Execution to Break

In this case, the highlighted line of code is searching for a file that does not exist, but an appropriate **`try..catch`** block has not been supplied, therefore the debugger broke at this point. If you fall into this situation, you can see the exception details window (the grey pop-up), which not only shows information about the error, such as the bad file name (a **`FileNotFoundException`** in this case), but also allows you to get details by clicking the **View Detail** hyperlink in the Actions area. This will open the View Details window (see Figure 7), where you can get information about the stack trace and which will allow you to investigate the method calls stack. It will also open the **`InnerException`** object, which is useful if the current exception has been caused by another exception. Also, you will get exception-specific information, such as the **`FileName`** property value, which is exposed by the **`FileNotFoundException`** class.



System.IO.FileNotFoundException	{"Could not find file 'C:\\Foo'.":"C:\\Foo"}
Data	{System.Collections.ListDictionaryInternal}
FileName	C:\Foo
FusionLog	null
HelpLink	null
HResult	-2147024894
InnerException	null
Message	Could not find file 'C:\Foo'.
Source	mscorlib
StackTrace	at System.IO.__Error.WinIOError(Int32 errorCode, String maybeFullPath) at System.IO.FileStream.Init
TargetSite	{Void WinIOError(Int32, System.String)}
Attributes	Assembly | Static | HideBySig
CallingConvention	Standard
ContainsGenericParameters	false
CustomAttributes	Count = 1
DeclaringType	{Name = "__Error" FullName = "System.IO.__Error"}
IsAbstract	false
IsAssembly	true
IsConstructor	false
IsFamily	false
IsFamilyAndAssembly	false
IsFamilyOrAssembly	false
IsFinal	false
IsGenericMethod	false
IsGenericMethodDefinition	false

Figure 7: Getting Details about Problems Causing an Exception at Runtime

Do not forget to enclose code that might potentially encounter runtime errors inside **`try..catch`** blocks, especially in the user interface's code-behind, in order to avoid these situations and also in order to allow users to make a decision based on the context.

Fixing code at runtime with Edit and Continue

In some situations, you are allowed to change code that has caused an error while in break mode, then resume the application execution. This feature is known as Edit and Continue and, starting from Visual Studio 2015, it has been enhanced to support lambda expressions. You can also change some code by pausing the application execution (e.g., not because of an error) by clicking Pause on the Debug toolbar. In some cases, you will not be able to use Edit and Continue; for example, if your changes will influence the general application behavior, you will need to stop, edit, and restart your code. And, by the way, Visual Studio will tell you if Edit and Continue is available when you are editing your code in break mode.

Stepping through code

Sometimes your code might seem to have no bugs, but at runtime the application will not work as expected. For cases in which you must find subtle bugs, debugging can be a very complicated task. For this reason, the powerful debugger in Visual Studio allows you to step through code, which means executing one line of code, or a limited set of lines of code, and investigating that behavior. You can step through your code using a number of debugging commands, all available in both the Debug menu and with keyboard shortcuts, as will be described in detail in this chapter.

Tip: In most cases, you will step through your code while in break mode. However, two debugging commands, Step Into and Step Over, allow you to start debugging and step through code directly. This is useful if you want to investigate the behavior of your WPF application from startup.

Before continuing, place a breakpoint on any method or event handler in the sample application. This will help you understand how the following debugging commands work.

Step Into and Step Over

Step Into (F11) and Step Over (F10) both execute one instruction at a time. Here is the difference between them: if the instruction to be executed is a method, Step Over does not enter the method body and will complete its execution before going back to the caller. However, Step Into enters a method body and executes one instruction at a time. Step Over is useful when you must debug a method that invokes previously tested and debugged methods that you need not check every time.

Step Out

Step Out (Shift+F11) only works within methods. It executes all the lines of code next to the current line until the method completes. For example, if you place a breakpoint on the **`if`** block in the **`OpenButton_Click`** event handler in the sample application, the debugger will break on that line. However, when using Step Out, it will execute all the following lines until the event handler completes.

Run To Cursor

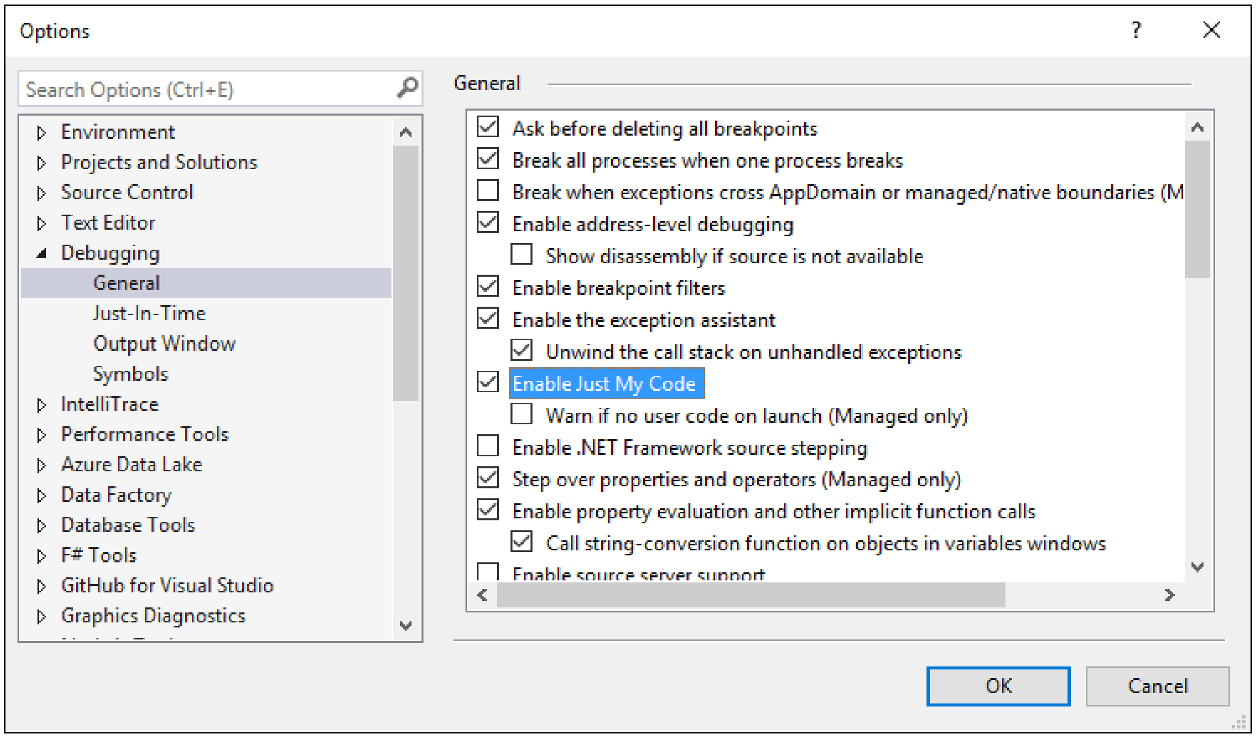
With Run To Cursor, you can place the cursor on any line of code, right-click, then select Run To Cursor. This will cause the debugger to execute all the lines until the selected one, then it will break and highlight the current line in yellow. This is useful when you want to debug a specific portion of code without using breakpoints.

Set Next Statement and Show Next Statement

While in break mode, you have the option to set and show the next statement that will be executed after resuming the application or after a breakpoint or stop. You can right-click the next statement you want to execute, thus excluding all the other lines from stepping through code, then select Set Next Statement. By contrast, with Show Next Statement, you can quickly move the cursor onto the next executable statement. This can be particularly useful with long code files where breakpoints are not immediately visible.

Debugging user code only

A WPF application is made of the code you write plus system code. You can decide whether to debug only your code or to debug system code, too. This feature is known as Just My Code, and it is enabled by default, which means the debugger will focus on user code only. In order to change this option and include system code, you must open the debugging options. This can be accomplished by selecting **Tools**, **Options**, then by opening the **Debugging** node in the Options window (see Figure 8).



- ☑ Ask before deleting all breakpoints
- ☑ Break all processes when one process breaks
- ☐ Break when exceptions cross AppDomain or managed/native boundaries (M
- ☑ Enable address-level debugging
  - ☐ Show disassembly if source is not available
- ☑ Enable breakpoint filters
- ☑ Enable the exception assistant
  - ☑ Unwind the call stack on unhandled exceptions
- ☑ Enable Just My Code
  - ☐ Warn if no user code on launch (Managed only)
- ☐ Enable .NET Framework source stepping
- ☑ Step over properties and operators (Managed only)
- ☑ Enable property evaluation and other implicit function calls
  - ☑ Call string-conversion function on objects in variables windows
- ☐ Enable source server support

Figure 8: Changing the Default for Just My Code

Remember that disabling Just My Code will cause Visual Studio and the debugger to load more symbols and to monitor additional resources, which means doing so might slow your debugging experience. Disable it only when strictly required.

Enabling native code debugging

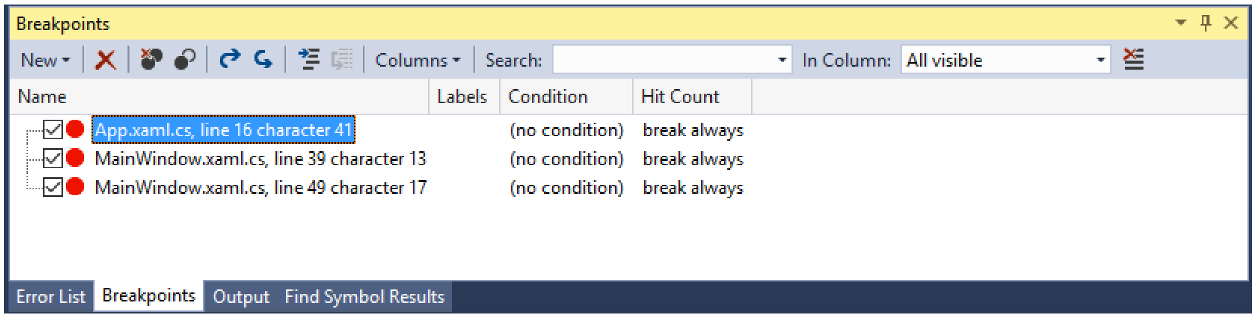
In case your WPF application invokes unmanaged code (such as the Win32 API), you can enable native code debugging. You can do this by opening the Properties window of a project and selecting the Enable native code debugging check box in the Debug tab. As for disabling Just My Code, enable native code debugging only when strictly required.

Customizing breakpoints

Now that you know how to set breakpoints and step through your code in break mode, it's time to learn how to get the most out of breakpoints by using some interesting features.

Managing breakpoints with the Breakpoints window

By pressing Ctrl+Alt+B, you enable the Breakpoints window, which is where you can easily manage all the breakpoints in your solution via a convenient user interface (see Figure 9). Simply hover over each button in the toolbar to get a description.



Name	Labels	Condition	Hit Count
✓ ● App.xaml.cs, line 16 character 41		(no condition)	break always
✓ ● MainWindow.xaml.cs, line 39 character 13		(no condition)	break always
✓ ● MainWindow.xaml.cs, line 49 character 17		(no condition)	break always

Figure 9: Managing Breakpoints in the Breakpoints Window

You can select or unselect the check box for a breakpoint in order to temporarily disable or enable a breakpoint. The toolbar has buttons to add, remove, and even import and export breakpoints. In fact, Visual Studio can store the list of breakpoints into an XML file for later use. You can also click Columns on the toolbar and add additional columns to the window for further details on each breakpoint. Such a window also shows important information, including labels, conditions, and hit counts.

Providing breakpoint labels

Though the Breakpoints window provides a convenient way to work with breakpoints, if you have dozens of breakpoints in your code, it can be very hard to remember what each breakpoint deals with, and viewing the file and line number where the breakpoint is located might not be very helpful. Visual Studio allows adding and editing breakpoint labels—a kind of identifier that will help you categorize, find, and manage breakpoints more easily. In order to provide a label to a breakpoint, you have several options:

- In the Breakpoints window, right-click a breakpoint, then select **Edit labels**.
- In the code editor, right-click the red glyph that identifies a breakpoint, then select **Edit labels**.

Whichever option you choose, you will see the Edit breakpoint labels window appear. For instance, suppose you have a breakpoint on the OpenFile method definition. In the Edit Breakpoint Labels window, you can type a description as shown in Figure 10.

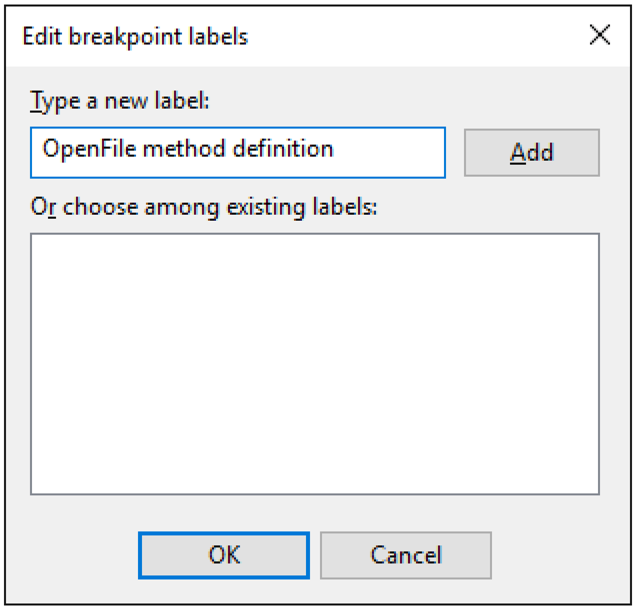


Figure 10: Assigning a Label to a Breakpoint

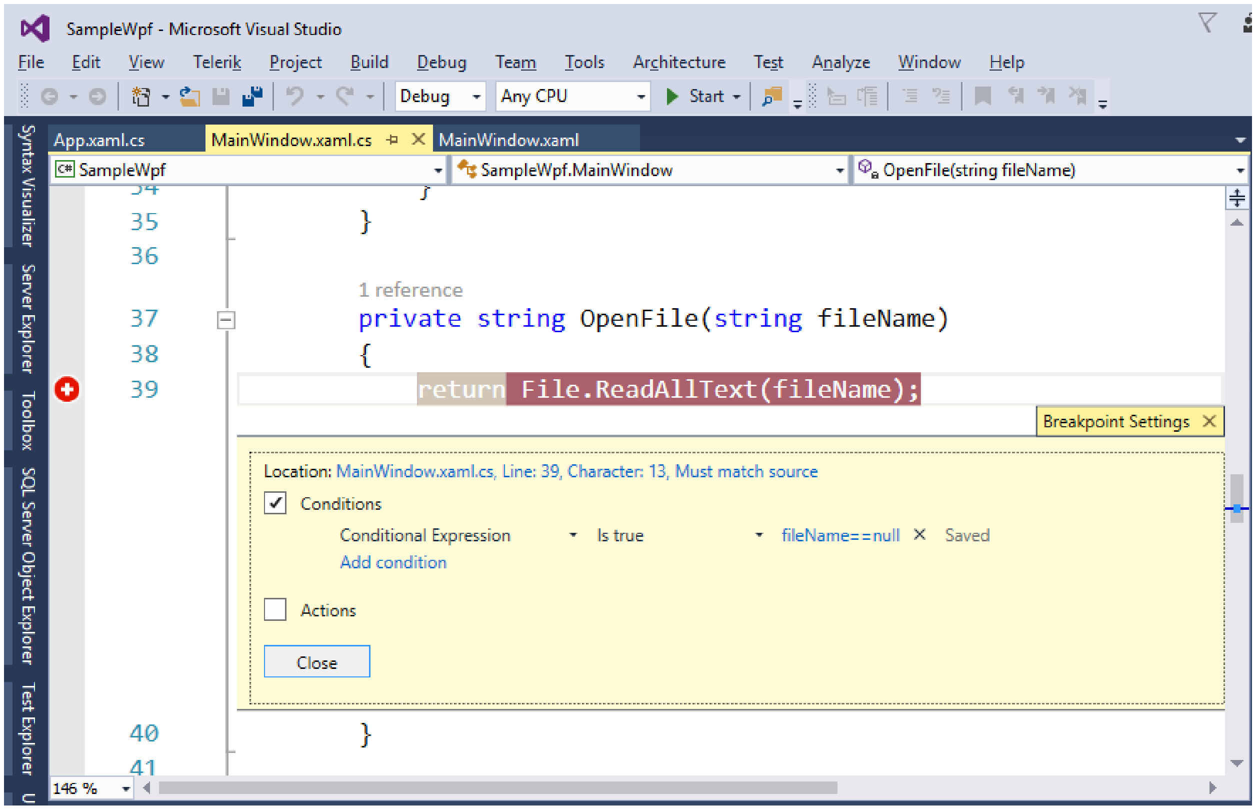
Click **Add**, then click **OK**. The new label will be immediately visible in the Breakpoints window, making it easier to remember what a breakpoint is about.

Setting breakpoint conditions

Conditions allow you to specify when the application execution must break as a breakpoint is hit. In order to understand how conditions work, place a breakpoint inside the body of the `OpenFile` method, thus the `return` statement. Then right-click the breakpoint's red icon and select **Conditions**. At this point, you can specify one or more conditions that will make the execution break when this breakpoint is encountered.

Note: The user interface for adding conditions and actions has changed starting with Visual Studio 2015. Now you have a convenient pop-up that allows you to keep your focus on the active editor window instead of modal dialogs, as with previous editions.

Imagine you want that breakpoint to cause the application execution break only if the supplied string is null. This kind of condition requires an expression to be evaluated; that is, the supplied string is equal to null. The Conditional Expression option allows evaluating conditions that must be true in order to break the application execution when the breakpoint is encountered. In this case, you can specify the condition shown in Figure 11.


SampleWpf - Microsoft Visual Studio
File Edit View Telerik Project Build Debug Team Tools Architecture Test Analyze Window Help
Debug
Any CPU
Start
App.xaml.cs
MainWindow.xaml.cs
MainWindow.xaml
SampleWpf
SampleWpf.MainWindow
OpenFile(string fileName)
Breakpoint Settings
Syntax Visualizer
Server Explorer
Toolbox
SQL Server Object Explorer
Test Explorer
146 %

```
        }

        1 reference
        private string OpenFile(string fileName)
        {
            return File.ReadAllText(fileName);
```

Location: MainWindow.xaml.cs, Line: 39, Character: 13, Must match source

☑ Conditions

Conditional Expression | Is true | fileName==null ✕ Saved

Add condition

☐ Actions

Close

```
        }
```

Figure 11: Specifying a Breakpoint Condition

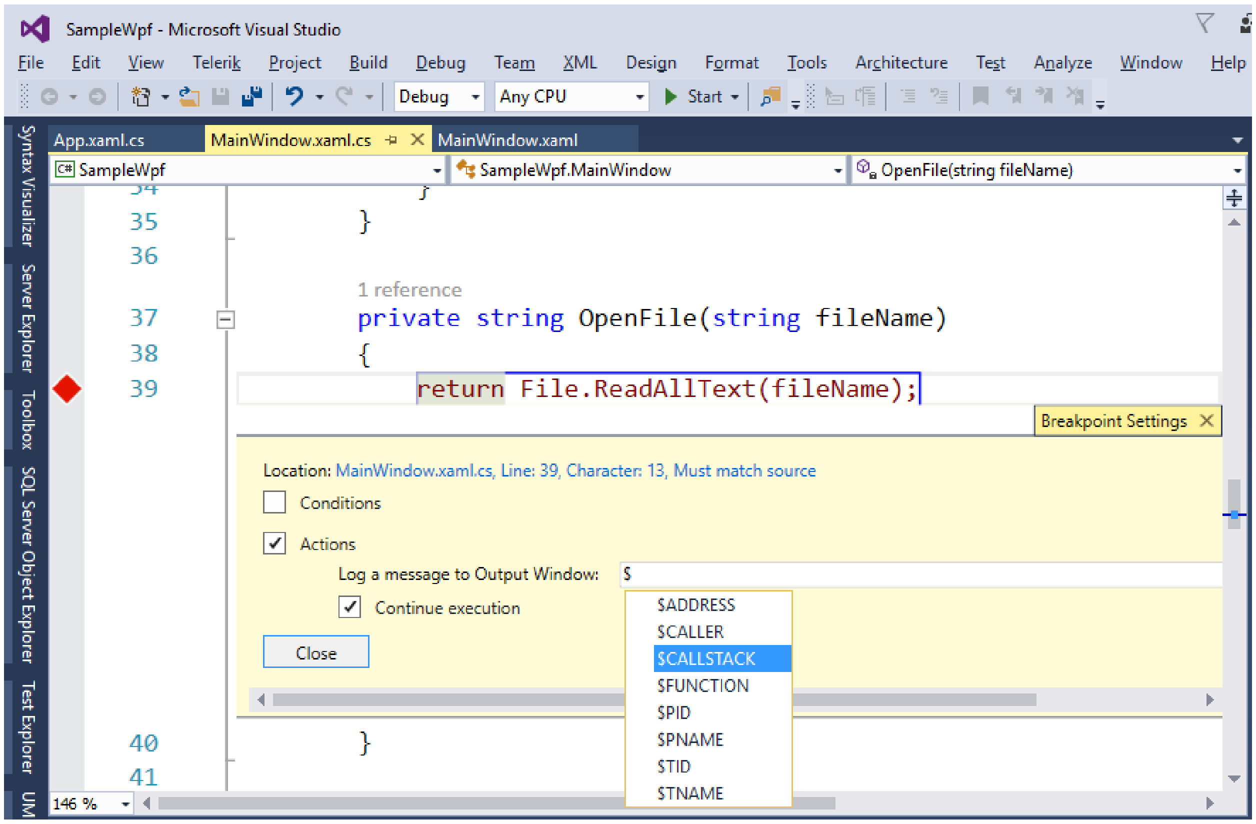
It is worth mentioning the availability of IntelliSense, which dramatically simplifies the way you write the expression to be evaluated. Now, let's talk about other conditions—Hit Count and Filter. Hit Count allows you to debug code from a certain point forward. Suppose you have a **`for`** loop with a breakpoint inside but you do not want the application execution to break at every iteration. With Hit Count, you can specify at which iteration the breakpoint must be hit and, consequently, the application execution must break. For example, if you enter Hit Count = 2, the breakpoint will be hit at the second iteration. Possible options are = (equal to), is a multiple of, and >= (greater than or equal to). With Filter, you can break the application execution only if the breakpoint is hit on the specified process, thread, or machine name.

Sending messages to the Output window with Actions

The Actions feature allows you to send log messages to the Output window using a built-in function.

Note: In previous editions of Visual Studio, this feature was known as Trace Points. It is now called Actions and is included in the new user interface for breakpoint settings.

To add an action, right-click the breakpoint's red icon and select **Actions**. You will see the same interactive pop-up described for conditions. In the Actions area, you can provide the name of a built-in function writing the $ symbol, then pick up the function name from the contextualized IntelliSense (see Figure 12).


SampleWpf - Microsoft Visual Studio
File Edit View Telerik Project Build Debug Team XML Design Format Tools Architecture Test Analyze Window Help
Debug
Any CPU
Start
App.xaml.cs
MainWindow.xaml.cs
MainWindow.xaml
SampleWpf
SampleWpf.MainWindow
OpenFile(string fileName)
Syntax Visualizer
Server Explorer
Toolbox
SQL Server Object Explorer
Test Explorer
146 %
Breakpoint Settings
Close

```
        }

        1 reference
        private string OpenFile(string fileName)
        {
            return File.ReadAllText(fileName);
```

Location: MainWindow.xaml.cs, Line: 39, Character: 13, Must match source

Conditions

[✓] Actions

Log a message to Output Window: $

[✓] Continue execution

$ADDRESS
$CALLER
$CALLSTACK
$FUNCTION
$PID
$PNAME
$TID
$TNAME

```
        }
```

Figure 12: Specifying an Action

For example, the **`$CALLSTACK`** function will send the call stack information to the Output window. Table 1 shows the supported functions.

Table 1: Supported Functions in the Actions Configuration

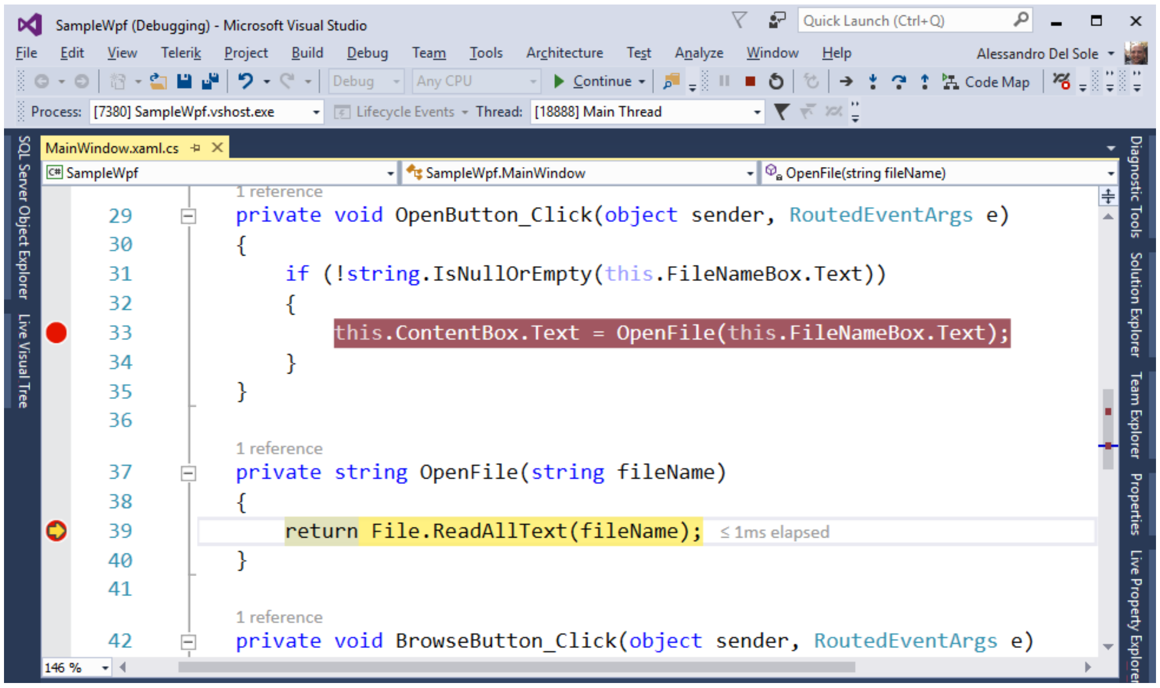
Breakpoint Actions	
$ADDRESS	The current instruction
$CALLER	The name of the caller function
$CALLSTACK	The method call stack

Breakpoint Actions	
$FUNCTION	The name of the current function
$PID	The process ID
$PNAME	The process name
$TID	The thread ID
$TNAME	The thread name

Custom expressions are also supported and documented at https://blogs.msdn.microsoft.com/visualstudioalm/2013/10/10/tracepoints/.

Introducing Performance Tips

Visual Studio 2015 introduced an interesting new feature to your debugging experience called Performance Tips (also referred to as **PerfTips**). With this feature, you can measure how long it takes to execute a section of code in break mode. In order to understand how it works, place two breakpoints in the sample application, one on the line of code that invokes the **`OpenFile`** method and assigns its result to the text box, and one on the return statement in the **`OpenFile`** method. Now start the application, select an existing file name with **Browse**, then click **Open**. At this point, the first breakpoint is hit. Press F5 to resume the execution. When the second breakpoint is hit, you will see a small tooltip near the highlighted line showing the number of milliseconds it took to execute the code between the two breakpoints, as shown in Figure 13.


SampleWpf (Debugging) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Telerik Project Build Debug Team Tools Architecture Test Analyze Window Help
Alessandro Del Sole
Debug Any CPU Continue Code Map
Process: [7380] SampleWpf.vshost.exe
Lifecycle Events
Thread: [18888] Main Thread
MainWindow.xaml.cs
SampleWpf
SampleWpf.MainWindow
OpenFile(string fileName)
SQL Server Object Explorer
Live Visual Tree
Diagnostic Tools
Solution Explorer
Team Explorer
Properties
Live Property Explorer
146 %

```
1 reference
private void OpenButton_Click(object sender, RoutedEventArgs e)
{
    if (!string.IsNullOrEmpty(this.FileNameBox.Text))
    {
        this.ContentBox.Text = OpenFile(this.FileNameBox.Text);
    }
}

1 reference
private string OpenFile(string fileName)
{
    return File.ReadAllText(fileName);  ≤ 1ms elapsed
}

1 reference
private void BrowseButton_Click(object sender, RoutedEventArgs e)
```

Figure 13: Measuring Code Execution Time with PerfTips

Note: PerfTips time values are approximate because they include debugging overhead. However, they are useful for establishing a good idea about the behavior of your code.

Chapter summary

Visual Studio 2015 and 2017 provide powerful debugging tools that you can use with any kind of .NET application and that you saw in action against a WPF project. With breakpoints, you highlight the lines of code you want to investigate, then you can step through your code by executing one line or a small set of lines at a time with commands such as Step Into, Step Over, and Step Out. But breakpoints offer more—you can specify conditions and actions to control when the application execution must break and what information you want to log to the Output window. The tools described in this chapter are typically available within the code editor, but the debugging experience in VS 2015 offers much more. In the next chapter, you will work with windows that allow deeper investigations into and evaluations of your code.

Chapter 3 Working with Debug Windows

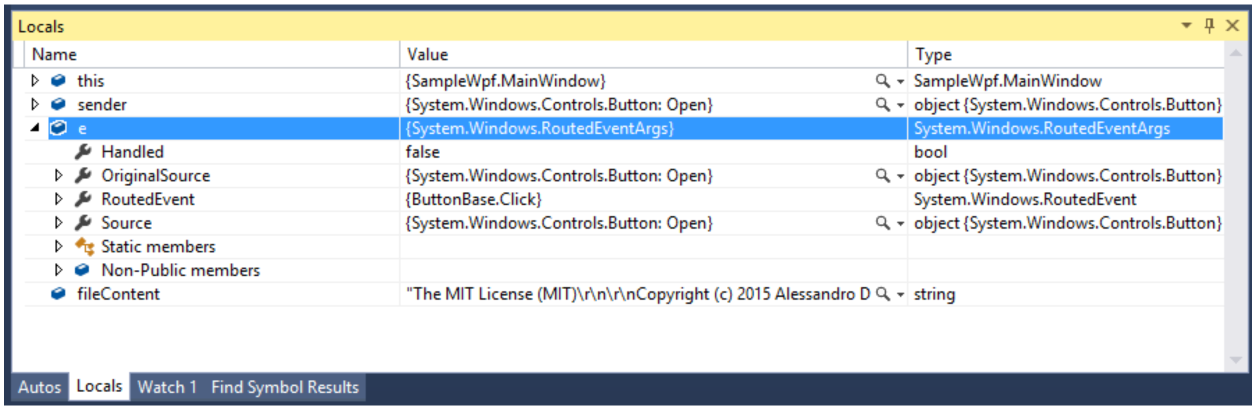
A WPF application is made of many building blocks, including the user interface, view models, and data, just to mention a few. For each, you might have methods, variables, expressions, and even multithreaded code. For this reason, debugging a WPF application can be difficult outside of an appropriate environment. Fortunately, Visual Studio provides you with many integrated tool windows that simplify debugging and cover a huge number of scenarios. This chapter will provide guidance about the most commonly used debugging windows you will need with a WPF application, and it will offer information about writing better code.

Investigating local variables with the Locals window

The Locals window is a useful debugging window because it allows you to show the active local variables and their values. If the local variable represents a composite type, such as a class, you will be also able to see the type's property and field values. In order to see it in action, consider the sample application created in Chapter 1 and place a breakpoint on the following line, inside the **`OpenButton_Click`** event handler:

```
this.ContentBox.Text = fileContent;
```

Run the application, browse for a file name, then click **Open**. When the breakpoint is hit and Visual Studio enters the break mode, select the **Locals** window, which should be enabled by default. If not, select Debug, Windows, Locals. As you can see in Figure 14, the Locals window shows a list of active local variables. For each variable, it shows the value and the type.



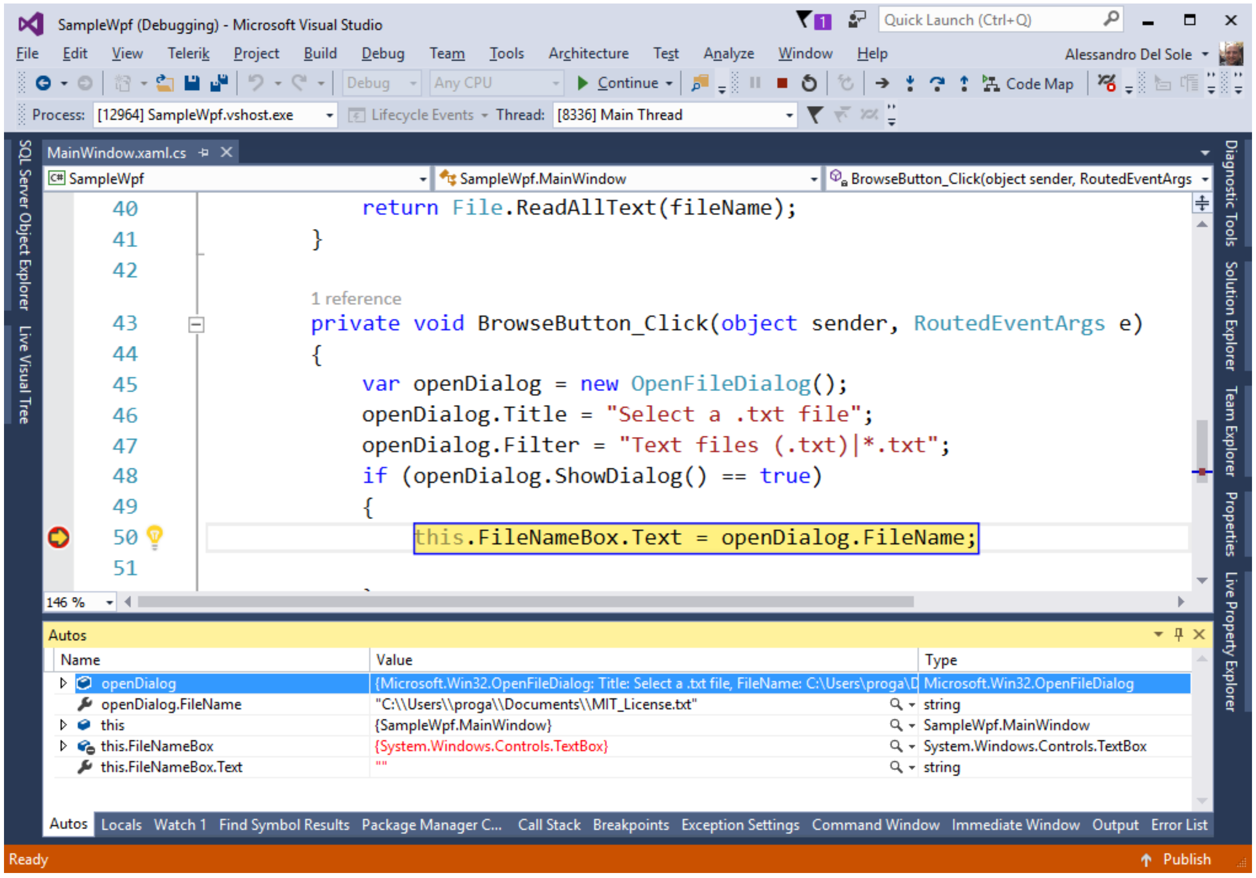
Name	Value	Type
this	{SampleWpf.MainWindow}	SampleWpf.MainWindow
sender	{System.Windows.Controls.Button: Open}	object {System.Windows.Controls.Button}
e	{System.Windows.RoutedEventArgs}	System.Windows.RoutedEventArgs
Handled	false	bool
OriginalSource	{System.Windows.Controls.Button: Open}	object {System.Windows.Controls.Button}
RoutedEvent	{ButtonBase.Click}	System.Windows.RoutedEvent
Source	{System.Windows.Controls.Button: Open}	object {System.Windows.Controls.Button}
Static members		
Non-Public members		
fileContent	"The MIT License (MIT)\r\n\r\nCopyright (c) 2015 Alessandro D	string

Figure 14: Investigating Local Variables with the Locals Window

If the variable type is a primitive type, such as a string, the Value column immediately shows the variable's value, as for the **`fileContent`** local variable, of type **`string`**. If the variable type is a composite type, such as the **`e`** variable of type **`System.Windows.RoutedEventArgs`**, you can expand the variable and see its members. For each member, you will be able to see the type and current value; again, you can expand the member if it is a composite type—for example, expanding the **`e`** variable will cause the Locals window to show all the properties and fields (with values) exposed by the **`System.Windows.RoutedEventArgs`** type. The Locals window can be a real lifesaver when you need to check if a variable stores an expected (or unexpected) value.

Investigating current variables with the Autos window

You can also investigate variables with the Autos window, which shows the variables in three forms: used by the current statement, used by the previous three statements, and used by following three statements. You can also change a variable's value with a simple double-click. In Figure 15, you can see how a variable is presented in red if it is related to a current breakpoint.


SampleWpf (Debugging) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Telerik Project Build Debug Team Tools Architecture Test Analyze Window Help
Alessandro Del Sole
Debug
Any CPU
Continue
Code Map
Process: [12964] SampleWpf.vshost.exe
Lifecycle Events
Thread: [8336] Main Thread
SQL Server Object Explorer
Live Visual Tree
MainWindow.xaml.cs
SampleWpf
SampleWpf.MainWindow
BrowseButton_Click(object sender, RoutedEventArgs
Diagnostic Tools
Solution Explorer
Team Explorer
Properties
Live Property Explorer
146 %

```
            return File.ReadAllText(fileName);
        }

        1 reference
        private void BrowseButton_Click(object sender, RoutedEventArgs e)
        {
            var openDialog = new OpenFileDialog();
            openDialog.Title = "Select a .txt file";
            openDialog.Filter = "Text files (.txt)|*.txt";
            if (openDialog.ShowDialog() == true)
            {
                this.FileNameBox.Text = openDialog.FileName;
```

Autos

Name	Value	Type
openDialog	{Microsoft.Win32.OpenFileDialog: Title: Select a .txt file, FileName: C:\Users\proga\D	Microsoft.Win32.OpenFileDialog
openDialog.FileName	"C:\\Users\\proga\\Documents\\MIT_License.txt"	string
this	{SampleWpf.MainWindow}	SampleWpf.MainWindow
this.FileNameBox	{System.Windows.Controls.TextBox}	System.Windows.Controls.TextBox
this.FileNameBox.Text	""	string

Autos Locals Watch 1 Find Symbol Results Package Manager C... Call Stack Breakpoints Exception Settings Command Window Immediate Window Output Error List
Ready
Publish

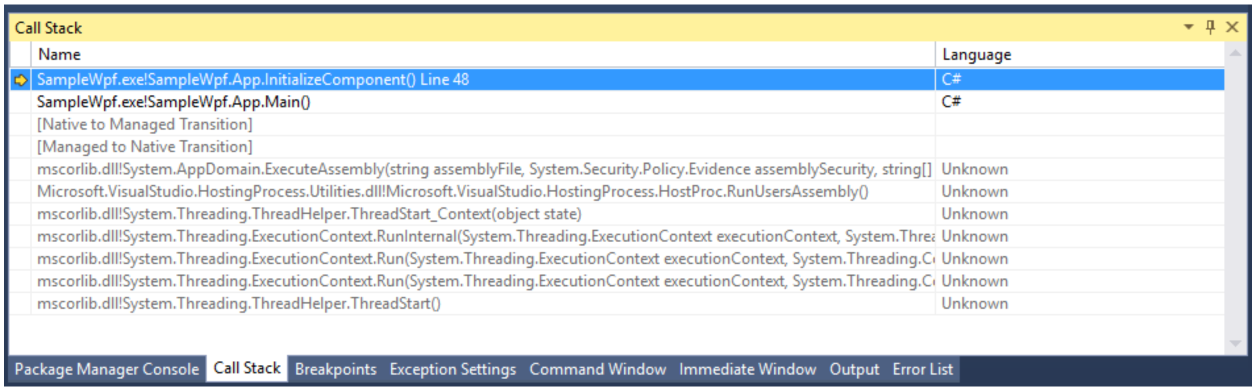
Figure 15: Investigating Variables with the Autos Window

Analyzing method calls: the Call Stack window

The Call Stack window shows how method calls run in the stack, and it is useful for understanding the method call hierarchy.

Tip: If Just My Code is enabled, the Call Stack window shows a limited set of information. For a better understanding of Call Stack, disable Just My Code. You can then decide to re-enable it after reading about Call Stack.

Call Stack goes live when you debug your code—for example, when you press F11. Figure 16 shows the window in action.



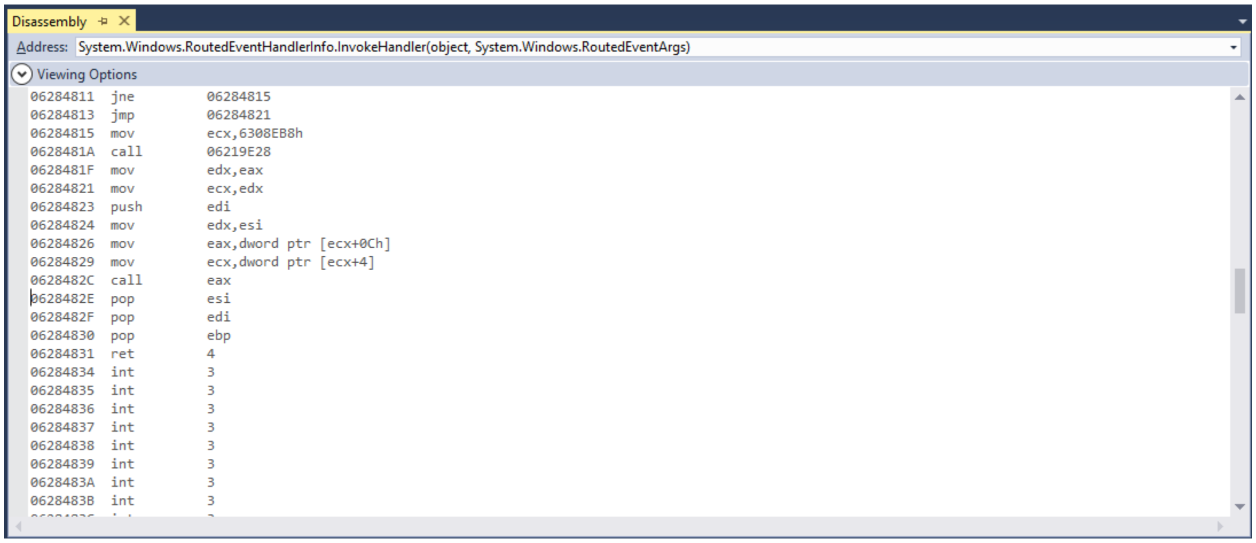
Name	Language
SampleWpf.exe!SampleWpf.App.InitializeComponent() Line 48	C#
SampleWpf.exe!SampleWpf.App.Main()	C#
[Native to Managed Transition]	
[Managed to Native Transition]	
mscorlib.dll!System.AppDomain.ExecuteAssembly(string assemblyFile, System.Security.Policy.Evidence assemblySecurity, string[]	Unknown
Microsoft.VisualStudio.HostingProcess.Utilities.dll!Microsoft.VisualStudio.HostingProcess.HostProc.RunUsersAssembly()	Unknown
mscorlib.dll!System.Threading.ThreadHelper.ThreadStart_Context(object state)	Unknown
mscorlib.dll!System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext executionContext, System.Threa	Unknown
mscorlib.dll!System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext executionContext, System.Threading.C	Unknown
mscorlib.dll!System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext executionContext, System.Threading.C	Unknown
mscorlib.dll!System.Threading.ThreadHelper.ThreadStart()	Unknown

Figure 16: Investigating Method Calls with Call Stack

Call Stack shows:

- The list of method names being executed and the programming language with which each is written.
- The list of calls to .NET system methods.
- Method calls to/from other threads.

If the source code for a given method is not available, you can still view the assembly code by right-clicking a method name, then selecting **Go to Disassembly**. Figure 17 shows an example.


Disassembly
Address: System.Windows.RoutedEventHandlerInfo.InvokeHandler(object, System.Windows.RoutedEventArgs)
Viewing Options

```
06284811  jne        06284815
06284813  jmp        06284821
06284815  mov        ecx,6308EB8h
0628481A  call       06219E28
0628481F  mov        edx,eax
06284821  mov        ecx,edx
06284823  push       edi
06284824  mov        edx,esi
06284826  mov        eax,dword ptr [ecx+0Ch]
06284829  mov        ecx,dword ptr [ecx+4]
0628482C  call       eax
0628482E  pop        esi
0628482F  pop        edi
06284830  pop        ebp
06284831  ret        4
06284834  int        3
06284835  int        3
06284836  int        3
06284837  int        3
06284838  int        3
06284839  int        3
0628483A  int        3
0628483B  int        3
```

Figure 17: Disassembling a Method Call

Notice how the Disassembly window shows not only the disassembled method call, but also the address in memory of each instruction. You can also customize the appearance of Call Stack by right-clicking a column header and selecting the available options from the context menu, such as (but not limited to) parameter values and hexadecimal display. Call Stack is particularly useful when you get into an issue that is not apparently caused by the current piece of code and that needs further investigation through the entire stack of method calls.

Evaluating expressions: Watch and Quick Watch windows

You can track a variable by monitoring an object or expression using the Watch and Quick Watch windows. The difference is that Watch can monitor multiple variables, whereas Quick Watch can monitor one variable per time. There are four Watch windows available, which means you can monitor a large number of variables. In order to understand how they work, let's make a slight modification to the **`OpenButton_Click`** event handler in the sample application, as shown in Code Listing 3.

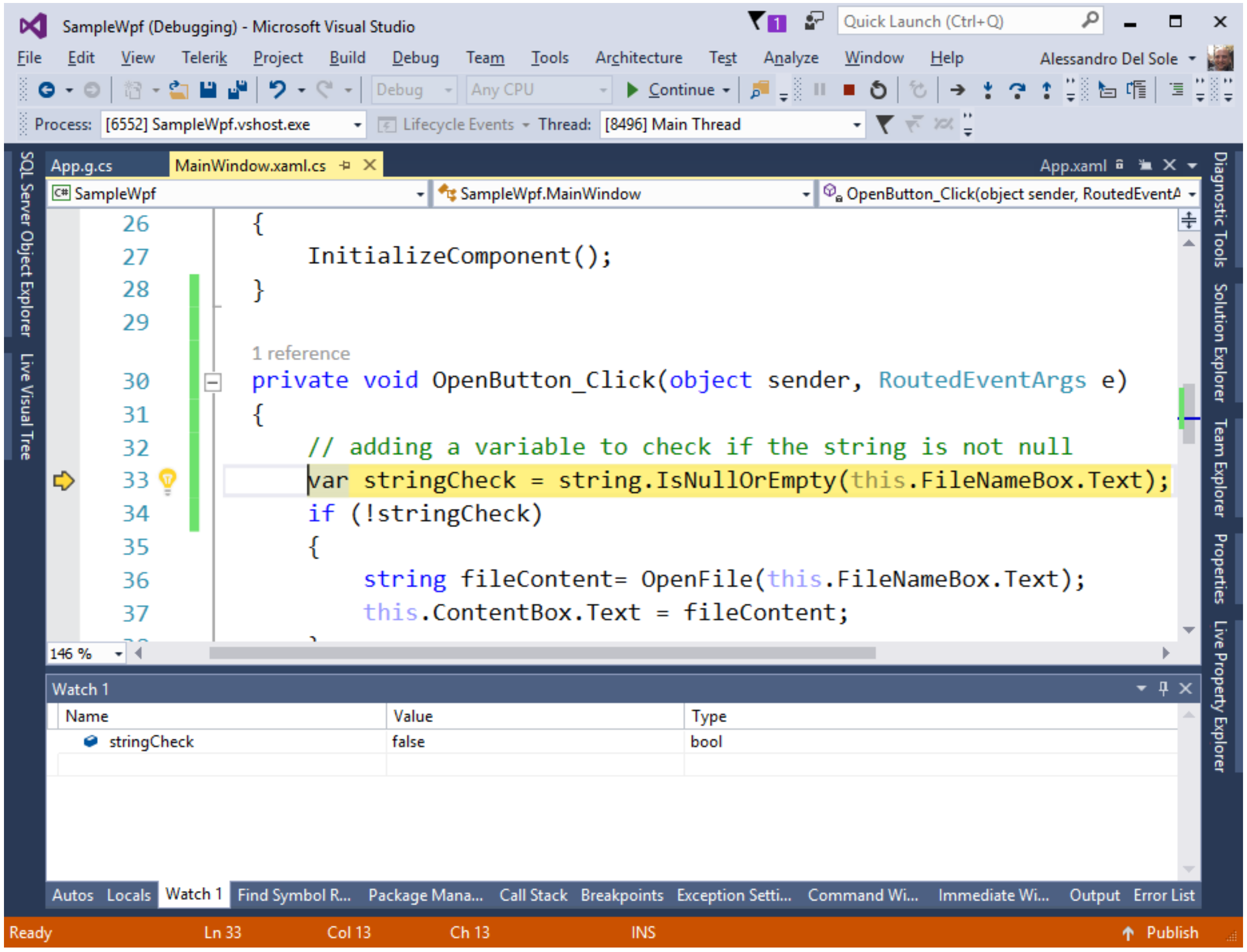
Code Listing 3

```
private void OpenButton_Click(object sender, RoutedEventArgs e)
{
    // Adding a variable to check if the string is not null.
    bool stringCheck =
        string.IsNullOrEmpty(this.FileNameBox.Text);
    if (!stringCheck)
    {
        string fileContent= OpenFile(this.FileNameBox.Text);
        this.ContentBox.Text = fileContent;
    }
}
```

Our goal is to monitor the behavior of the **`stringCheck`** variable, which we do in break mode. Next, press F11 to start the application instead of F5. Then we locate the stringCheck variable, right-click, and select **Add Watch**. At this point, the Watch 1 window will show the variable and display a message saying it does not exist in the current context, which is to be expected because the debugger has not yet been entered into the event handler.

Tip: Generally speaking, selecting Add Watch will open the first Watch window available among the four VS offers.

If you step into the code and enter into the **`OpenButton_Click`** handler, the Watch window will show the current evaluation for the expression assigned to the variable, which is False, as you can see in Figure 18.



```
{
    InitializeComponent();
}

1 reference
private void OpenButton_Click(object sender, RoutedEventArgs e)
{
    // adding a variable to check if the string is not null
    var stringCheck = string.IsNullOrEmpty(this.FileNameBox.Text);
    if (!stringCheck)
    {
        string fileContent= OpenFile(this.FileNameBox.Text);
        this.ContentBox.Text = fileContent;
```

Name	Value	Type
stringCheck	false	bool

SampleWpf (Debugging) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Telerik Project Build Debug Team Tools Architecture Test Analyze Window Help
Alessandro Del Sole
Debug
Any CPU
Continue
Process: [6552] SampleWpf.vshost.exe
Lifecycle Events
Thread: [8496] Main Thread
App.g.cs
MainWindow.xaml.cs
App.xaml
SampleWpf
SampleWpf.MainWindow
OpenButton_Click(object sender, RoutedEventA
146 %
Watch 1
SQL Server Object Explorer
Live Visual Tree
Diagnostic Tools
Solution Explorer
Team Explorer
Properties
Live Property Explorer
Autos Locals Watch 1 Find Symbol R... Package Mana... Call Stack Breakpoints Exception Setti... Command Wi... Immediate Wi... Output Error List
Ready Ln 33 Col 13 Ch 13 INS
Publish

Figure 18: Monitoring an Expression with the Watch Window

As you continue stepping through the code, you'll see how the Watch window will show the result of the evaluation of the expression assigned to the **`stringCheck`** variable, which might or might not be **`true`**. The Quick Watch window works similarly, but it works on a single variable and is offered through a modal dialog.

Debugging lambda expressions

Visual Studio 2015 has introduced an important feature that allows for debugging lambda expressions and LINQ queries inside the Watch window. In order to understand how to leverage this new tool, let's make a couple modifications to the sample project. First, we'll add a new button to the user interface as follows:

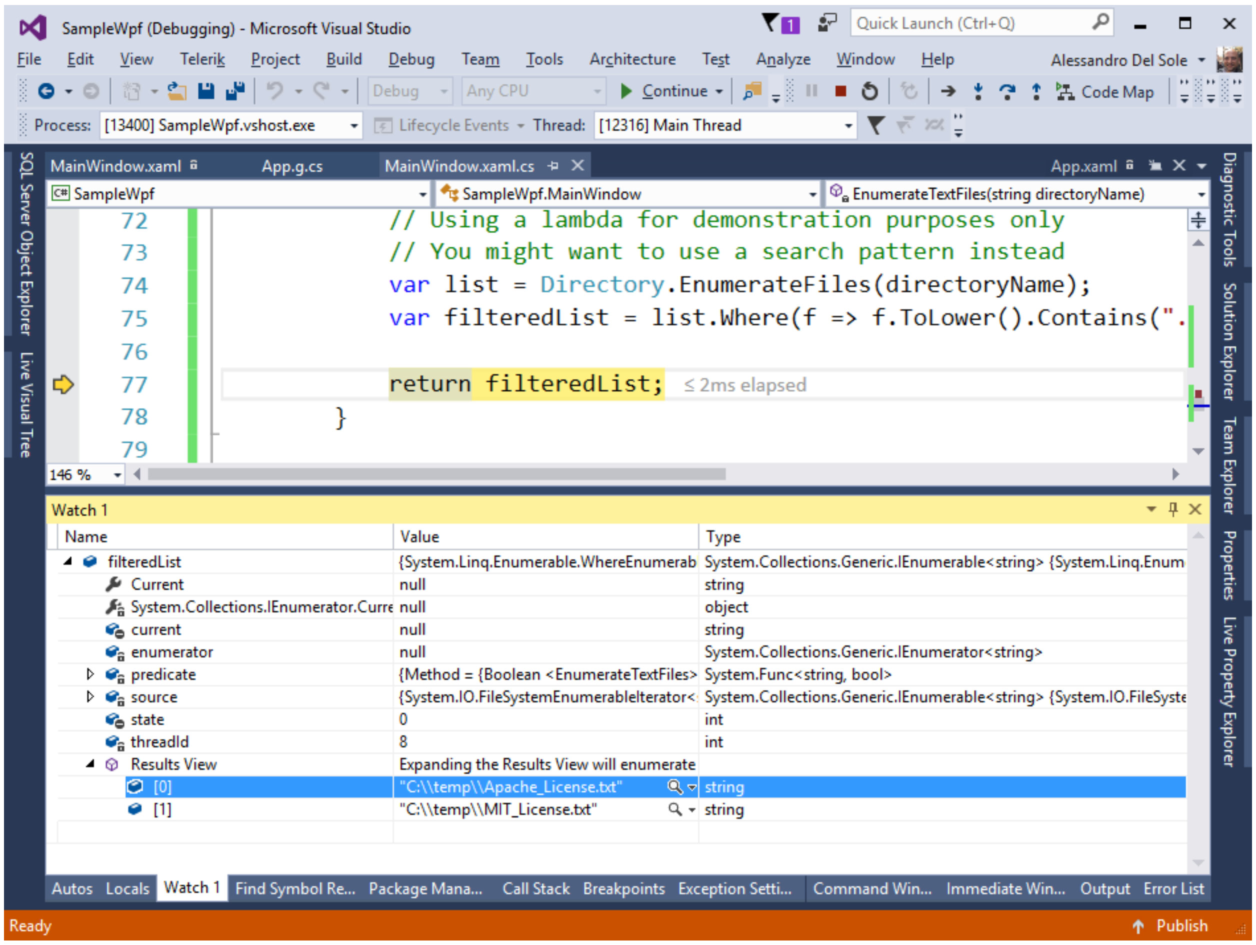
```
<Button Width="100" Height="30" Content="Feeling lucky"
        x:Name="FeelingLuckyButton" Click="FeelingLuckyButton_Click"/>
```

This new button's purpose is to launch a method that will open the first .txt document inside a given folder, then add the event handler and method shown in Code Listing 4.

Code Listing 4

```
        private IEnumerable<string>
                EnumerateTextFiles(string directoryName)
        {
            // Using a lambda for demonstration purposes only.
            // You might want to use a search pattern instead.
            var list = Directory.EnumerateFiles(directoryName);
            var filteredList = list.Where(f =>
                f.ToLower().Contains(".txt"));

            return filteredList;
        }

        private void FeelingLuckyButton_Click(object sender,
                RoutedEventArgs e)
        {
            this.FileNameBox.Text =
            OpenFile(EnumerateTextFiles("C:\\temp").FirstOrDefault());
        }
```

The **`EnumerateTextFiles`** method is using a lambda expression to filter the list of files in the specified folder. In your real code, you might prefer specifying the search pattern parameter for the **`Directory.EnumerateFiles`** method, but using a lambda is necessary in order to demonstrate how to use the new debugging features. At this point, place a breakpoint on the **`EnumerateTextFiles`** method, then start debugging. When the debugger enters this method, right-click the **`filteredList`** variable, then select **Add Watch**. When you press F11 over this variable, the Watch window will evaluate the expression, which includes expanding the Results View element that shows the result of the lambda. Figure 19 demonstrates this.


SampleWpf (Debugging) - Microsoft Visual Studio
File Edit View Telerik Project Build Debug Team Tools Architecture Test Analyze Window Help
Continue
Code Map
Process: [13400] SampleWpf.vshost.exe
Lifecycle Events
Thread: [12316] Main Thread
MainWindow.xaml App.g.cs MainWindow.xaml.cs App.xaml
SampleWpf
SampleWpf.MainWindow
EnumerateTextFiles(string directoryName)
≤ 2ms elapsed
146 %
Watch 1
Autos Locals Watch 1 Find Symbol Re... Package Mana... Call Stack Breakpoints Exception Setti... Command Win... Immediate Win... Output Error List
Ready
Publish

```
            // Using a lambda for demonstration purposes only
            // You might want to use a search pattern instead
            var list = Directory.EnumerateFiles(directoryName);
            var filteredList = list.Where(f => f.ToLower().Contains(".

            return filteredList;
        }
```

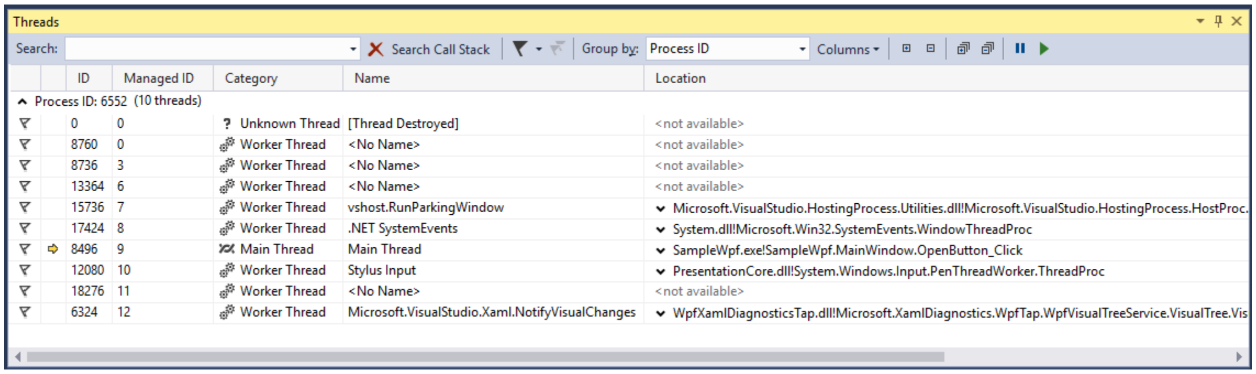
Name	Value	Type
filteredList	{System.Linq.Enumerable.WhereEnumerab	System.Collections.Generic.IEnumerable<string> {System.Linq.Enum
Current	null	string
System.Collections.IEnumerator.Curre	null	object
current	null	string
enumerator	null	System.Collections.Generic.IEnumerator<string>
predicate	{Method = {Boolean <EnumerateTextFiles>	System.Func<string, bool>
source	{System.IO.FileSystemEnumerableIterator<	System.Collections.Generic.IEnumerable<string> {System.IO.FileSyste
state	0	int
threadId	8	int
Results View	Expanding the Results View will enumerate	
[0]	"C:\\temp\\Apache_License.txt"	string
[1]	"C:\\temp\\MIT_License.txt"	string

Figure 19: Debugging a Lambda Expression

In this way, you can investigate the expression result and see if it is working as expected.

Debugging threads: the Threads window

More often than not, a .NET application can run multiple threads. Sometimes multiple threads are run in order to suit how the system manages an application or because we have programmatically created new threads for delegating tasks to separate units of execution. Visual Studio offers the Threads window (Ctrl+Alt+H), in which you can see the list of running threads (see Figure 20).



Process ID: 6552 (10 threads)

ID	Managed ID	Category	Name	Location
0	0	Unknown Thread	[Thread Destroyed]	<not available>
8760	0	Worker Thread	<No Name>	<not available>
8736	3	Worker Thread	<No Name>	<not available>
13364	6	Worker Thread	<No Name>	<not available>
15736	7	Worker Thread	vshost.RunParkingWindow	Microsoft.VisualStudio.HostingProcess.Utilities.dll!Microsoft.VisualStudio.HostingProcess.HostProc.
17424	8	Worker Thread	.NET SystemEvents	System.dll!Microsoft.Win32.SystemEvents.WindowThreadProc
8496	9	Main Thread	Main Thread	SampleWpf.exe!SampleWpf.MainWindow.OpenButton_Click
12080	10	Worker Thread	Stylus Input	PresentationCore.dll!System.Windows.Input.PenThreadWorker.ThreadProc
18276	11	Worker Thread	<No Name>	<not available>
6324	12	Worker Thread	Microsoft.VisualStudio.Xaml.NotifyVisualChanges	WpfXamlDiagnosticsTap.dll!Microsoft.XamlDiagnostics.WpfTap.WpfVisualTreeService.VisualTree.Vis

Figure 20: Watching the List of Running Threads

For each thread, you can see the name, the ID, the category, and the location. With the category, you can see how the main thread corresponds to the user interface thread in the application. This is demonstrated in the Location column, where you not only find the name of the module, but also the method that is currently being executed. In the case of threads created programmatically, they will be listed in the window and you will be able to get information about them. The most interesting feature of the Threads window is that you can also select a thread, then click **Search Call Stack**. This action will open the Call Stack window, thereby showing the method calls in the specified thread.

Asynchronous debugging: the Tasks window

Visual Studio offers the Tasks window, a convenient way of investigating tasks that the runtime creates when you code asynchronous methods using the Async/Await pattern. In order to understand how this works, let's add a new asynchronous method to the sample application that opens a text file based on the Async/Await pattern. This is shown in Code Listing 5.

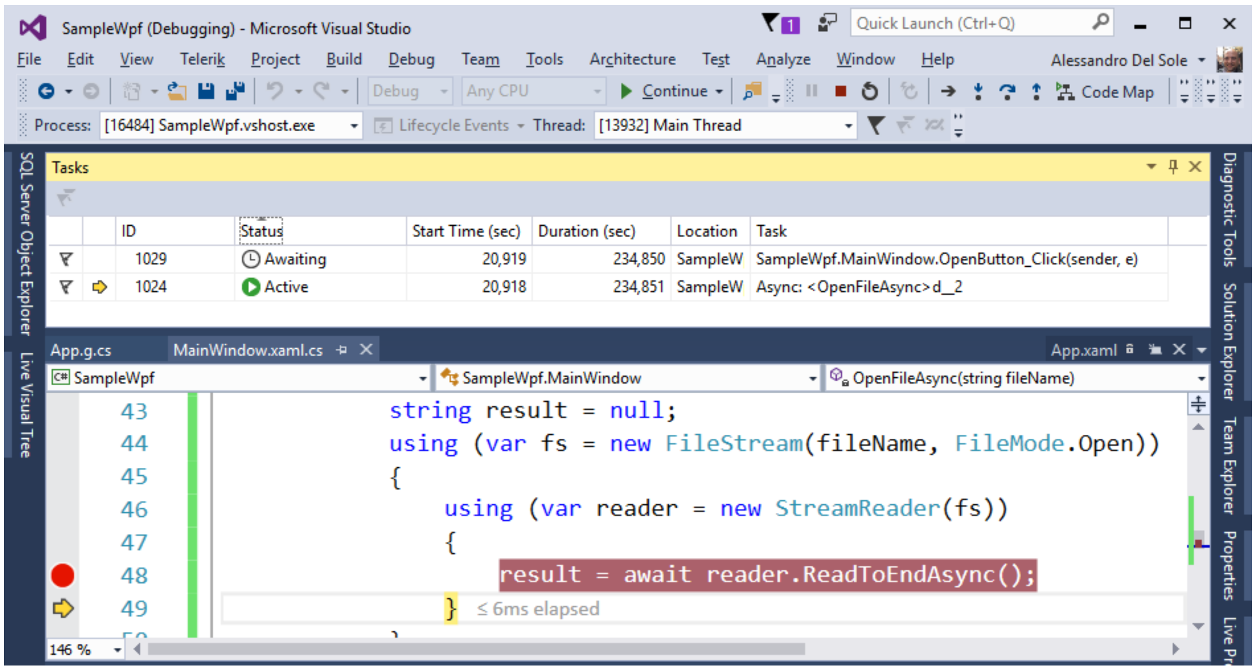
Code Listing 5

```
        private async Task<string> OpenFileAsync(string fileName)
        {
            string result = null;
            using (var fs = new FileStream(fileName, FileMode.Open))
            {
                using (var reader = new StreamReader(fs))
                {
                    result = await reader.ReadToEndAsync();
                }
            }
            return result;
        }
```

In order to use this method, you will need to mark the **`OpenButton_Click`** event handler with the **`async`** modifier, and you will need to change the assignment of the **`fileContent`** variable as follows:

```
string fileContent = await OpenFileAsync(this.FileNameBox.Text);
```

If you place a breakpoint on the **`OpenFileAsync`** method and enable the Tasks window (Ctrl+Shift+D, K) while in break mode, you will see the status of a task, its time of execution, and its location (see Figure 21).


SampleWpf (Debugging) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Telerik Project Build Debug Team Tools Architecture Test Analyze Window Help
Alessandro Del Sole
Debug Any CPU Continue Code Map
Process: [16484] SampleWpf.vshost.exe Lifecycle Events Thread: [13932] Main Thread
Tasks
App.g.cs MainWindow.xaml.cs App.xaml
SampleWpf SampleWpf.MainWindow OpenFileAsync(string fileName)
≤ 6ms elapsed
146 %
SQL Server Object Explorer Live Visual Tree
Diagnostic Tools Solution Explorer Team Explorer Properties Live Pr

ID	Status	Start Time (sec)	Duration (sec)	Location	Task
1029	Awaiting	20,919	234,850	SampleW	SampleWpf.MainWindow.OpenButton_Click(sender, e)
1024	Active	20,918	234,851	SampleW	Async: <OpenFileAsync>d__2

```
string result = null;
using (var fs = new FileStream(fileName, FileMode.Open))
{
    using (var reader = new StreamReader(fs))
    {
        result = await reader.ReadToEndAsync();
    }
```

Figure 21: Monitoring Asynchronous Tasks

Note: The Tasks window can also be used with parallel programming based on the Task Parallel Library (TPL) for investigating the execution of parallel tasks.

Chapter summary

The more complex the WPF application, the more you will need sophisticated debugging tools. Visual Studio offers everything you need in order to perform complex debugging over your apps. In fact, the IDE provides many debugging windows that you can use to inspect variables (Locals and Autos windows), evaluate expressions (Watch windows), investigate method calls (Call Stack window), and analyze threads and tasks (Threads and Tasks windows). All of these windows are nonmodal, they work docked inside the IDE and, therefore, you can always keep an eye on your code, which means you can be more productive.

Chapter 4 Debugger Visualizers and Trace Listeners

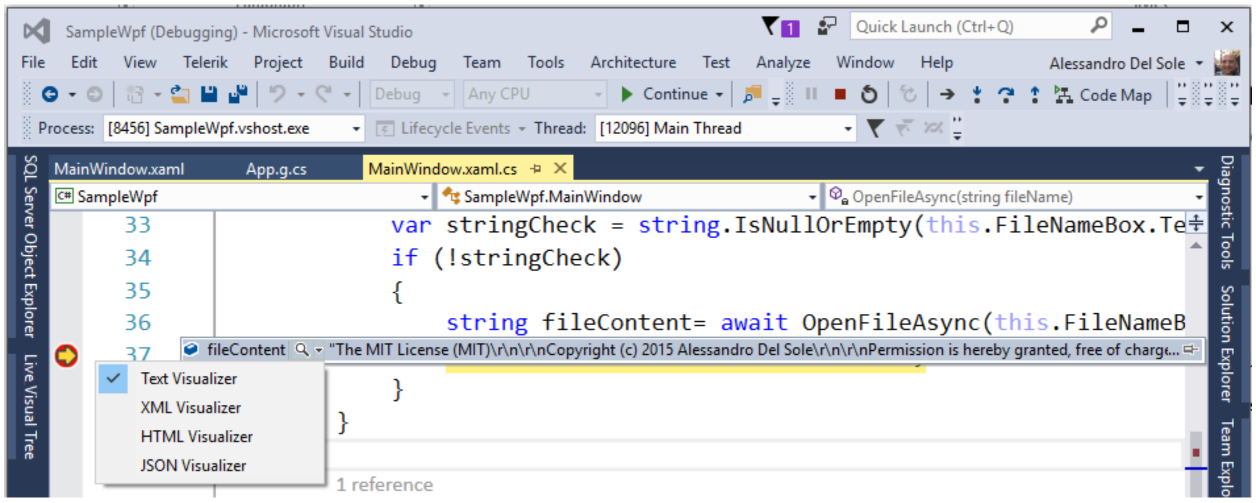
Debugging not only means searching for and fixing errors, it also means collecting diagnostic information that helps developers understand how the application behaves at specific points during its lifetime. This is even more important in WPF applications because they can work with a large range of resources, such as graphics, media, and data. This chapter will explain how to collect information using debugger visualizers and trace listeners.

Introducing debugger visualizers

In some situations, you might have objects, controls, or variables that store data in a particular format. For example, you might have a string representing XML or JSON data. While debugging, you might need a way to investigate this kind of information with an appropriate view. Visual Studio makes this possible with debugger visualizers. For a better understanding, in the sample application place a breakpoint on the following line, which is located in the **`OpenButton_Click`** event handler:

```
this.ContentBox.Text = fileContent;
```

When the breakpoint is hit, if you hover over the **`fileContent`** variable you will see a data tip that shows the variable's value. The data tip also provides a small magnifying glass icon that you can click to pick one of the available visualizers (see Figure 22).



```
            var stringCheck = string.IsNullOrEmpty(this.FileNameBox.Te
            if (!stringCheck)
            {
                string fileContent= await OpenFileAsync(this.FileNameB
```

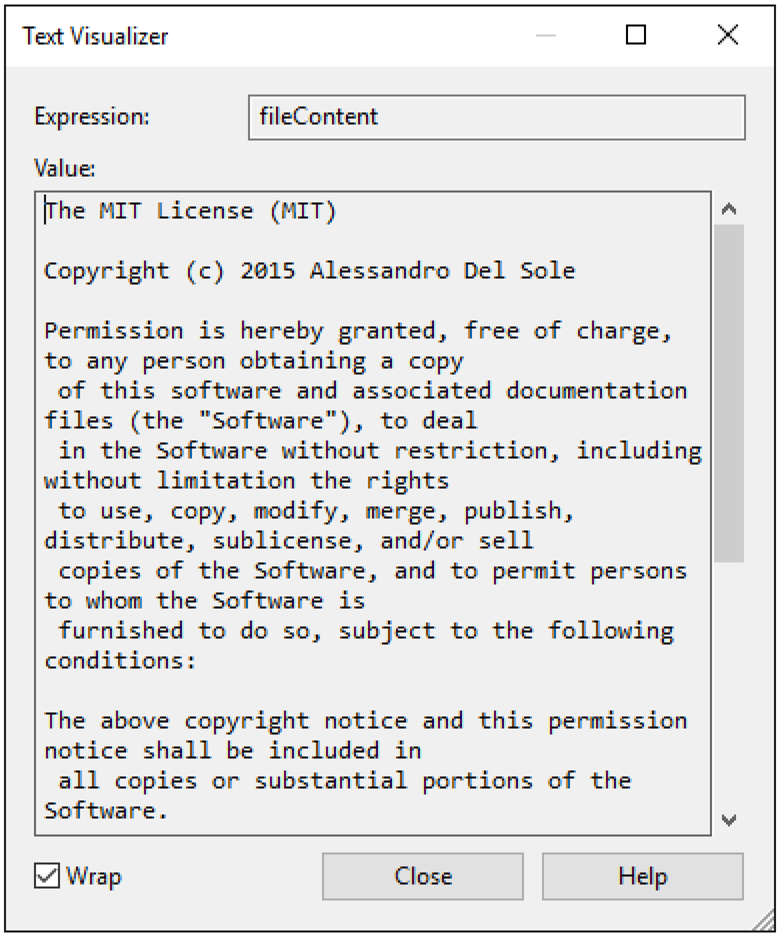
fileContent "The MIT License (MIT)\r\n\r\nCopyright (c) 2015 Alessandro Del Sole\r\n\r\nPermission is hereby granted, free of charge...

```
            }
        }
```

1 reference

Figure 22: Selecting a Debugger Visualizer

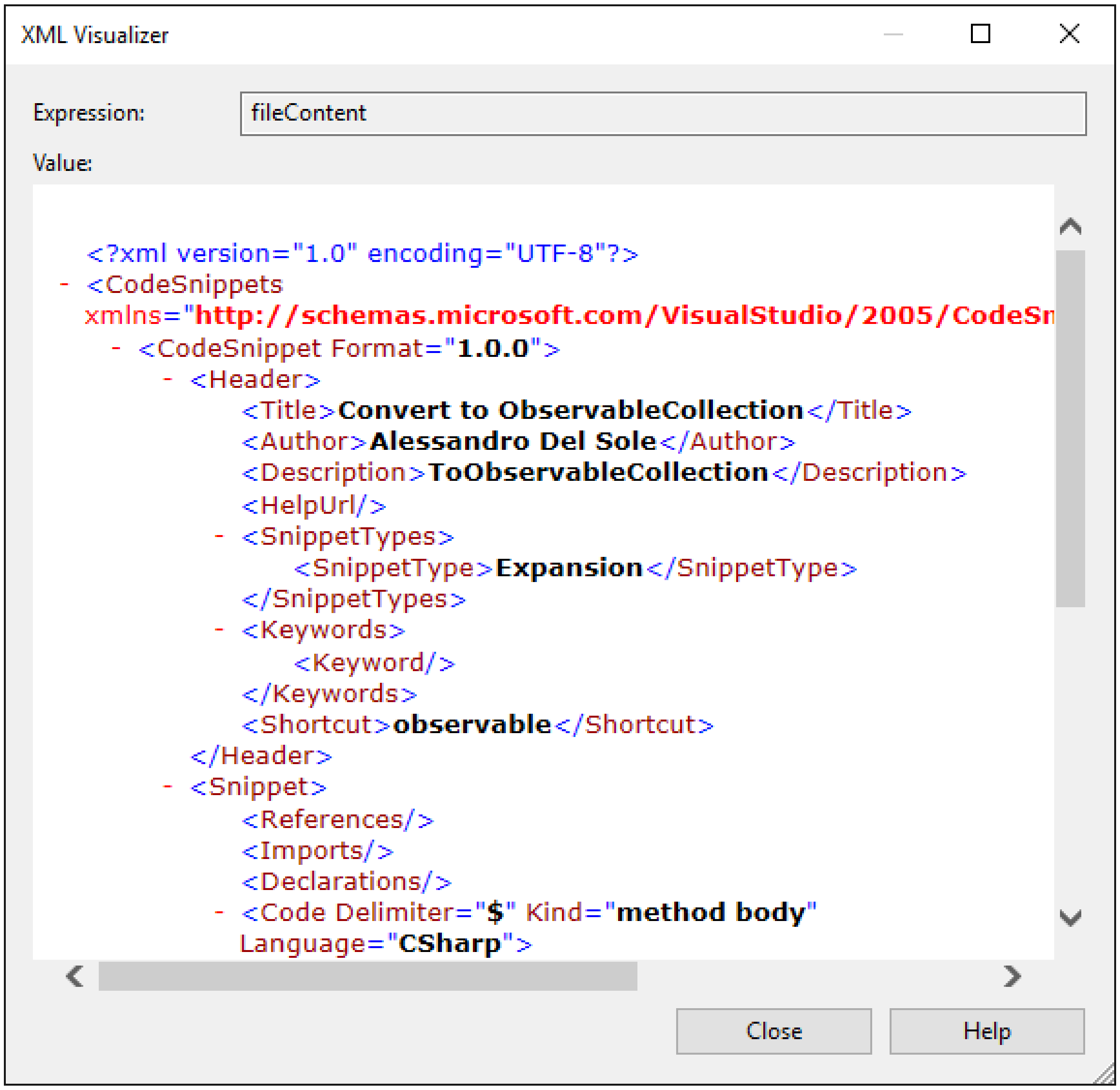
Depending on the format of your information, you can pick the most appropriate visualizer. In this case, the string contains plain text, which means that selecting the Text Visualizer is the proper choice. Figure 23 shows the Text Visualizer in action.



```
The MIT License (MIT)

Copyright (c) 2015 Alessandro Del Sole

Permission is hereby granted, free of charge,
to any person obtaining a copy
 of this software and associated documentation
files (the "Software"), to deal
 in the Software without restriction, including
without limitation the rights
 to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell
 copies of the Software, and to permit persons
to whom the Software is
 furnished to do so, subject to the following
conditions:

The above copyright notice and this permission
notice shall be included in
 all copies or substantial portions of the
Software.
```

Figure 23: Text Visualizer Allows Viewing Plain Text

Now suppose your variable contains XML data. You could select the XML Visualizer to get a structured view of your file instead of plain text. Figure 24 shows an example.



```
  <?xml version="1.0" encoding="UTF-8"?>
- <CodeSnippets
  xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSn
  - <CodeSnippet Format="1.0.0">
    - <Header>
        <Title>Convert to ObservableCollection</Title>
        <Author>Alessandro Del Sole</Author>
        <Description>ToObservableCollection</Description>
        <HelpUrl/>
      - <SnippetTypes>
          <SnippetType>Expansion</SnippetType>
        </SnippetTypes>
      - <Keywords>
          <Keyword/>
        </Keywords>
        <Shortcut>observable</Shortcut>
      </Header>
    - <Snippet>
        <References/>
        <Imports/>
        <Declarations/>
      - <Code Delimiter="$" Kind="method body"
        Language="CSharp">
```

Figure 24: XML Visualizer Allows Viewing Structured XML Contents

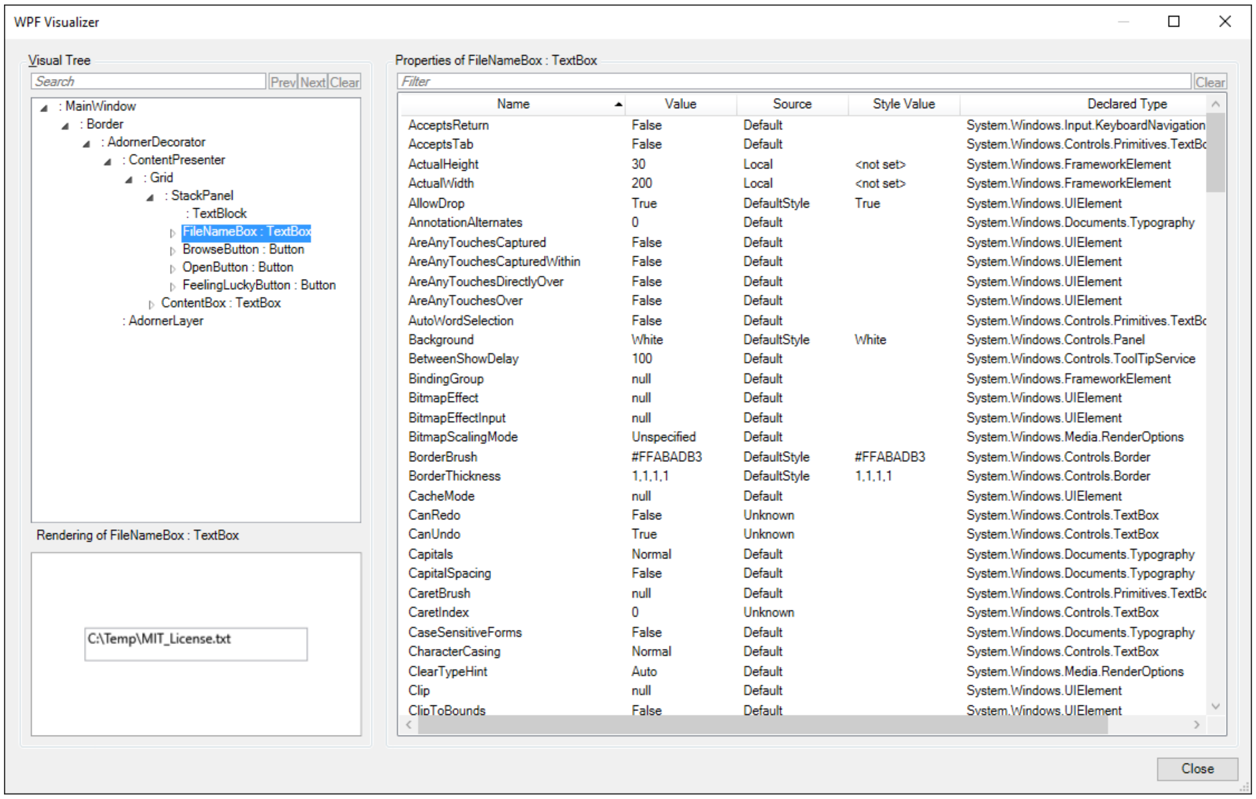
As with the XML Visualizer, the JSON and HTML Visualizers help you investigate formatted contents. Debugger visualizers are very useful, especially when you have long and complex information you need to view and the data tips are not enough.

The WPF Tree Visualizer

WPF has a special debugger visualizer called WPF Tree Visualizer, which allows you to inspect values of controls' properties at runtime.

Note: This visualizer is discussed for consistency with the WPF platform, but you might prefer the Live Property Explorer and Live Visual Tree windows described in Chapter 5 XAML Debugging.

This visualizer's goal is to provide a hierarchical view of the visual tree of the current window or user control. You enable it by hovering over a control's name in the code editor while in break mode, then clicking the magnifier icon that appears on the data tip. Figure 25 shows an example based on the **`FileNameBox`** control in the sample application.


WPF Visualizer
Visual Tree
Search
Prev
Next
Clear
Properties of FileNameBox : TextBox
Filter
Clear
Rendering of FileNameBox : TextBox
Close

```
: MainWindow
  : Border
    : AdornerDecorator
      : ContentPresenter
        : Grid
          : StackPanel
              : TextBlock
            FileNameBox : TextBox
            BrowseButton : Button
            OpenButton : Button
            FeelingLuckyButton : Button
          ContentBox : TextBox
      : AdornerLayer
```

C:\Temp\MIT_License.txt

Name	Value	Source	Style Value	Declared Type
AcceptsReturn	False	Default		System.Windows.Input.KeyboardNavigation
AcceptsTab	False	Default		System.Windows.Controls.Primitives.TextBc
ActualHeight	30	Local	<not set>	System.Windows.FrameworkElement
ActualWidth	200	Local	<not set>	System.Windows.FrameworkElement
AllowDrop	True	DefaultStyle	True	System.Windows.UIElement
AnnotationAlternates	0	Default		System.Windows.Documents.Typography
AreAnyTouchesCaptured	False	Default		System.Windows.UIElement
AreAnyTouchesCapturedWithin	False	Default		System.Windows.UIElement
AreAnyTouchesDirectlyOver	False	Default		System.Windows.UIElement
AreAnyTouchesOver	False	Default		System.Windows.UIElement
AutoWordSelection	False	Default		System.Windows.Controls.Primitives.TextBc
Background	White	DefaultStyle	White	System.Windows.Controls.Panel
BetweenShowDelay	100	Default		System.Windows.Controls.ToolTipService
BindingGroup	null	Default		System.Windows.FrameworkElement
BitmapEffect	null	Default		System.Windows.UIElement
BitmapEffectInput	null	Default		System.Windows.UIElement
BitmapScalingMode	Unspecified	Default		System.Windows.Media.RenderOptions
BorderBrush	#FFABADB3	DefaultStyle	#FFABADB3	System.Windows.Controls.Border
BorderThickness	1,1,1,1	DefaultStyle	1,1,1,1	System.Windows.Controls.Border
CacheMode	null	Default		System.Windows.UIElement
CanRedo	False	Unknown		System.Windows.Controls.TextBox
CanUndo	True	Unknown		System.Windows.Controls.TextBox
Capitals	Normal	Default		System.Windows.Documents.Typography
CapitalSpacing	False	Default		System.Windows.Documents.Typography
CaretBrush	null	Default		System.Windows.Controls.Primitives.TextBc
CaretIndex	0	Unknown		System.Windows.Controls.TextBox
CaseSensitiveForms	False	Default		System.Windows.Documents.Typography
CharacterCasing	Normal	Default		System.Windows.Controls.TextBox
ClearTypeHint	Auto	Default		System.Windows.Media.RenderOptions
Clip	null	Default		System.Windows.UIElement
ClipToBounds	False	Default		System.Windows.UIElement

Figure 25: WPF Tree Visualizer Showing a Control's Property Values

As you can see in Figure 25, the left side the WPF Tree Visualizer shows the visual tree of the user interface and a preview of how the selected control is rendered at runtime. On the right side, the tool shows the full list of properties for the selected control and their values. You can also filter the property list using the text box at the top of the window. The WPF Tree Visualizer does not allow you to change property values at runtime, but in Chapter 5 you will discover how to accomplish this with more flexible tools.

Interaction with the debugger: the `Debug` and `Trace` classes

There are situations in which you do not need (or you do not want) to step through the code in break mode in order to understand what's happening with your application during the execution, yet you will still need to retrieve some information—for example, a variable's value. Visual Studio and the .NET Framework provide the **`Debug`** and **`Trace`** classes, both from the **`System.Diagnostics`** namespace, that allow interacting with the debugger in managed code and that also allow you to show information in the Output window. These classes are essentially identical, which means you can use them interchangeably, and both are static, exposing only static members. They evaluate conditions at a certain point of your code, then they can display contents in the Output window. For example, if you place the following line at the end of the

`OpenFileAsync` method described in the previous chapter, you will see the content of the resulting string in the Output window:

```
System.Diagnostics.Debug.WriteLine(result);
```

Table 2 shows the list of methods exposed by **`Debug`** and **`Trace`** classes.

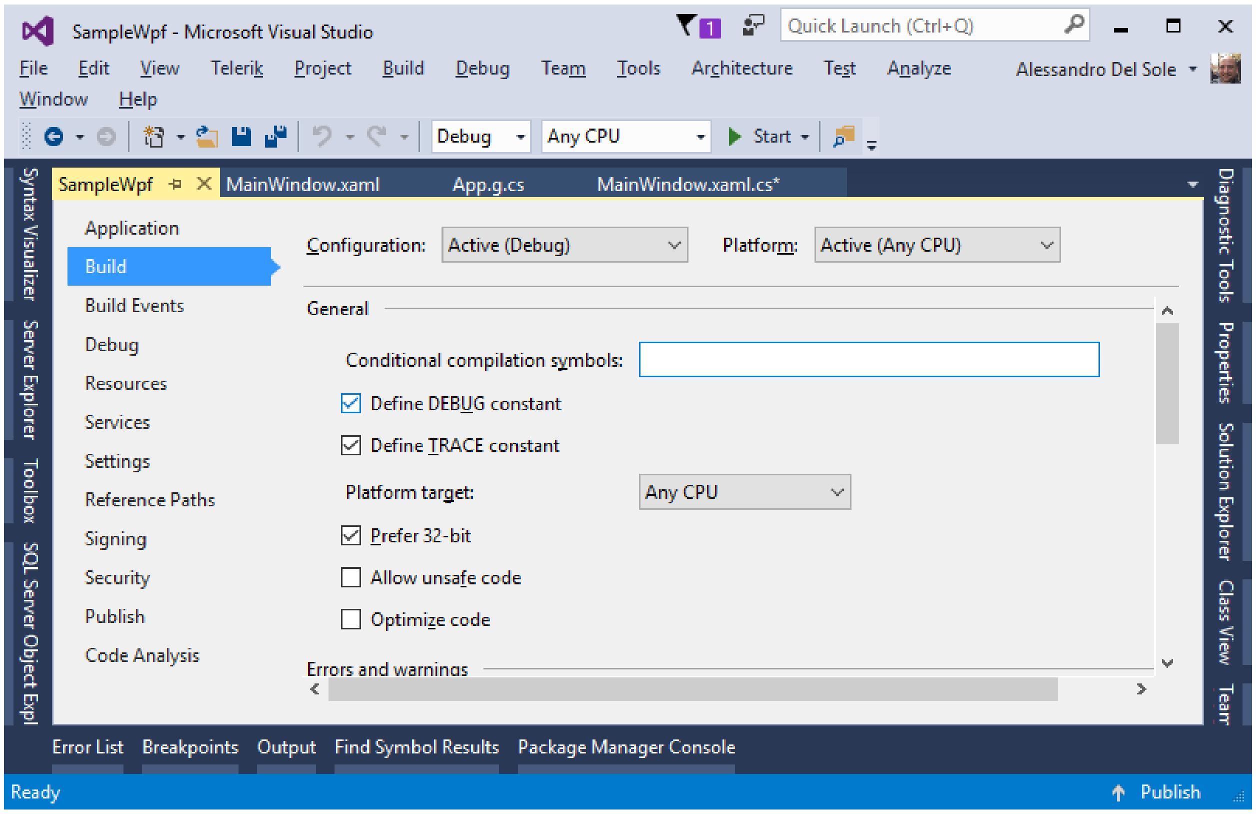
Table 2: Methods Exposed by Debug and Trace Classes

`Assert`	Evaluates a condition and shows a message if it is false.
`Fail`	Shows an error message.
`Indent`	Increases text indentation when writing to the Output window.
`Print`	Prints the specified message with support for formatting.
`Unindent`	Decreases indentation when writing to the Output window.
`Write`	Writes the specified message without a line terminator.
`WriteIf`	Writes the specified message without a line terminator if the specified condition is true.
`WriteLine`	Writes the specified message with a line terminator.
`WriteLineIf`	Writes the specified message with a line terminator if the specified condition is true.

Actually, **`Debug`** and **`Trace`** not only allow writing diagnostic information to the Output window, but they also allow redirecting output from the debugger to the so-called trace listeners, which are described in the next section. In this case, both classes also offer the **`Close`** and **`Flush`** methods that empty the debugger's buffer immediately and cause data to be written to the underlying listeners.

Controlling trace information

By default, when you build a project with the Debug configuration, the output of both **`Debug`** and **`Trace`** classes is included in the build output. This happens because the Debug configuration defines two constants, **`DEBUG`** and **`TRACE`**, which influence what output must be included. In order to change this behavior, you need to open the project's Properties window, select the **Build** tab, and change the selection for the **Define DEBUG Constant** and **Define TRACE Constant** options. Figure 26 shows where these options can be found.



Configuration: Active (Debug)
Platform: Active (Any CPU)

General

Conditional compilation symbols:

☑ Define DEBUG constant

☑ Define TRACE constant

Platform target: Any CPU

☑ Prefer 32-bit

☐ Allow unsafe code

☐ Optimize code

Errors and warnings

Figure 26: Controlling Whether DEBUG and TRACE Constants Should Be Defined

The Release configuration automatically includes the **TRACE** constant definition.

Exporting debugger information with trace listeners

By default, the **Debug** and **Trace** classes allow us to send information to the Output window. Actually, the Output window is one of the so-called **trace listeners**, special objects that can listen to the debugger and collect information in various forms. This means that you are not limited to sending information to the Output window. In fact, you can collect debugger information in the form of text files, XML files, and more. Both classes expose a **Listeners** property that can contain one or more built-in listeners. For example, the code shown in Code Listing 6 will redirect the debugger output to a text file instead of the Output window. This is accomplished by instantiating a listener called **TextWriterTraceListener**.

Code Listing 6

```
Trace.Listeners.Clear(); // Remove any listener.
// Redirect the output to a text file.
Trace.Listeners.Add(
      new TextWriterTraceListener("Diagnostics.txt"));
// Ensure the output file is closed when
// the debugger shuts down.
```

```
Trace.AutoFlush = true;

// Write the message only if the specified condition is true.
Trace.WriteLineIf(!string.IsNullOrEmpty(result), "Valid file");
```

Notice that listeners do not overwrite existing files, they append information. If you want a new file every time, you must first remove the previous file. Table 3 shows the list of available trace listeners in the .NET Framework.

Table 3: Built-in Trace Listeners in the .NET Framework

`DefaultTraceListener`	Sends the debugger output to the Output window.
`ConsoleTraceListener`	Sends the debugger output to the Console window.
`DelimitedListTraceListener`	Sends the debugger output to a text file with information delimited by a symbol.
`EventLogTraceListener`	Sends the debugger output to the Windows OS events log (requires administrator privileges).
`EventSchemaTraceListener`	Sends the debugger output to an XML file that will be generated on an XML schema formed on the supplied parameters.
`TextWriterTraceListener`	Sends the debugger output to a text file.
`XmlWriterTraceListener`	Sends the debugger output to an XML file.

Each listener is used in similar fashion, and, as usual, IntelliSense will help you pass the proper arguments to the constructor.

Working with trace listeners at configuration level

You are not limited to using trace listeners in C# or Visual Basic code. In fact, you can add listeners to a configuration file (App.config), which will make it easy to collect diagnostic information for a system administrator. Adding listeners to a configuration file must be done inside a node called **`system.diagnostics`**. For example, Code Listing 7 shows how to implement a trace listener that redirects the debugger output to a text file.

Code Listing 7

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0"
         sku=".NETFramework,Version=v4.6.1" />
    </startup>
  <system.diagnostics>
    <trace autoflush="true">
      <listeners>
        <add name="DiagnosticTextWriter"
        type="System.Diagnostics.TextWriterTraceListener"
        initializeData="Diagnostics.txt"/>
        <!-- If you want to disable the DefaultTraceListener-->
        <remove name="Default"/>
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

You will be able to use other trace listeners in similar ways. If you do this in code instead of the configuration file, you will have more control over these objects. Further information about trace and debug settings in the configuration file can be found in the MSDN documentation.

Chapter summary

Visual Studio offers additional useful tools for investigating objects and values. With debugger visualizers, you have an option to visualize objects and members that are supplied in particular formats, such as text, XML, or JSON. With WPF, the WPF Tree Visualizer allows you to inspect the visual tree for a window or user control at runtime and examine its property values. If you do not need to break the execution and inspect an object's value in break mode, you can also leverage the **`Debug`** and **`Trace`** classes to send the evaluation of an expression or an object's value to the Output window. Sometimes you do not want to send information to the Output window, so you can work with trace listeners and redirect the debugger output to text files, XML files, or the Windows' event log.

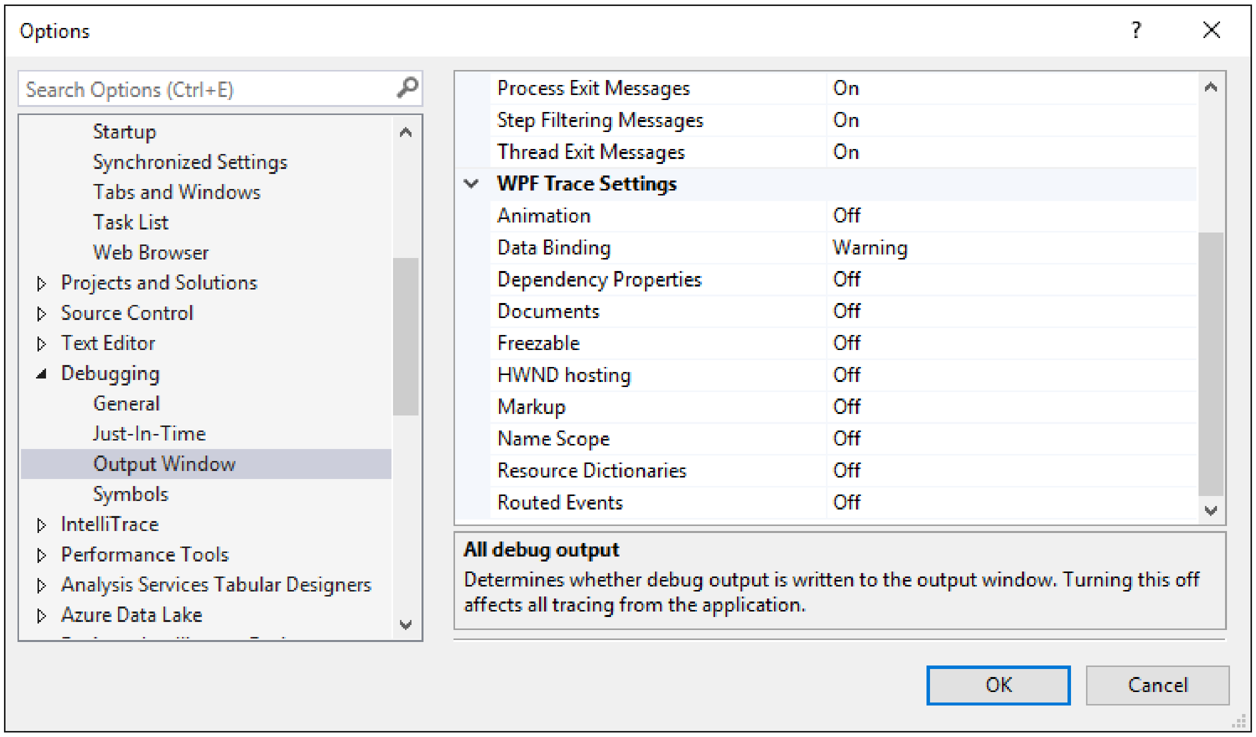
Chapter 5 XAML Debugging

Without a doubt, the most important part of the WPF development platform is the user interface. With WPF, you can create powerful, beautiful, and dynamic applications with a modern user interface. While designing, developing, and testing your apps' user interface, you might need to inspect the behavior of controls, styles, and templates at runtime, or you might need to discover subtle bugs, especially with data-binding. Before Visual Studio 2015, every time you needed to make an edit to your XAML, you had to break the application execution. Investigating the visual tree at runtime was impossible unless you used external tools. Because investigating XAML and the visual tree at runtime can be very important, Microsoft improved the debugging experience in Visual Studio 2015, offering tools that make it easier to understand if the UI is behaving as expected and also easier to make changes while the app is running.

Note: This chapter requires at least Visual Studio 2015 Update 2. As of writing this, Microsoft has released Update 3, which you can download at https://go.microsoft.com/fwlink/?LinkId=691129.

WPF Trace

WPF Trace is a very useful but often forgotten tool. At debugging time, WPF Trace sends diagnostic information to the Output window based on varying verbosity levels. For example, if your XAML has invalid data-bindings, these are reported in the Output window, which makes investigating (and solving) problems easier. WPF Trace is not limited to XAML and data-binding; it actually includes some tracing scenarios related to XAML, but it offers more. WPF Trace can be enabled by selecting **Tools**, **Options**, **Debugging**, **Output Window**, then scrolling to the WPF Trace Settings section, as you can see in Figure 27.



Process Exit Messages	On
Step Filtering Messages	On
Thread Exit Messages	On
WPF Trace Settings	
Animation	Off
Data Binding	Warning
Dependency Properties	Off
Documents	Off
Freezable	Off
HWND hosting	Off
Markup	Off
Name Scope	Off
Resource Dictionaries	Off
Routed Events	Off

All debug output
Determines whether debug output is written to the output window. Turning this off affects all tracing from the application.

Figure 27: Displaying WPF Trace Settings

By default, tracing is disabled for all scenarios except for Data Binding, whose default level is Warning. This means that warning messages will be shown in the Output window if data-binding is not working properly. To understand how tracing works, let's consider an invalid data-binding, which is probably the most common situation for using the tool. Suppose, for example, you have the **`ListBox`** control shown in Code Listing 8, which is used to display a list of active processes on your machine.

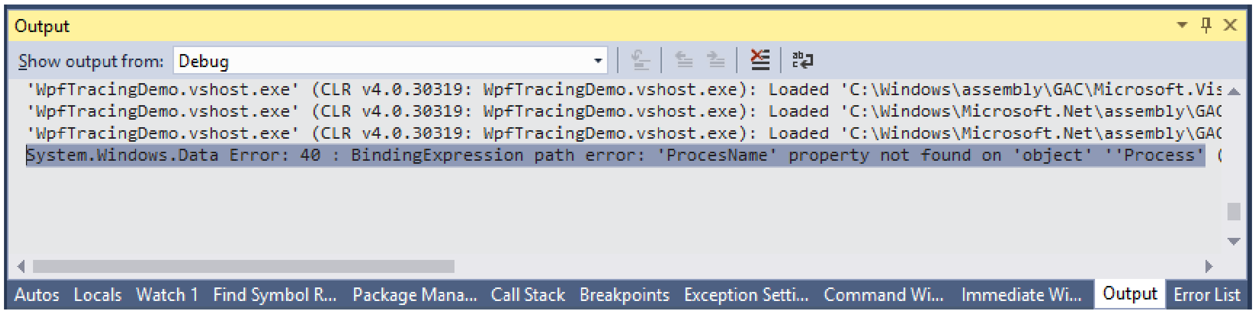
Code Listing 8

```
        <ListBox Name="FileListBox" ItemsSource="{Binding}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <!-- Binding target name is intentionally wrong -->
                    <TextBlock Text="{Binding ProcesName}"/>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
```

As you can see, the binding target name is wrong (which is intentional for demonstration purposes); in fact, it should be **`ProcessName`**, not **`ProcesName`**. The data source for the ListBox could be assigned as follows:

```
FileListBox.ItemsSource =
System.Diagnostics.Process.GetProcesses().AsEnumerable();
```

When you run this code, the Output window will show a warning message (see Figure 28) explaining that there is a data-binding error because a property called **`ProcesName`** was not found on the bound object called **`Process`**.



```
'WpfTracingDemo.vshost.exe' (CLR v4.0.30319: WpfTracingDemo.vshost.exe): Loaded 'C:\Windows\assembly\GAC\Microsoft.Vis
'WpfTracingDemo.vshost.exe' (CLR v4.0.30319: WpfTracingDemo.vshost.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC
'WpfTracingDemo.vshost.exe' (CLR v4.0.30319: WpfTracingDemo.vshost.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC
System.Windows.Data Error: 40 : BindingExpression path error: 'ProcesName' property not found on 'object' ''Process' (
```

Figure 28: Displaying WPF Trace Settings

Tip: Scroll the Output window horizontally to see the full diagnostic message.

WPF Trace tells you that the supplied binding source (the property name) has not been found on the bound object (**`Process`**, in this case). This will help you detect and fix the problem immediately. You can get even more detailed information by changing the trace level from **Warning** to **Verbose**.

Note: You might expect that setting the trace level to Error would break the application execution. Actually, setting the trace level to Error will only change the way the diagnostic information is presented—it will not break the execution.

Though WPF Trace can be used in a number of scenarios, without a doubt it is particularly useful with invalid data-binding and animation bindings.

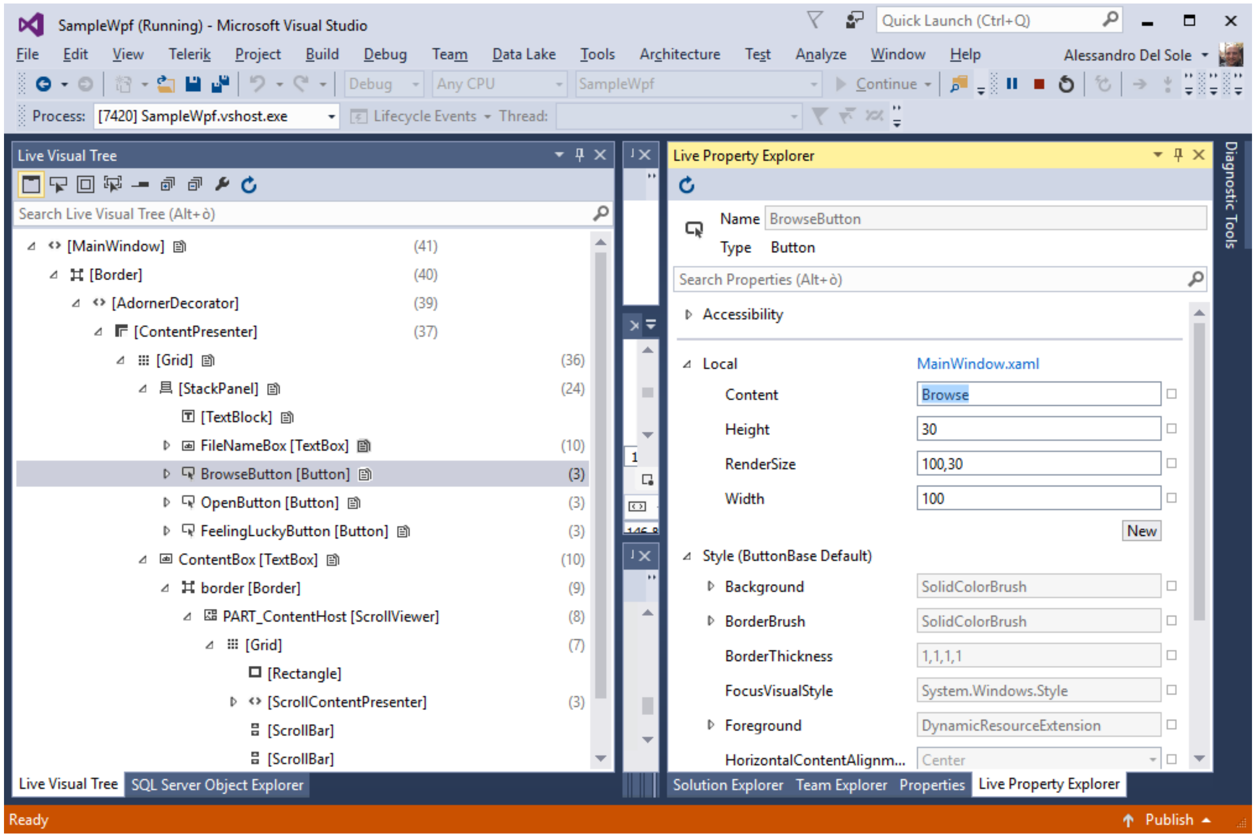
UI debugging tools

Visual Studio 2015 has dramatically improved the debugging experience for WPF (and Universal Windows Platform, as well) by introducing a number of tools that make it easy to investigate and change the behavior of the user interface at runtime. This section describes new tool windows introduced with Visual Studio 2015, Live Visual Tree and Live Property Explorer, and the In-App Menu, available with Visual Studio 2015 Update 2 and higher.

Live Visual Tree and Live Property Explorer

Note: In order to fully discuss the UI debugging tools, make sure you go to Tools, Options, Debugging, then select an option called "Preview selected elements in Live Visual Tree."

Visual Studio 2015 introduced two new tool windows to the WPF debugging experience: Live Visual Tree and Live Property Explorer. Live Visual Tree shows the full visual tree of the user interface; when you select an element in the visual tree view, Live Property Explorer shows the full list of property values for the selected item and allows changing property values at runtime (if supported), so that you can immediately see how the user interface changes. Both windows should be automatically visible when you begin debugging a WPF application. If not, you can pick them up from Debug, Windows. Figure 29 shows Live Visual Tree and Live Property Explorer over the SampleWpf application we previously created.



```
Process: [7420] SampleWpf.vshost.exe

[MainWindow]                         (41)
  [Border]                           (40)
    [AdornerDecorator]               (39)
      [ContentPresenter]             (37)
        [Grid]                       (36)
          [StackPanel]               (24)
            [TextBlock]
            FileNameBox [TextBox]    (10)
            BrowseButton [Button]    (3)
            OpenButton [Button]      (3)
            FeelingLuckyButton [Button] (3)
          ContentBox [TextBox]       (10)
            border [Border]          (9)
              PART_ContentHost [ScrollViewer] (8)
                [Grid]               (7)
                  [Rectangle]
                  [ScrollContentPresenter] (3)
                  [ScrollBar]
                  [ScrollBar]
```

```
Name  BrowseButton
Type  Button

Accessibility
Local                        MainWindow.xaml
  Content                    Browse
  Height                     30
  RenderSize                 100,30
  Width                      100
Style (ButtonBase Default)
  Background                 SolidColorBrush
  BorderBrush                SolidColorBrush
  BorderThickness            1,1,1,1
  FocusVisualStyle           System.Windows.Style
  Foreground                 DynamicResourceExtension
  HorizontalContentAlignm... Center
```

Figure 29: Live Visual Tree and Live Property Explorer

As you can see, selecting an item in the visual tree will cause Live Property Explorer to display properties and their values in that exact moment of the application lifecycle. Notice that some property values can be changed at runtime so that you can immediately see how the user interface reflects those changes. For instance, Figure 29 shows how you can change the value for the button's **`Content`** property with a new string, and you will immediately see the new value in the user interface while running. When you select an element in the Live Visual Tree, the corresponding line of code is selected in the XAML editor (you can also achieve this by right-clicking the element, then selecting View Source). Before Visual Studio 2015 Update 2, interacting with Live Visual Tree and Live Property Explorer required you to move your focus from the active window to Visual Studio. As we'll see in the next section, this situation has been improved.

XAML In-App Menu

Visual Studio 2015 Update 2 introduced the In-App Menu, a tool that makes it easier to inspect the behavior of your XAML at runtime. This tool consists of a pane that appears over a window at debugging time and provides a number of buttons that will be discussed shortly. Figure 30 shows the In-App Menu.

Tip: The In-App Menu is enabled by default in XAML-based platforms such as WPF and Universal Windows Platform. If you want to disable it, go to Tools, Options, Debugging, then unselect the "Show runtime tools in application" option.

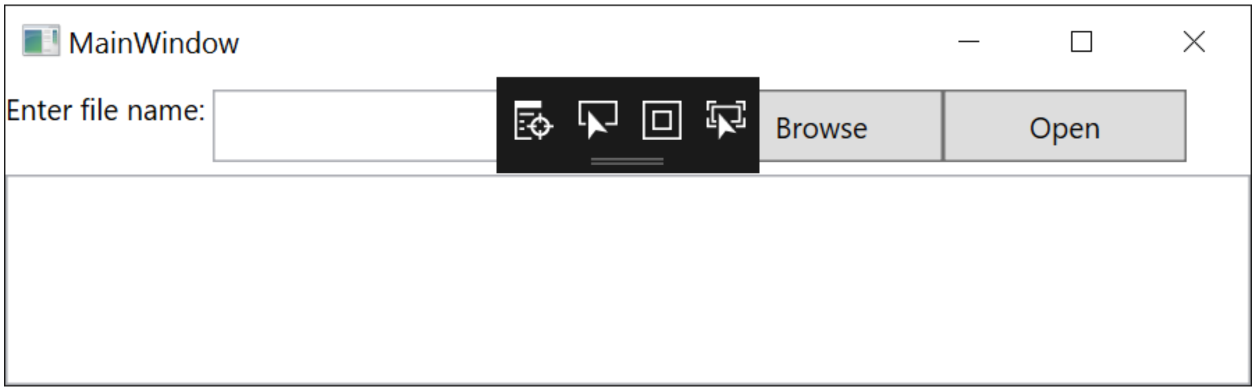


Figure 30: The In-App Menu Expanded

The In-App Menu can certainly be minimized, which helps us avoid overlaying parts of the UI, but for now leave it open. The menu has four buttons (described from left to right in the next paragraphs).

Go to Live Visual Tree

As the name implies, this button simply opens the Live Visual Tree tool window. I suggest you dock the Live Visual Tree window so that you will immediately see the result of the next buttons.

Enable Selection

This button allows you to select controls on the user interface. When you select a control, this is surrounded with a red border, and the Live Visual Tree window automatically shows the selected control within the visual tree. Figure 31 shows an example.

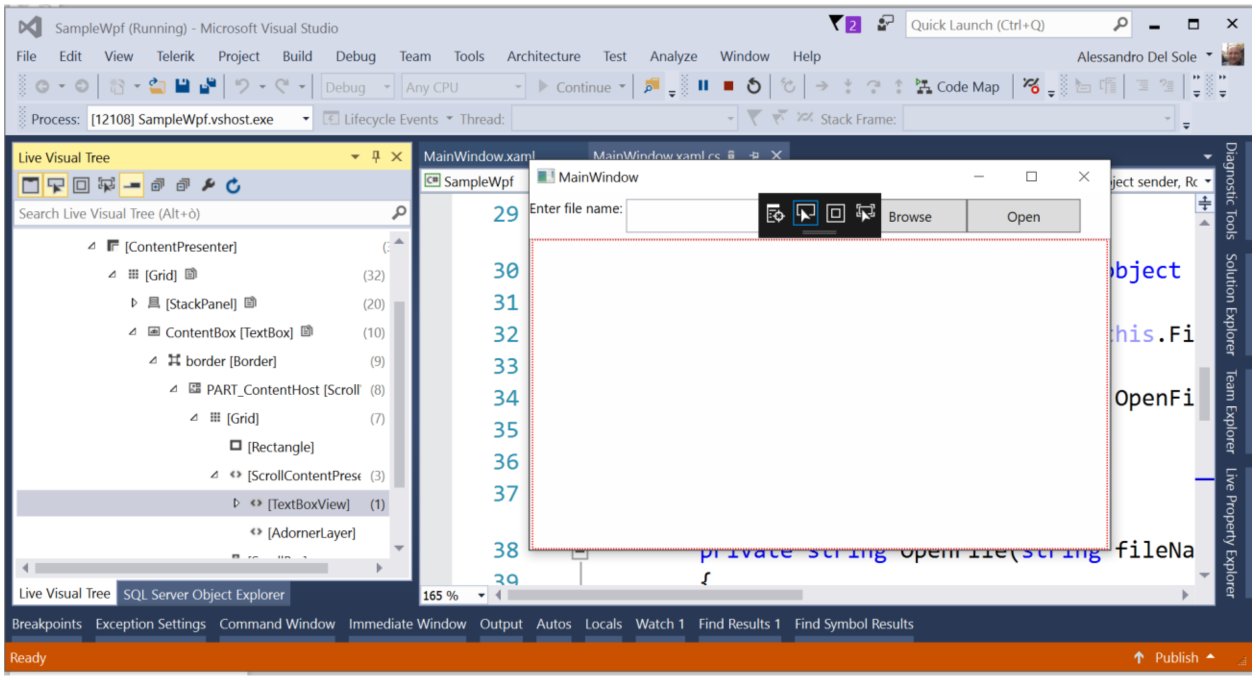


Figure 31: Enabling Selection in the In-App Menu

Display Layout Adornments

This button allows you to highlight the surface of a control. If combined with Enable Selection, a control is both highlighted and selected. This is useful for understanding the delimiters of a control. Figure 32 shows an example based on the combination of both buttons.

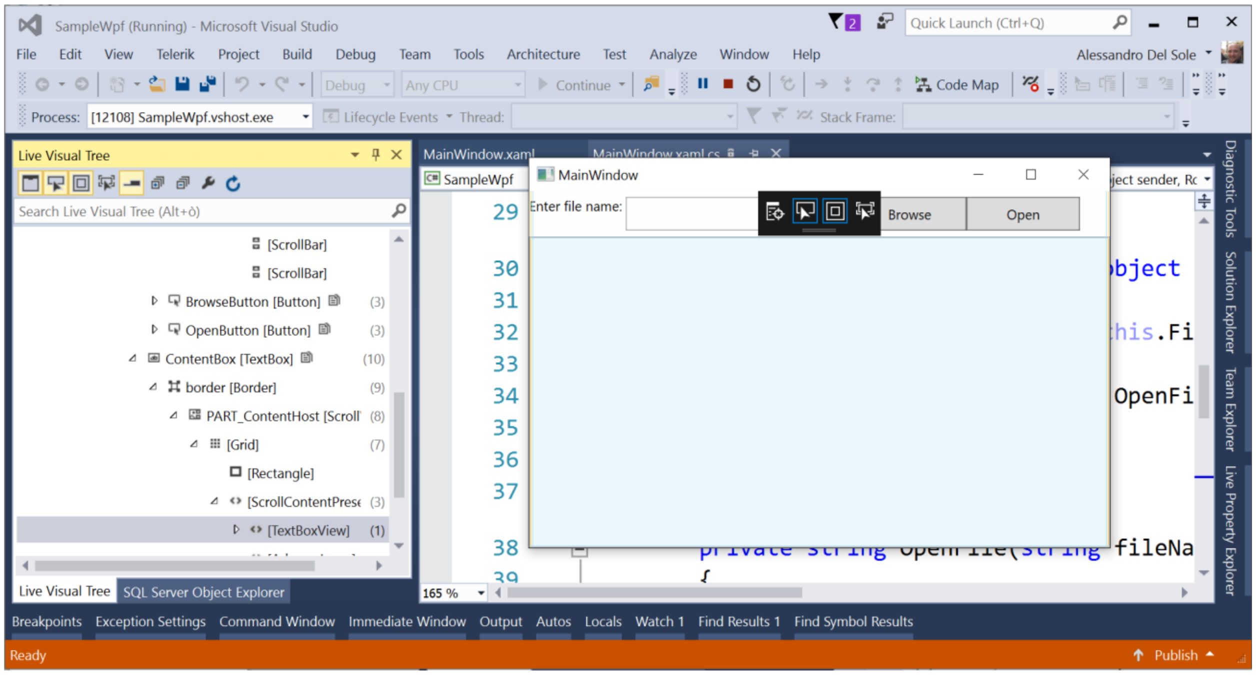

SampleWpf (Running) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Telerik Project Build Debug Team Tools Architecture Test Analyze Window Help
Alessandro Del Sole
Process: [12108] SampleWpf.vshost.exe
Live Visual Tree
Search Live Visual Tree (Alt+ò)
[ScrollBar]
[ScrollBar]
BrowseButton [Button] (3)
OpenButton [Button] (3)
ContentBox [TextBox] (10)
border [Border] (9)
PART_ContentHost [Scroll (8)
[Grid] (7)
[Rectangle]
[ScrollContentPrese (3)
[TextBoxView] (1)
Live Visual Tree
SQL Server Object Explorer
MainWindow.xaml
MainWindow.xaml.cs
SampleWpf
MainWindow
Enter file name:
Browse
Open
ject sender, Rc
bject
his.Fi
OpenFi
private string OpenFile(string fileNa
165 %
Breakpoints Exception Settings Command Window Immediate Window Output Autos Locals Watch 1 Find Results 1 Find Symbol Results
Ready
Publish

Figure 32: Displaying Layout Adornments

Track Focused Element

The Track Focused Element is similar to Enable Selection in that it allows selecting a control and reflects the selection in Live Visual Tree, but it only allows selecting controls that can receive focus (for instance, `TextBox` controls).

XAML Edit and Continue

The new release of Visual Studio includes a new feature known as **XAML Edit and Continue**. This feature allows editing some property values in your XAML at debugging time, with changes immediately reflected in the user interface of your application. For instance, you might want to see how changing a style or control template affects the user interface while running.

Chapter summary

With WPF, debugging means not only investigating code for bugs and checking for errors. It also means checking if the user interface is designed and behaves as expected, which involves analyzing controls and XAML at debugging time. Visual Studio 2015 and 2017 have specific tools that make this task a comfortable experience. With the WPF Trace tool, you can easily detect errors in your XAML code, especially for data-binding. The Live Visual Tree and Live Property Explorer tool windows represent the visual tree and change UI elements' property values, where supported. An easier interaction with both windows is possible through the new In-App Menu, which provides buttons that easily map controls to XAML code and to visual tree elements.

Chapter 6 Analyzing the UI Performances

WPF has support for multimedia, animations, and documents. It also provides a very powerful data-binding engine with built-in virtualization for long lists of data. If your application works with hundreds of UI elements, including animations, multimedia, and data-bound controls, you must be aware of possible performance issues. The UI might slow down, and perceived performances might become tedious. Fortunately, Visual Studio has an interesting tool called Application Timeline that allows you to analyze the user interface performances, thereby making it easier to improve the user experience.

Preparing an example

Our previous sample application is very simple, and its user interface cannot have significant performance issues. For this reason, it's a good idea to prepare a window that does more intensive work. First, in Solution Explorer, right-click the project name and select **Add**, **Window**. When the Add New Item dialog appears, enter ImageRenderingWindow.xaml as the name. This window's goal is to display 1000 images inside a **`ListBox`** in order to analyze how the WPF rendering engine works. You need only this control with a simple data template that shows the image and its file name. Code Listing 9 shows the full XAML for the new window.

Code Listing 9

```xml
<Window x:Class="SampleWpf.ImageRenderingWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
                  compatibility/2006"
        xmlns:local="clr-namespace:SampleWpf"
        mc:Ignorable="d"
        Title="ImageRenderingWindow" Height="300" Width="300">
    <Grid>
        <ListBox Name="ImageListBox" ItemsSource="{Binding}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Vertical">
                        <Image Width="320" Height="240"
                               Source="{Binding ImagePath}"/>
                        <TextBlock Text="{Binding ImageName}" />
                    </StackPanel>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
</Window>
```

The data template for the **ListBox** expects a class with two properties—one for the image path and one for the image file name. The class is called **ImageFile** and is defined in Code Listing 10 together with a collection called **ImageFileCollection**.

Code Listing 10

```
    public class ImageFile
    {
        public Uri ImagePath { get; set; }
        public string ImageName { get; set; }
    }

    public class ImageFileCollection: ObservableCollection<ImageFile>
    {
        public ImageFileCollection()
        {
            for (int i=0; i <= 1000; i++)
            {
                // Replace with a file you have on your pc.
                this.Add(new ImageFile { ImageName="Beach.jpg",
                    ImagePath = new Uri("Beach.jpg",UriKind.Relative) });
            }
        }
    }
```

For demonstration purposes only, the **ImageFileCollection** class is populated with 1000 instances of the same image in order to cause some overhead. Now make the new window the startup object for the application by editing the **StartupUri** attribute of the **Application** node in the App.xaml file as follows:

StartupUri="ImageRenderingWindow.xaml"

Next, edit the new **Window**'s constructor as shown in Code Listing 11 in order to assign a new **ImageFileCollection** as the data source.

Code Listing 11

```
        public ImageRenderingWindow()
        {
            InitializeComponent();
            this.DataContext = new ImageFileCollection();
        }
```

When ready, launch the application. It should look like in Figure 33.

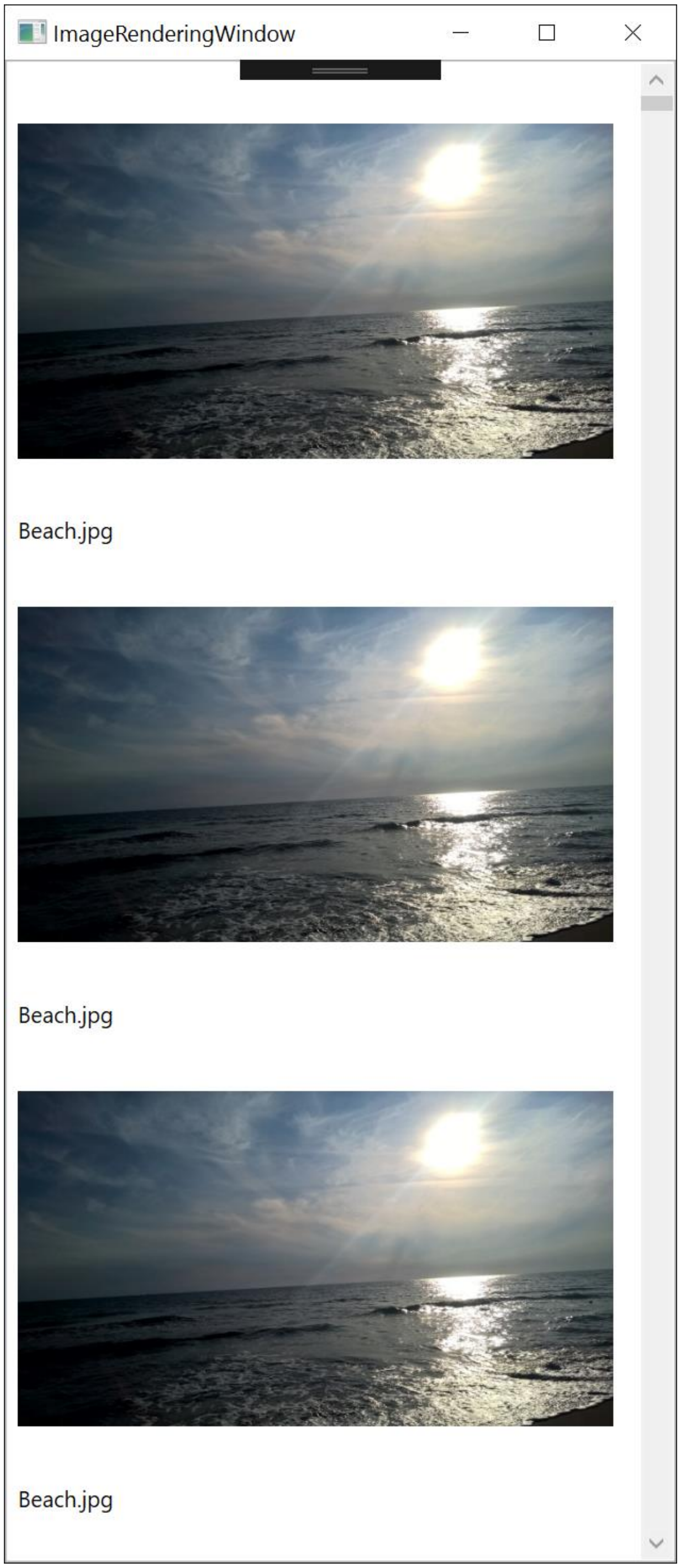


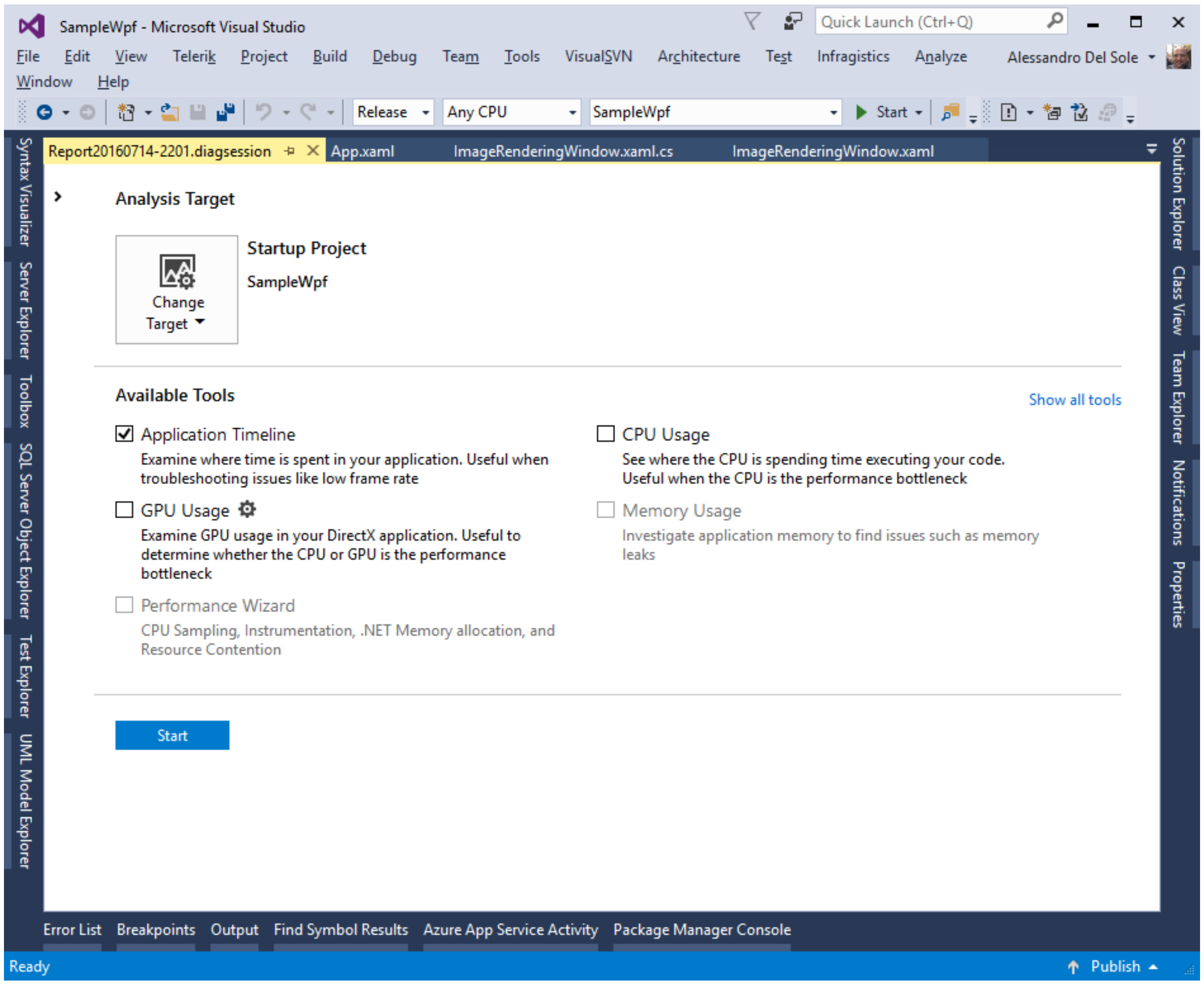
Figure 33: Sample App Showing 1000 Instances of Image

Next, let's examine how to analyze the UI performance with the integrated Visual Studio tools.

The Application Timeline tool

Visual Studio 2015 brings to WPF a diagnostic tool called the XAML UI Responsiveness Tool. It was originally developed for Windows Store apps and was introduced in Visual Studio 2013; it is certainly still available in version 2017. Now this tool is called Application Timeline (or, more simply, Timeline), and you can leverage it with your WPF applications. The Timeline tool analyzes the application's behavior in order to help you detect where it spends time. Though this tool can detect events such as (but not limited to) startup, disk I/O, and rendering the UI (including thread utilization), its focus is investigating the UI behavior in more detail. In order to understand how it works, press ALT+F2 (or **Debug**, **Performance Profiler**) to show the so-called Diagnostics Hub, which contains a list of performance analysis tools, then select **Application Timeline** (see Figure 34).

Tip: If it looks disabled, make sure you first unselect any other diagnostic tool.


SampleWpf - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Telerik Project Build Debug Team Tools VisualSVN Architecture Test Infragistics Analyze Alessandro Del Sole
Window Help
Release
Any CPU
SampleWpf
Start
Report20160714-2201.diagsession
App.xaml
ImageRenderingWindow.xaml.cs
ImageRenderingWindow.xaml
Syntax Visualizer
Server Explorer
Toolbox
SQL Server Object Explorer
Test Explorer
UML Model Explorer
Solution Explorer
Class View
Team Explorer
Notifications
Properties
Error List
Breakpoints
Output
Find Symbol Results
Azure App Service Activity
Package Manager Console
Ready
Publish

Analysis Target

Change Target

Startup Project

SampleWpf

Available Tools

Show all tools

☑ Application Timeline

Examine where time is spent in your application. Useful when troubleshooting issues like low frame rate

☐ CPU Usage

See where the CPU is spending time executing your code. Useful when the CPU is the performance bottleneck

☐ GPU Usage

Examine GPU usage in your DirectX application. Useful to determine whether the CPU or GPU is the performance bottleneck

☐ Memory Usage

Investigate application memory to find issues such as memory leaks

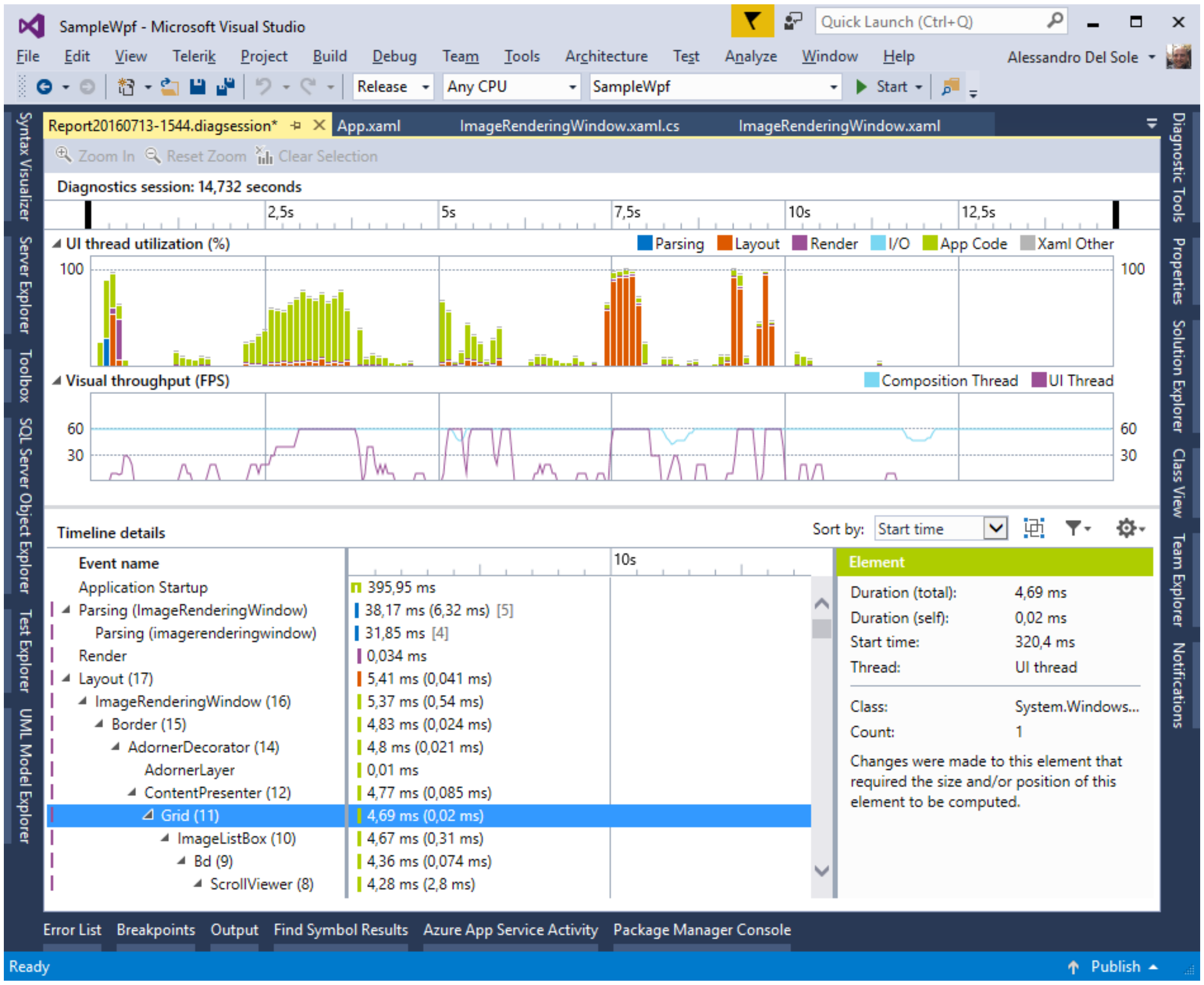
☐ Performance Wizard

CPU Sampling, Instrumentation, .NET Memory allocation, and Resource Contention

Start

Figure 34: Selecting the Application Timeline

Because the Timeline is a specialized profiler, for accurate results you should first change the output configuration for your project from Debug to Release. When ready, click **Start** so that the application will start with an instance of the profiler attached. When the application has started and all images have been loaded, play a bit with the user interface—for example, resize the window or scroll the image list. When your application has been working for a while in different situations, click the **Stop Collection** hyperlink in Visual Studio. After a few seconds, you will get a very detailed visual report (see Figure 35 for an example).



Event name	10s
Application Startup	395,95 ms
Parsing (ImageRenderingWindow)	38,17 ms (6,32 ms) [5]
Parsing (imagerenderingwindow)	31,85 ms [4]
Render	0,034 ms
Layout (17)	5,41 ms (0,041 ms)
ImageRenderingWindow (16)	5,37 ms (0,54 ms)
Border (15)	4,83 ms (0,024 ms)
AdornerDecorator (14)	4,8 ms (0,021 ms)
AdornerLayer	0,01 ms
ContentPresenter (12)	4,77 ms (0,085 ms)
Grid (11)	4,69 ms (0,02 ms)
ImageListBox (10)	4,67 ms (0,31 ms)
Bd (9)	4,36 ms (0,074 ms)
ScrollViewer (8)	4,28 ms (2,8 ms)

Element	
Duration (total):	4,69 ms
Duration (self):	0,02 ms
Start time:	320,4 ms
Thread:	UI thread
Class:	System.Windows...
Count:	1

Changes were made to this element that required the size and/or position of this element to be computed.

Figure 35: Report Generated by the Application Timeline

The report consists of four main areas: the Diagnostic session, the UI thread utilization, the Visual throughput (FPS), and the Timeline details. Let’s discuss each more thoroughly.

The Diagnostic session

The Diagnostic session report shows information about the duration of the diagnostic session. You can use the black markers to analyze a specific interval of time during the application lifetime. When you select a specific interval, the other areas will automatically show diagnostic information for that interval.

UI thread utilization

The UI thread utilization section gives a percentage for how the UI thread has been exploited by different tasks managed by the runtime. You can understand how many resources have been consumed by the XAML parser (blue), how many consumed in rendering the user interface (dark orange), how many consumed in executing the app code (light green), how many consumed in disk I/O (light blue), and how many consumed in other tasks related to XAML (not parsing). In conjunction with the Diagnostic session report, this can be very useful for understanding which areas of your code have the most negative impact on the overall performance at a specified interval of time in the application lifetime. For instance, in Figure 35 you will see there is an important impact in rendering the layout at 7.5 seconds after the application startup, which is the point at which 1000 images were rendered.

Visual throughput (FPS)

This section shows how many frames per second (FPS) have been rendered during the application lifecycle. For timing, you can take the Diagnostic session as a reference. This tool is very straightforward—it can show frames in both the UI thread and the composition thread. If you pass the mouse pointer over the graphic, you will see a tooltip showing frames per second for both threads at the given time.

Note: If you are not familiar with the composition thread, it is a companion thread for the UI thread, in that it does some work that should otherwise be done by the UI thread. The composition thread is normally responsible for combining graphic textures and sending them to the GPU for rendering. This is all managed by the runtime; by invoking the composition thread, the runtime can make an app stay much more responsive and you, the developer, need not do any additional work manually.

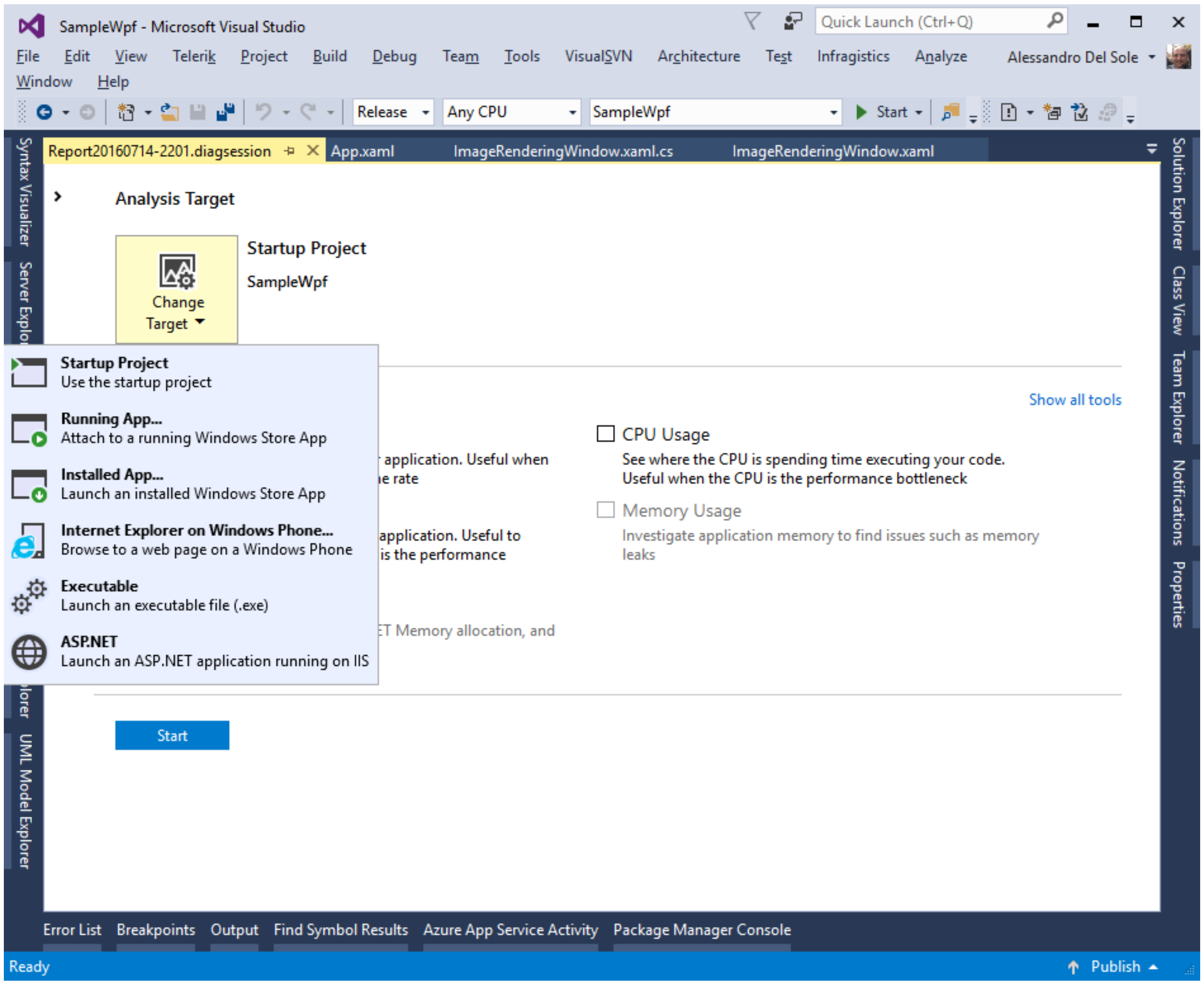
Timeline details

At the bottom of the report, you will find the Timeline details, which provide further information about events in the application lifecycle and about UI elements involved in such events. More specifically, you can see a list of events and UI elements. For each, you can see the time in milliseconds. For example, if you consider Figure 35, you can see how the Application Startup event has required 395.95 milliseconds. Some events, such as Layout, are split into multiple parts, which is typical because rendering the layout requires two different threads (UI and Composition) that run asynchronously. This means each node can be expanded to get more information. When you expand a Layout node, you also see the visual tree of UI elements involved in that particular event.

These UI elements can be expanded to show nested controls and types. When you click an event or object, you will also see detailed information on the right side of the report, including the number of instances of an object, the containing .NET type, and the time required to render a UI element. For example, if you again consider Figure 35, you'll see a description of a **`Grid`** on the right side of the report that is currently selected in the list of UI elements; such a description also shows how much time it took to render this **`Grid`** in milliseconds, the thread that was responsible for drawing the element (the UI thread in this case), and the count, which is one instance. Having the option to discover how many instances of an object have been created will help you understand which UI elements might negatively impact the overall UI performance.

Changing the analysis target

Analysis tools listed in the Diagnostics Hub can run against a number of possible targets. If you look at Figure 34, you'll see a button called Change Target. Click this button and you will see a list of possible analysis targets, such as a running or installed Windows Store app, an existing .exe file, and even an ASP.NET application running on IIS (see Figure 36).


SampleWpf - Microsoft Visual Studio
File Edit View Telerik Project Build Debug Team Tools VisualSVN Architecture Test Infragistics Analyze
Window Help
Release
Any CPU
SampleWpf
Start
Report20160714-2201.diagsession
App.xaml
ImageRenderingWindow.xaml.cs
ImageRenderingWindow.xaml
Analysis Target
Change Target
Startup Project
SampleWpf
Show all tools
Start
Error List Breakpoints Output Find Symbol Results Azure App Service Activity Package Manager Console
Ready
Publish

Startup Project
Use the startup project

Running App...
Attach to a running Windows Store App

Installed App...
Launch an installed Windows Store App

Internet Explorer on Windows Phone...
Browse to a web page on a Windows Phone

Executable
Launch an executable file (.exe)

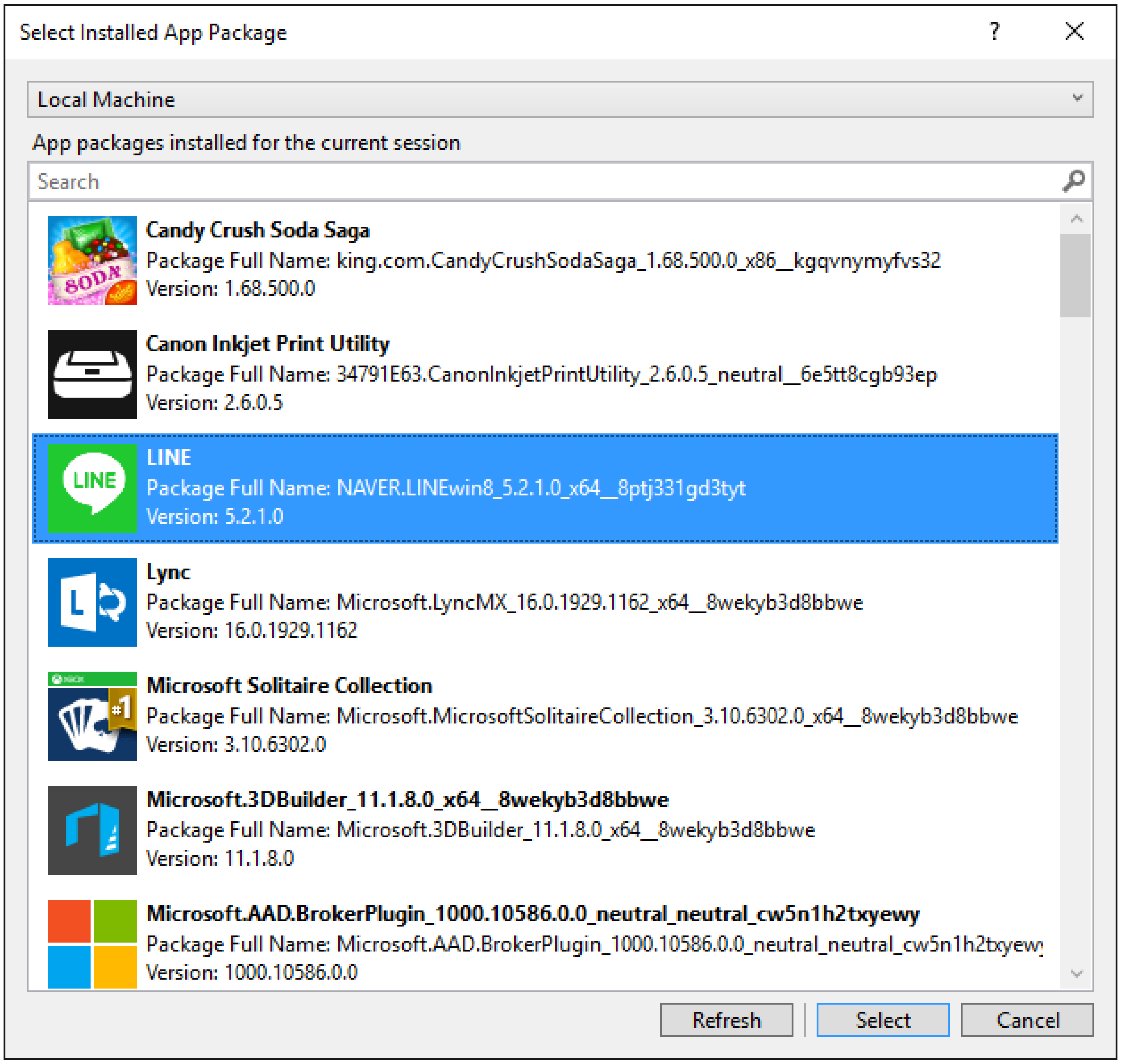
ASP.NET
Launch an ASP.NET application running on IIS

CPU Usage
See where the CPU is spending time executing your code. Useful when the CPU is the performance bottleneck

Memory Usage
Investigate application memory to find issues such as memory leaks

Figure 36: Possible Analysis Targets

The Application Timeline can only run against the first three targets: Startup Project, Running App, and Installed App. Both Running App and Installed App refer to Windows Store apps, not WPF applications. More precisely, note that if you select Installed App, you will be prompted with a list of installed Windows Store apps on your machine (which includes Universal Windows apps), as you can see in Figure 37.


Select Installed App Package
?
Local Machine
App packages installed for the current session
Search
Refresh
Select
Cancel

Candy Crush Soda Saga
Package Full Name: king.com.CandyCrushSodaSaga_1.68.500.0_x86__kgqvnymyfvs32
Version: 1.68.500.0

Canon Inkjet Print Utility
Package Full Name: 34791E63.CanonInkjetPrintUtility_2.6.0.5_neutral__6e5tt8cgb93ep
Version: 2.6.0.5

LINE
Package Full Name: NAVER.LINEwin8_5.2.1.0_x64__8ptj331gd3tyt
Version: 5.2.1.0

Lync
Package Full Name: Microsoft.LyncMX_16.0.1929.1162_x64__8wekyb3d8bbwe
Version: 16.0.1929.1162

Microsoft Solitaire Collection
Package Full Name: Microsoft.MicrosoftSolitaireCollection_3.10.6302.0_x64__8wekyb3d8bbwe
Version: 3.10.6302.0

Microsoft.3DBuilder_11.1.8.0_x64__8wekyb3d8bbwe
Package Full Name: Microsoft.3DBuilder_11.1.8.0_x64__8wekyb3d8bbwe
Version: 11.1.8.0

Microsoft.AAD.BrokerPlugin_1000.10586.0.0_neutral_neutral_cw5n1h2txyewy
Package Full Name: Microsoft.AAD.BrokerPlugin_1000.10586.0.0_neutral_neutral_cw5n1h2txyewy
Version: 1000.10586.0.0

Figure 37: Selecting an Installed Windows Store App as the Analysis Target

Simply select an application, click **Select**, then start the diagnostic session with the Application Timeline.

Chapter summary

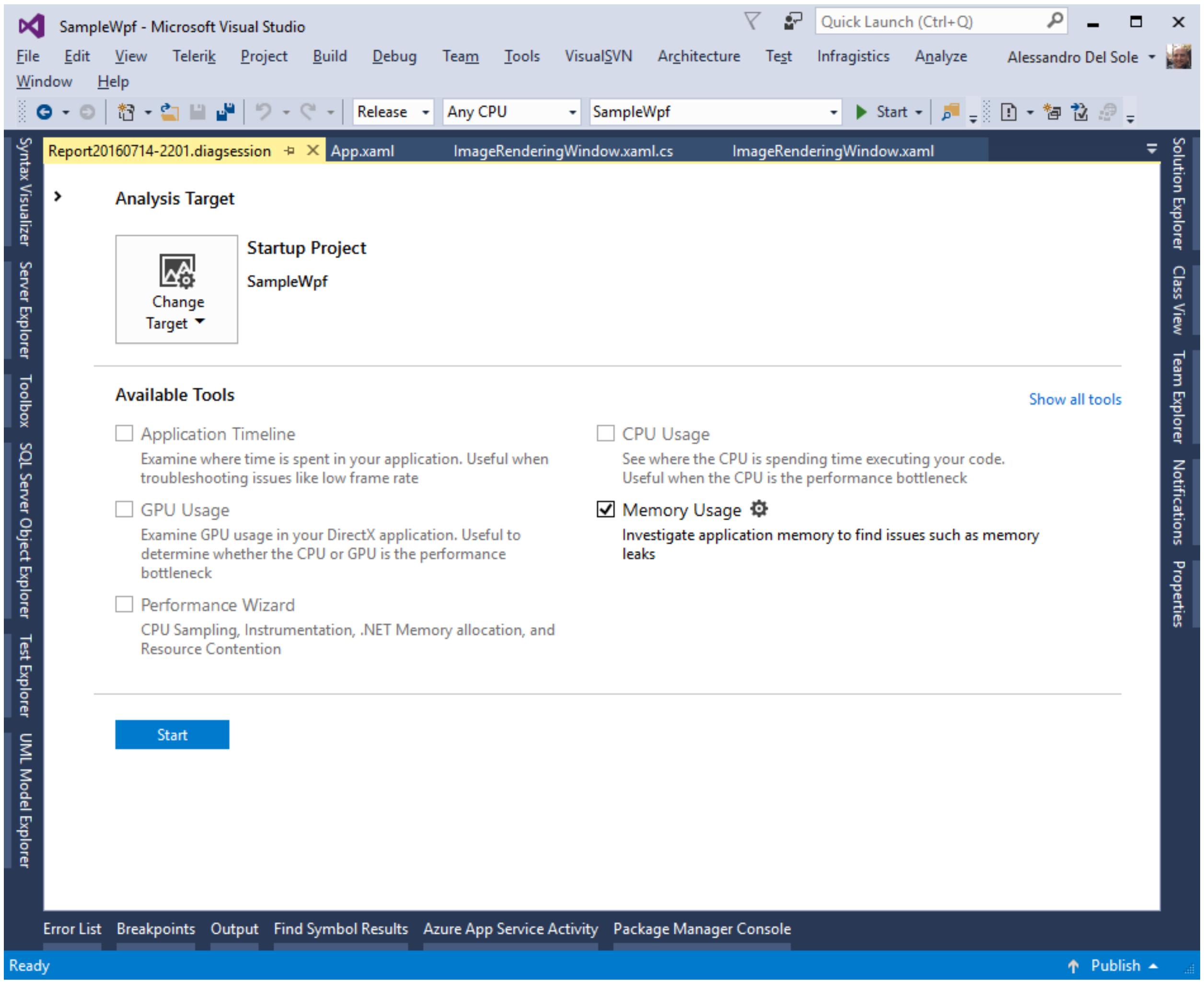
Performance is one of the most important aspects of any professional application. You can build a beautiful application, but if it is slow or unresponsive, customers and users will be very dissatisfied. This is even more important with WPF applications in which the user interface natively supports media, documents, and XAML data-binding. In order to help you analyze the performance of the user interface in your applications, Visual Studio 2015 ships with a tool called Application Timeline that analyzes events such as startup and disk I/O but focuses on how much time it takes to render the application’s user interface, which will allow you to understand which elements are consuming which resources at a particular point in the application lifecycle. By analyzing the report produced by Timeline, you will be able to understand which parts of the user interface need to be restructured in order to improve the customer experience.

Chapter 7 Analyzing the Application Performances

We have seen how the Application Timeline tool provides a convenient way to analyze the behavior of the user interface of your application during its lifecycle. This is particularly important, especially for perceived performance, but an application is not just user interface. You might have a simple UI with code that works with hundreds of objects but that could cause memory leaks, or your application might perform CPU-intensive work. In such cases, analyzing memory allocation and CPU utilization is both useful and important. This chapter provides guidance on built-in profiling tools in Visual Studio that will help you solve problems related to memory allocation and CPU utilization.

Investigating memory allocation

Understanding how the application uses memory is one of the most important steps in building performant applications. Visual Studio provides an analysis tool called Memory Usage you can find in the Diagnostics Hub and that you can see by pressing Alt+F2 (or **Debug**, **Performance Profiler**). Figure 38 shows how to enable this tool.



Analysis Target

Startup Project
SampleWpf

Available Tools

[ ] Application Timeline
Examine where time is spent in your application. Useful when troubleshooting issues like low frame rate

[ ] CPU Usage
See where the CPU is spending time executing your code. Useful when the CPU is the performance bottleneck

[ ] GPU Usage
Examine GPU usage in your DirectX application. Useful to determine whether the CPU or GPU is the performance bottleneck

[✓] Memory Usage
Investigate application memory to find issues such as memory leaks

[ ] Performance Wizard
CPU Sampling, Instrumentation, .NET Memory allocation, and Resource Contention

Figure 38: Selecting Memory Usage as the Analysis Tool

Make sure the Release configuration is selected, then click **Start** to begin a diagnostic session. After a few seconds, you will see that Visual Studio shows the Live Graph and immediately begins reporting the memory usage in MB during the entire application lifecycle (see Figure 39).

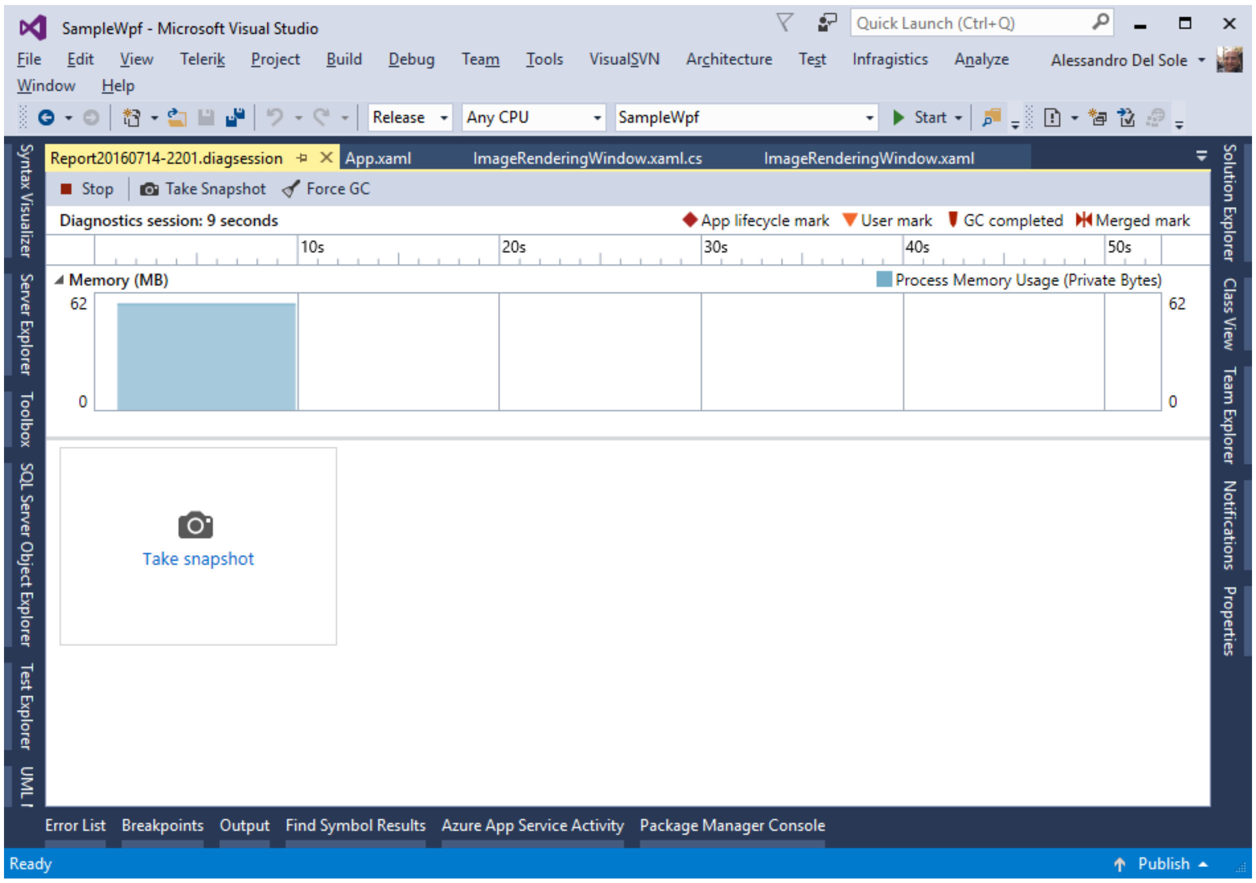


Figure 39: Memory Usage Reporting Memory Usage

This kind of report can be useful for determining if memory usage increases or decreases, but the great benefit of this tool is its ability to take snapshots of the memory at a specific point in time. You make this happen by clicking **Take Snapshot**. Each snapshot contains information about object instances and managed heap size, and, by comparing that shot with a previous snapshot, we can see if the amount of used memory has increased or decreased. Figure 40 shows the result of capturing two snapshots.

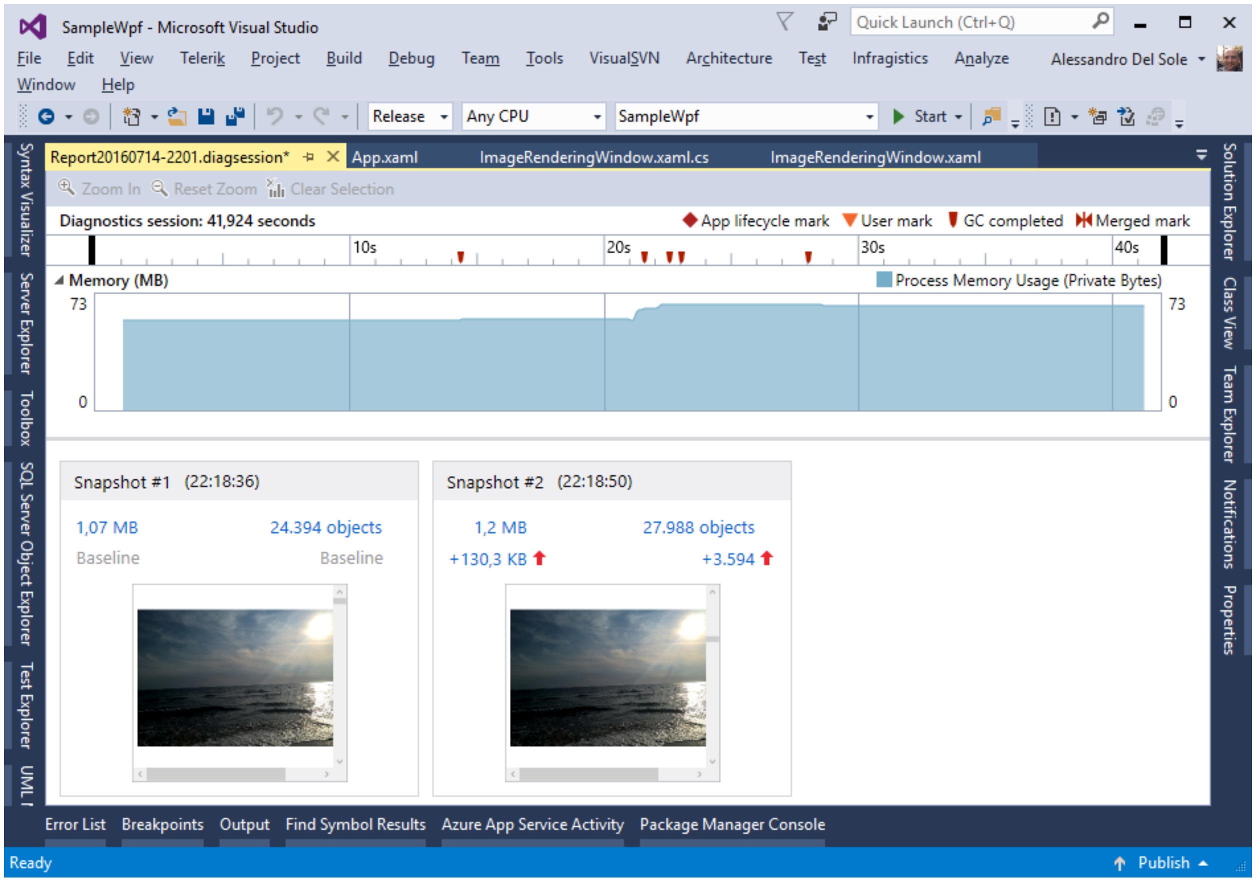


Figure 40: Memory Usage Again Reporting Memory Used

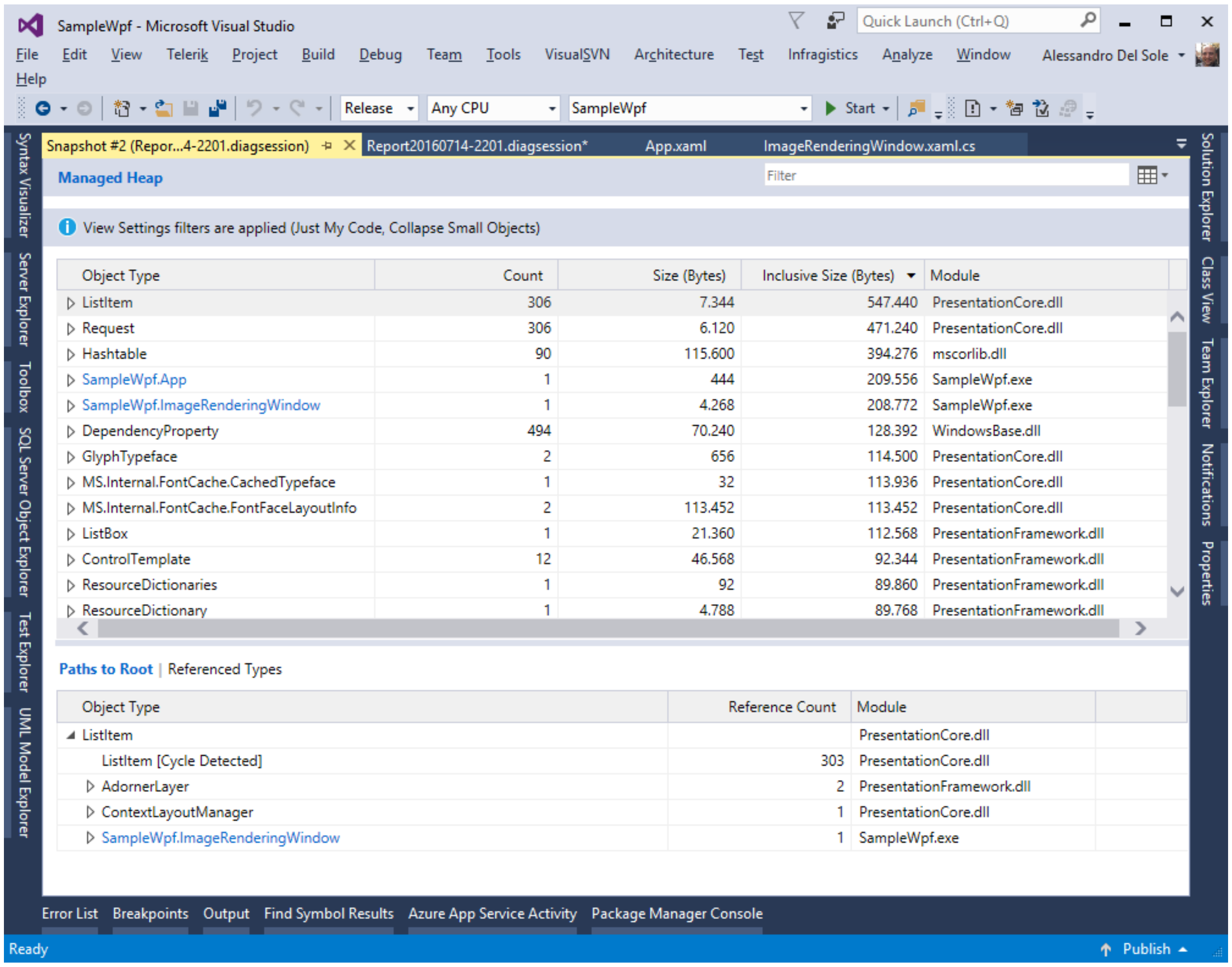
The following is a list of important considerations:

- The Diagnostic session area reports the session duration and information such as application events and invocations to the Garbage Collector. Figure 40 shows how a garbage collection, which is indicated by a red, triangular mark, has been performed five times. If you hover over a mark with the mouse, you will get a tooltip explaining the reason for the garbage collection. The tooltip is self-explanatory and makes it easier to understand the reason for each garbage collection. Too many garbage collections might be a symptom of bad memory usage.
- Every snapshot shows the size of the managed heap on the left and the number of allocated objects on the right. In the case of multiple snapshots, they also show the difference between the sizes of the managed heap and between the numbers of object instances.
- You can click the managed heap size and the list of objects in order to get detailed information, and you can compare snapshots visually.

Let's discuss this last point in detail.

Investigating the managed heap size

By clicking the managed heap size, you get a representation of the heap size at the time the snapshot was taken, as shown in Figure 41. This view focuses on the order in which objects were allocated rather than their count.


SampleWpf - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Telerik Project Build Debug Team Tools VisualSVN Architecture Test Infragistics Analyze Window Alessandro Del Sole
Help
Release
Any CPU
SampleWpf
Start
Snapshot #2 (Repor...4-2201.diagsession)
Report20160714-2201.diagsession*
App.xaml
ImageRenderingWindow.xaml.cs
Syntax Visualizer
Server Explorer
Toolbox
SQL Server Object Explorer
Test Explorer
UML Model Explorer
Solution Explorer
Class View
Team Explorer
Notifications
Properties
Filter
Error List
Breakpoints
Output
Find Symbol Results
Azure App Service Activity
Package Manager Console
Ready
Publish

Managed Heap

View Settings filters are applied (Just My Code, Collapse Small Objects)

Object Type	Count	Size (Bytes)	Inclusive Size (Bytes)	Module
ListItem	306	7.344	547.440	PresentationCore.dll
Request	306	6.120	471.240	PresentationCore.dll
Hashtable	90	115.600	394.276	mscorlib.dll
SampleWpf.App	1	444	209.556	SampleWpf.exe
SampleWpf.ImageRenderingWindow	1	4.268	208.772	SampleWpf.exe
DependencyProperty	494	70.240	128.392	WindowsBase.dll
GlyphTypeface	2	656	114.500	PresentationCore.dll
MS.Internal.FontCache.CachedTypeface	1	32	113.936	PresentationCore.dll
MS.Internal.FontCache.FontFaceLayoutInfo	2	113.452	113.452	PresentationCore.dll
ListBox	1	21.360	112.568	PresentationFramework.dll
ControlTemplate	12	46.568	92.344	PresentationFramework.dll
ResourceDictionaries	1	92	89.860	PresentationFramework.dll
ResourceDictionary	1	4.788	89.768	PresentationFramework.dll

Paths to Root | Referenced Types

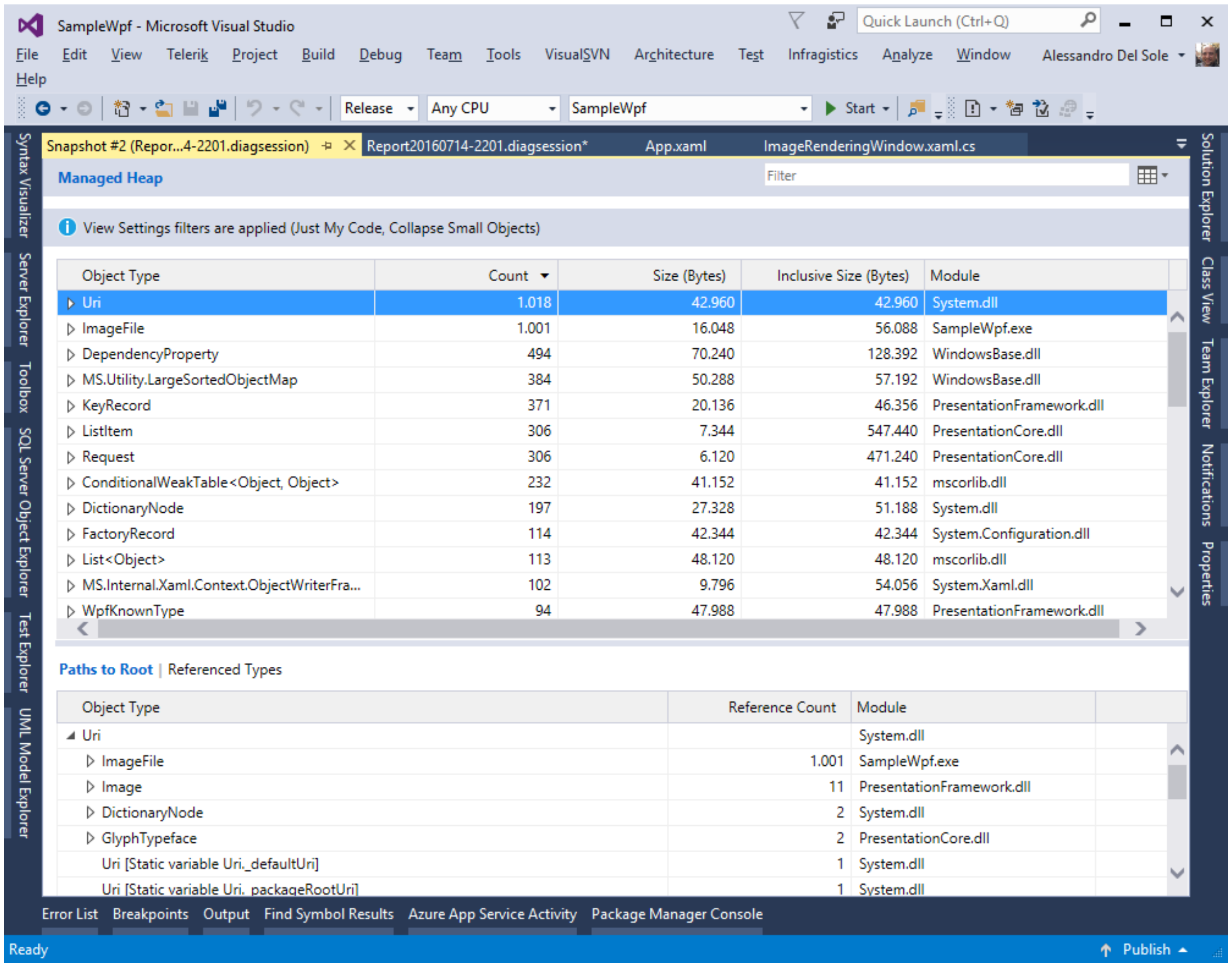
Object Type	Reference Count	Module
ListItem		PresentationCore.dll
ListItem [Cycle Detected]	303	PresentationCore.dll
AdornerLayer	2	PresentationFramework.dll
ContextLayoutManager	1	PresentationCore.dll
SampleWpf.ImageRenderingWindow	1	SampleWpf.exe

Figure 41: Objects in the Managed Heap at the Moment of a Snapshot

Among the other things, you can see that there are 306 objects for the **`ListItem`** type. Because the application is loading a large number of images in a **`ListBox`** control, this is certainly an expected behavior, so we won't get worried about it. However, this view becomes useful if you have an unexpected, large number of instantiated objects—by knowing the type, it is easier to understand if code is creating unnecessary object instances. Every object can be expanded to get more information about single instances. Notice that the report provides two columns called Size and Inclusive Size. The Size column shows the actual size of the object, whereas Inclusive Size aggregates the object size and the size of children objects. At the bottom of the view, there is a secondary grid called Paths to Root. As the name implies, it shows parent items for the selected object and the reference count. The Referenced Types tab shows you a list of types for which the selected object has a reference.

Analyzing object count

If you go back to the report and click the list of objects within a snapshot, you will get a view that is more focused on the object counts (see Figure 42).



View Settings filters are applied (Just My Code, Collapse Small Objects)

Object Type	Count	Size (Bytes)	Inclusive Size (Bytes)	Module
Uri	1.018	42.960	42.960	System.dll
ImageFile	1.001	16.048	56.088	SampleWpf.exe
DependencyProperty	494	70.240	128.392	WindowsBase.dll
MS.Utility.LargeSortedObjectMap	384	50.288	57.192	WindowsBase.dll
KeyRecord	371	20.136	46.356	PresentationFramework.dll
ListItem	306	7.344	547.440	PresentationCore.dll
Request	306	6.120	471.240	PresentationCore.dll
ConditionalWeakTable<Object, Object>	232	41.152	41.152	mscorlib.dll
DictionaryNode	197	27.328	51.188	System.dll
FactoryRecord	114	42.344	42.344	System.Configuration.dll
List<Object>	113	48.120	48.120	mscorlib.dll
MS.Internal.Xaml.Context.ObjectWriterFra...	102	9.796	54.056	System.Xaml.dll
WpfKnownType	94	47.988	47.988	PresentationFramework.dll

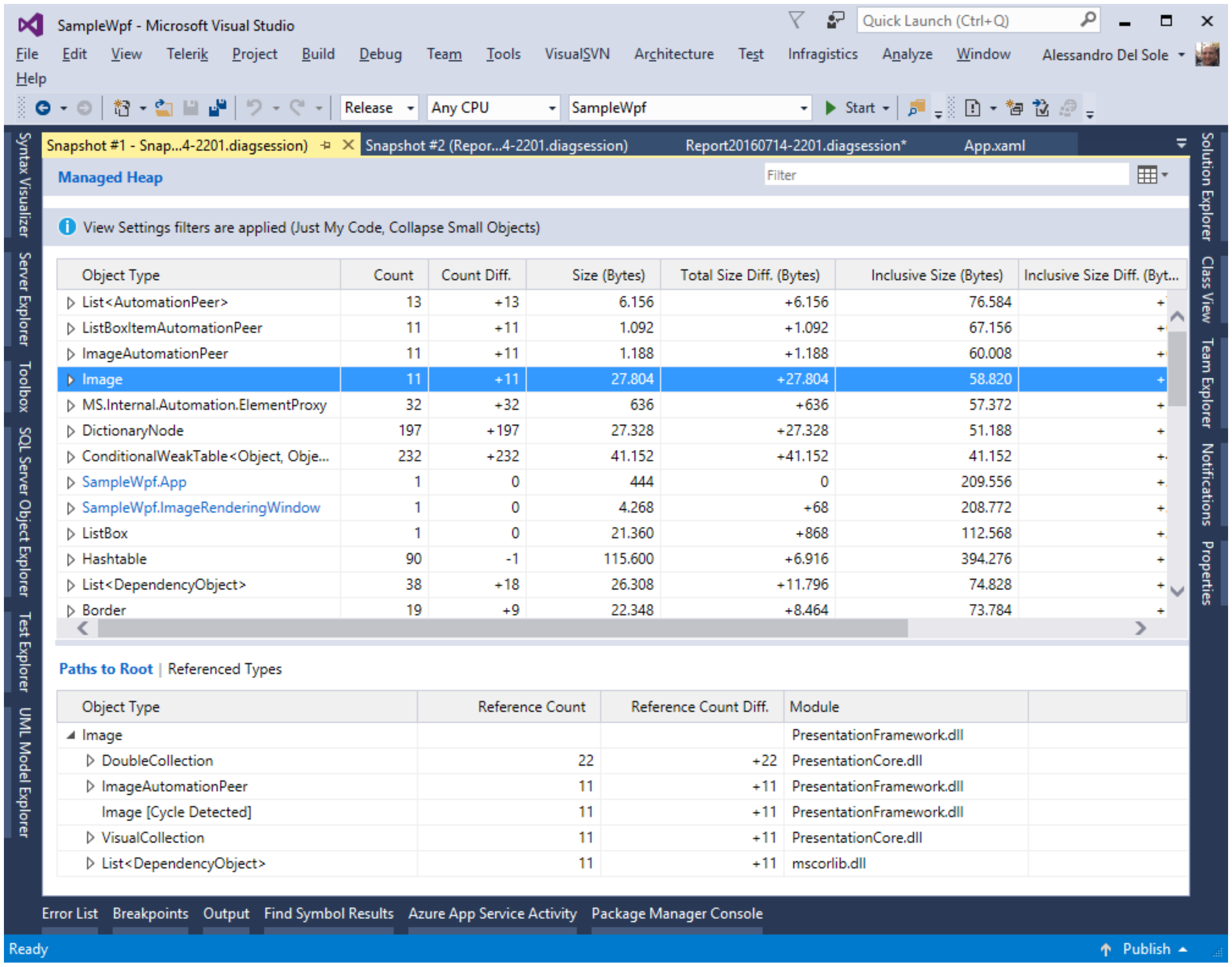
Object Type	Reference Count	Module
Uri		System.dll
ImageFile	1.001	SampleWpf.exe
Image	11	PresentationFramework.dll
DictionaryNode	2	System.dll
GlyphTypeface	2	PresentationCore.dll
Uri [Static variable Uri._defaultUri]	1	System.dll
Uri [Static variable Uri._packageRootUri]	1	System.dll

Figure 42: Object Count at Moment of Snapshot

This view gives you an immediate perception of the number of objects allocated at the time of a snapshot, but the same considerations made in the previous section apply to the objects count view, too. In this specific case, you can see that the number of instantiated objects is consistent with the large number of images the application loads, so this is no surprise. But if you see a type that was unexpectedly instantiated too many times, you have a chance to investigate your code in order to understand where and why it is creating so many instances.

Comparing snapshots

Viewing details of a single snapshot is certainly useful for understanding how memory is used at a specific point of the application lifecycle, but it is perhaps even more useful for understanding if there are more or fewer object instances between two snapshots. The Memory Usage tool provides an option to compare two snapshots very easily. In order to do this, right-click a snapshot, select **Compare To**, then pick a snapshot from the list that appears. Figure 43 shows the result of the comparison between two snapshots I took for the sample application.


SampleWpf - Microsoft Visual Studio
File Edit View Telerik Project Build Debug Team Tools VisualSVN Architecture Test Infragistics Analyze Window Help
Release
Any CPU
SampleWpf
Start
Snapshot #1 - Snap...4-2201.diagsession)
Snapshot #2 (Repor...4-2201.diagsession)
Report20160714-2201.diagsession*
App.xaml
Managed Heap
Filter
Error List Breakpoints Output Find Symbol Results Azure App Service Activity Package Manager Console
Ready
Publish

View Settings filters are applied (Just My Code, Collapse Small Objects)

Object Type	Count	Count Diff.	Size (Bytes)	Total Size Diff. (Bytes)	Inclusive Size (Bytes)	Inclusive Size Diff. (Byt...
List<AutomationPeer>	13	+13	6.156	+6.156	76.584	+
ListBoxItemAutomationPeer	11	+11	1.092	+1.092	67.156	+
ImageAutomationPeer	11	+11	1.188	+1.188	60.008	+
Image	11	+11	27.804	+27.804	58.820	+
MS.Internal.Automation.ElementProxy	32	+32	636	+636	57.372	+
DictionaryNode	197	+197	27.328	+27.328	51.188	+
ConditionalWeakTable<Object, Obje...	232	+232	41.152	+41.152	41.152	+
SampleWpf.App	1	0	444	0	209.556	+
SampleWpf.ImageRenderingWindow	1	0	4.268	+68	208.772	+
ListBox	1	0	21.360	+868	112.568	+
Hashtable	90	-1	115.600	+6.916	394.276	+
List<DependencyObject>	38	+18	26.308	+11.796	74.828	+
Border	19	+9	22.348	+8.464	73.784	+

Paths to Root | Referenced Types

Object Type	Reference Count	Reference Count Diff.	Module
Image			PresentationFramework.dll
DoubleCollection	22	+22	PresentationCore.dll
ImageAutomationPeer	11	+11	PresentationFramework.dll
Image [Cycle Detected]	11	+11	PresentationFramework.dll
VisualCollection	11	+11	PresentationCore.dll
List<DependencyObject>	11	+11	mscorlib.dll

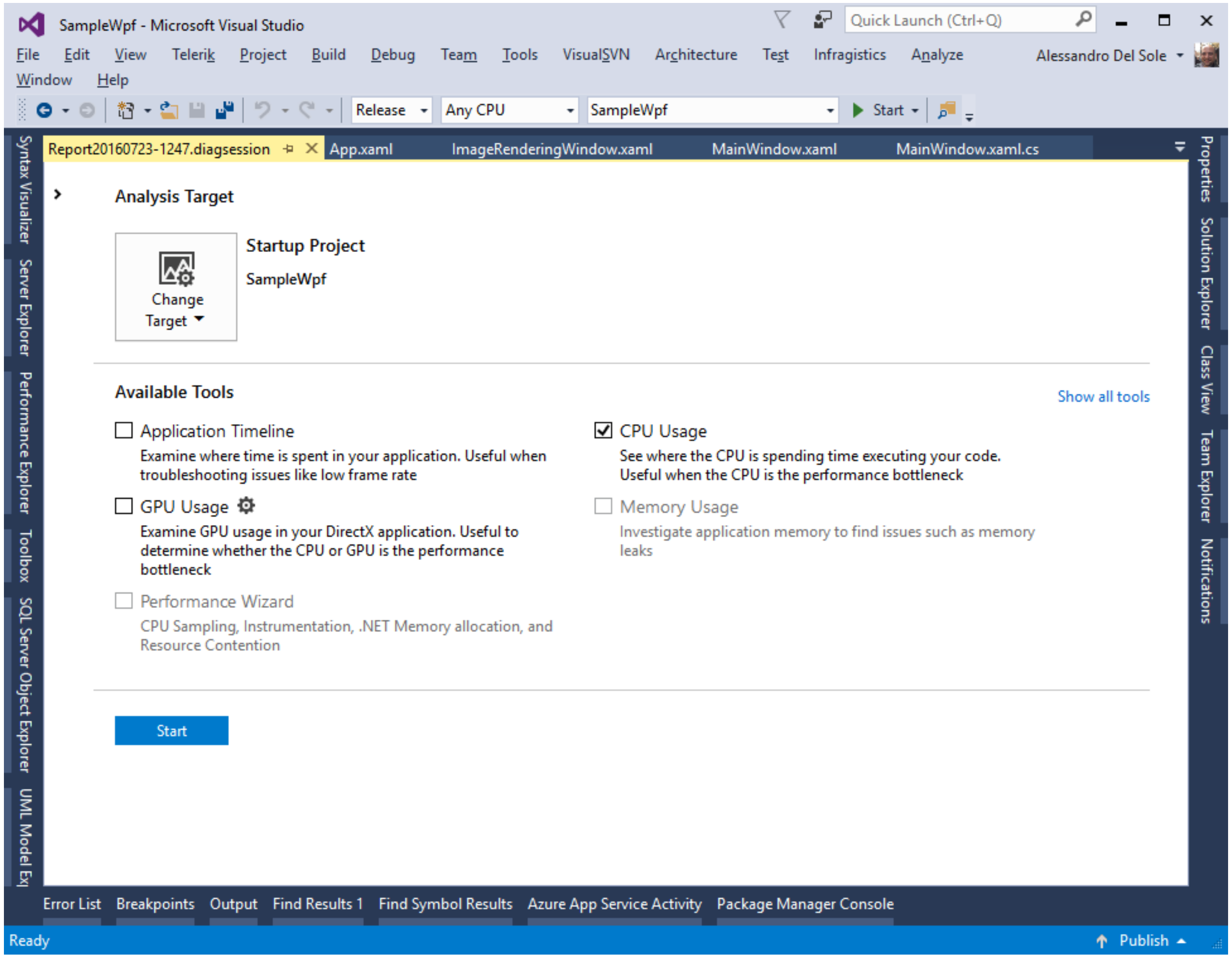
Figure 43: Comparing Snapshots

The Count column shows the number of instances of an object in the selected snapshot, while Count Diff. shows the count difference from the other snapshot. Similarly, Size and Total Size Diff. show the current total size of all the instances of an object and the difference from the previous snapshot. Finally, Inclusive Size and Inclusive Size Diff. show the current total size of the instances of an object plus children objects and also show the difference from the previous snapshot. This is probably the most important report you can get with Memory Usage because it allows you to determine if there is a normal or unexpected behavior of your objects at different intervals during the application lifecycle.

Note: Reports generated by any diagnostic tool can be saved for later analysis and comparison. Visual Studio stores reports into .diagsession files that can be reopened later inside the IDE. This is not limited to the Memory Usage tool—it is also true for all the diagnostic tools described in this chapter.

Analyzing CPU utilization

In some situations, you might want to detect where the CPU is spending time executing your code. One of the available tools in the Diagnostics Hub is the CPU Usage. As you can see in Figure 44, this tool makes it easier to analyze the CPU usage in the case of intensive work.



Analysis Target

Startup Project

SampleWpf

Available Tools

- [ ] Application Timeline
  Examine where time is spent in your application. Useful when troubleshooting issues like low frame rate
- [x] CPU Usage
  See where the CPU is spending time executing your code. Useful when the CPU is the performance bottleneck
- [ ] GPU Usage
  Examine GPU usage in your DirectX application. Useful to determine whether the CPU or GPU is the performance bottleneck
- [ ] Memory Usage
  Investigate application memory to find issues such as memory leaks
- [ ] Performance Wizard
  CPU Sampling, Instrumentation, .NET Memory allocation, and Resource Contention

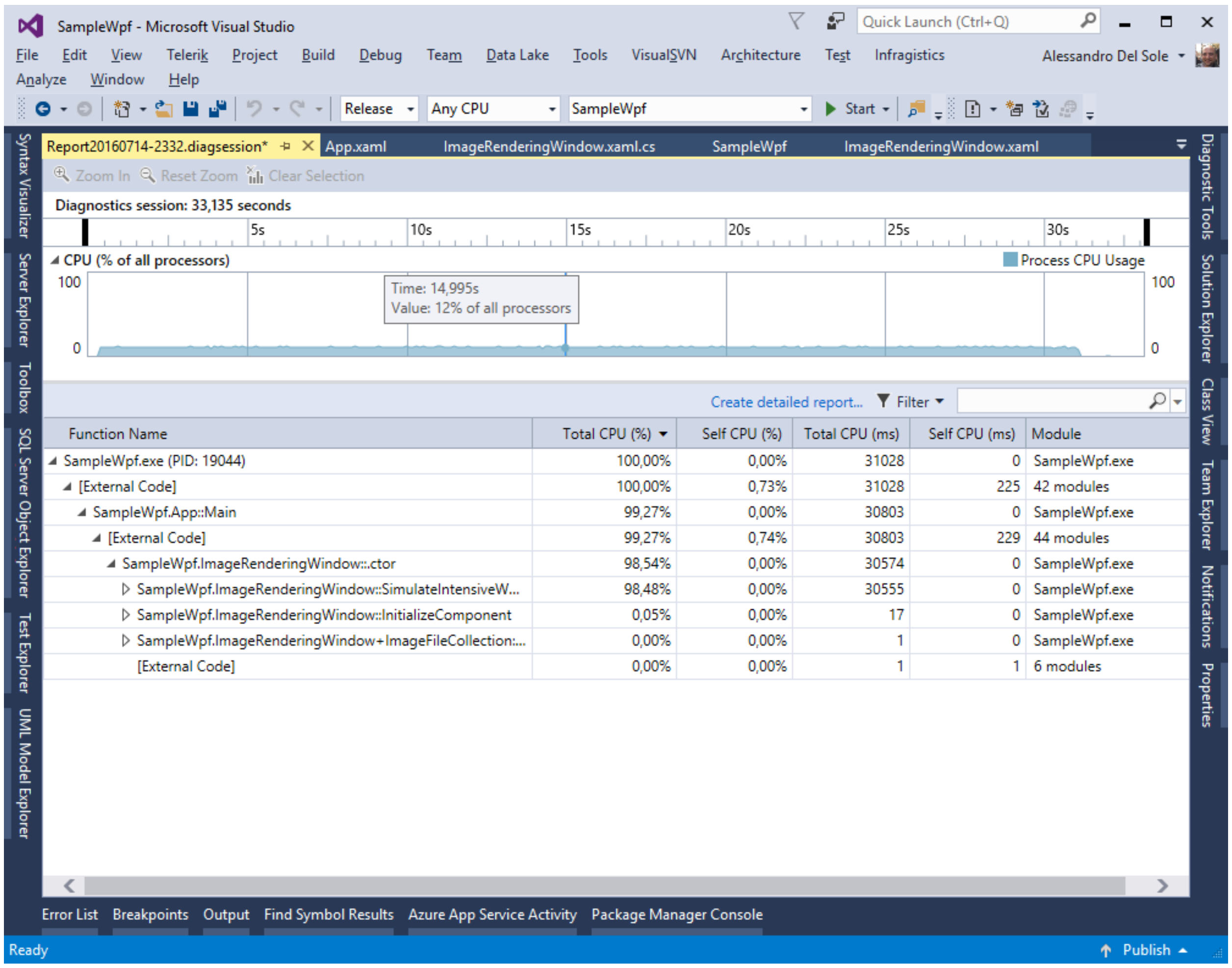
Figure 44: Enabling the CPU Usage Diagnostic Tool

Before continuing, let's add some code to the sample application that simulates intensive CPU work. Add the method shown in Code Listing 12, and make sure you invoke such a method after the **`ImageFileCollection`** assignment to the Window's **`DataContext`** property. The **`System.Threading.Thread.SpinWait`** method causes the running thread to wait for the specified milliseconds, and this is repeated within a 10,000-iterations loop with the purpose of causing CPU overhead.

Code Listing 12

```
        private void SimulateIntensiveWork()
        {
            var watch = new Stopwatch();
            watch.Start();
            for (int i=0; i < 10000; i++)
            {
                //Simulates intensive processing.
                System.Threading.Thread.SpinWait(800000);
            }
            watch.Stop();
        }
```

When ready, click **Start** in the Diagnostic Hub. You will see how Visual Studio starts reporting the CPU usage in the Live Graph. Wait for 30-40 seconds, then stop the diagnostic session. When finished, Visual Studio gives a detailed report, as shown in Figure 45.


SampleWpf - Microsoft Visual Studio
File Edit View Telerik Project Build Debug Team Data Lake Tools VisualSVN Architecture Test Infragistics
Analyze Window Help
Release
Any CPU
SampleWpf
Start
Report20160714-2332.diagsession*
App.xaml
ImageRenderingWindow.xaml.cs
SampleWpf
ImageRenderingWindow.xaml
Zoom In
Reset Zoom
Clear Selection
CPU (% of all processors)
Process CPU Usage
Time: 14,995s
Value: 12% of all processors
Create detailed report...
Filter
Error List Breakpoints Output Find Symbol Results Azure App Service Activity Package Manager Console
Ready
Publish

Diagnostics session: 33,135 seconds

Function Name	Total CPU (%)	Self CPU (%)	Total CPU (ms)	Self CPU (ms)	Module
SampleWpf.exe (PID: 19044)	100,00%	0,00%	31028	0	SampleWpf.exe
[External Code]	100,00%	0,73%	31028	225	42 modules
SampleWpf.App::Main	99,27%	0,00%	30803	0	SampleWpf.exe
[External Code]	99,27%	0,74%	30803	229	44 modules
SampleWpf.ImageRenderingWindow::.ctor	98,54%	0,00%	30574	0	SampleWpf.exe
SampleWpf.ImageRenderingWindow::SimulateIntensiveW...	98,48%	0,00%	30555	0	SampleWpf.exe
SampleWpf.ImageRenderingWindow::InitializeComponent	0,05%	0,00%	17	0	SampleWpf.exe
SampleWpf.ImageRenderingWindow+ImageFileCollection:...	0,00%	0,00%	1	0	SampleWpf.exe
[External Code]	0,00%	0,00%	1	1	6 modules

Figure 45: Investigating the CPU Usage

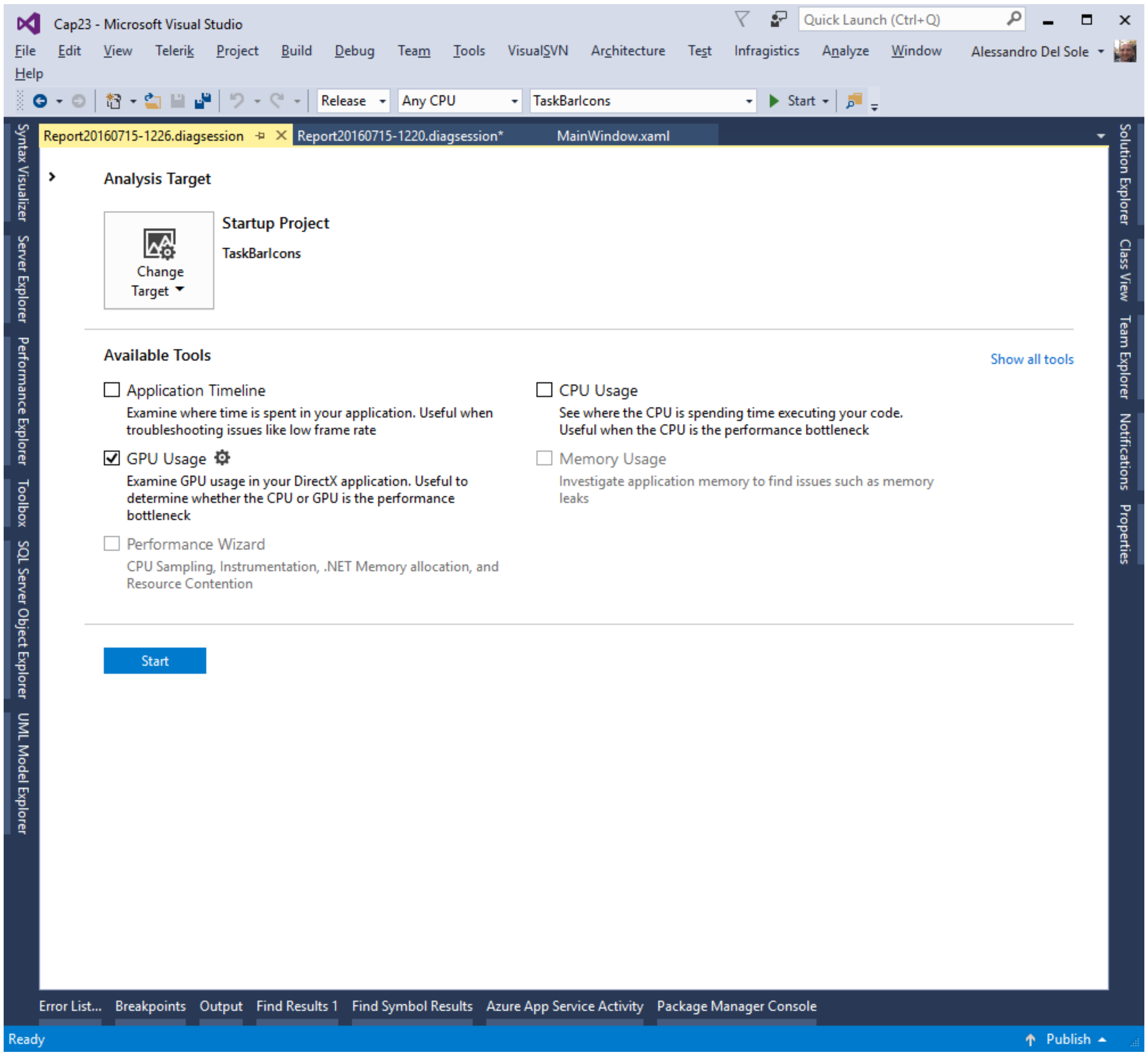
At the top, the report shows the duration of the diagnostic session and the CPU utilization during the application lifecycle. At the bottom, you can see a list of method calls, including constructors and external code, and the CPU usage they caused. In this particular case, the **`SimulateIntensiveWork`** method caused the more intensive work for the CPU. The report shows five columns:

- Total CPU (%), which shows the percentage of usage caused by the selected function and the functions it called.
- Self CPU (%), which shows the percentage of usage caused by the selected function, excluding the functions it called.
- Total CPU (ms), which shows the time in milliseconds the CPU was busy because of the selected function and the functions it called.
- Self CPU (ms), which shows the time in milliseconds the CPU was busy because of the selected function, excluding the functions it called.
- Module, which shows the component name that contains the selected function or the number of external modules referenced.

As you can easily imagine, the more a function causes the CPU to be busy, the more it should be analyzed in code to see if it is performing expected, extensive work or if a bottleneck has occurred.

Analyzing GPU performances

The Graphics Processing Unit (GPU) is the video card on your machine that makes it possible to render anything you see on screen, from text and windows, to videos and images. In the case of applications that make intensive usage of the GPU, especially with media and games, you can leverage a diagnostic tool called GPU Usage, which is available in the Diagnostics Hub (see Figure 46).


Cap23 - Microsoft Visual Studio
File Edit View Telerik Project Build Debug Team Tools VisualSVN Architecture Test Infragistics Analyze Window Help
Release
Any CPU
TaskBarIcons
Start
Report20160715-1226.diagsession
Report20160715-1220.diagsession*
MainWindow.xaml
Change Target
Show all tools
Start
Error List... Breakpoints Output Find Results 1 Find Symbol Results Azure App Service Activity Package Manager Console
Ready
Publish

Analysis Target

Startup Project

TaskBarIcons

Available Tools

☐ Application Timeline
Examine where time is spent in your application. Useful when troubleshooting issues like low frame rate

☑ GPU Usage
Examine GPU usage in your DirectX application. Useful to determine whether the CPU or GPU is the performance bottleneck

☐ Performance Wizard
CPU Sampling, Instrumentation, .NET Memory allocation, and Resource Contention

☐ CPU Usage
See where the CPU is spending time executing your code. Useful when the CPU is the performance bottleneck

☐ Memory Usage
Investigate application memory to find issues such as memory leaks

Figure 46: Enabling GPU Usage

This tool's primary purpose is analyzing applications that heavily use the DirectX graphic libraries. Though WPF invokes DirectX behind the scenes, you will get limited information from GPU Usage unless you code 3-D graphics and animations. However, you can definitely get information about the GPU utilization in any WPF application that works with videos, animations, and, more generally, with media content.

Tip: The goal of this e-book is to cover common WPF scenarios, not specific development contexts such as gaming or 3-D graphics. If you want to see a more specific example of GPU usage with WPF, you can check out the [Walkthrough: Hosting Direct3D9 Content in WPF]() on MSDN.

When you click **Start**, Visual Studio begins collecting information about the GPU utilization that is immediately reported in the Live Graph. Figure 47 shows a sample report based on a WPF application while playing a video.

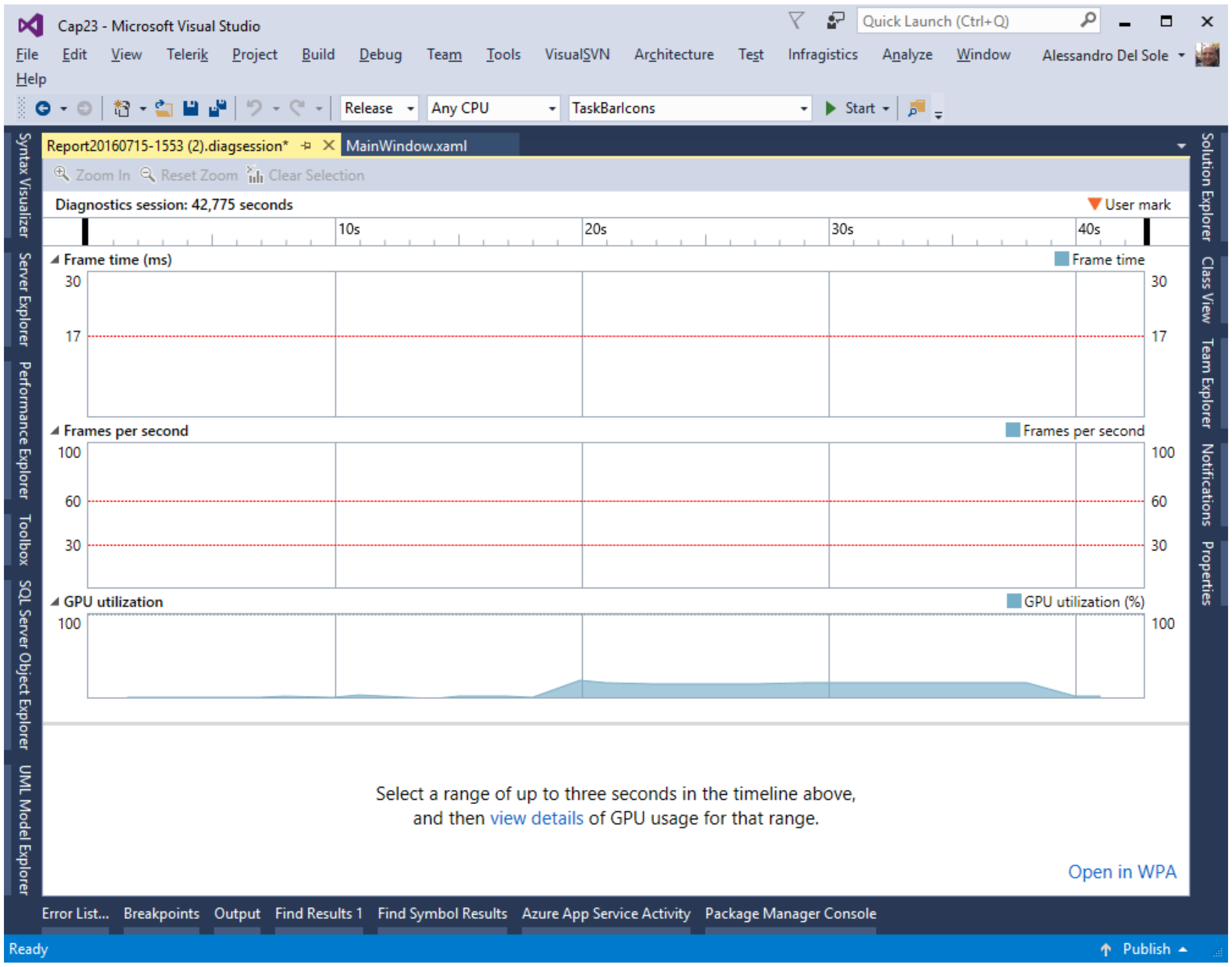
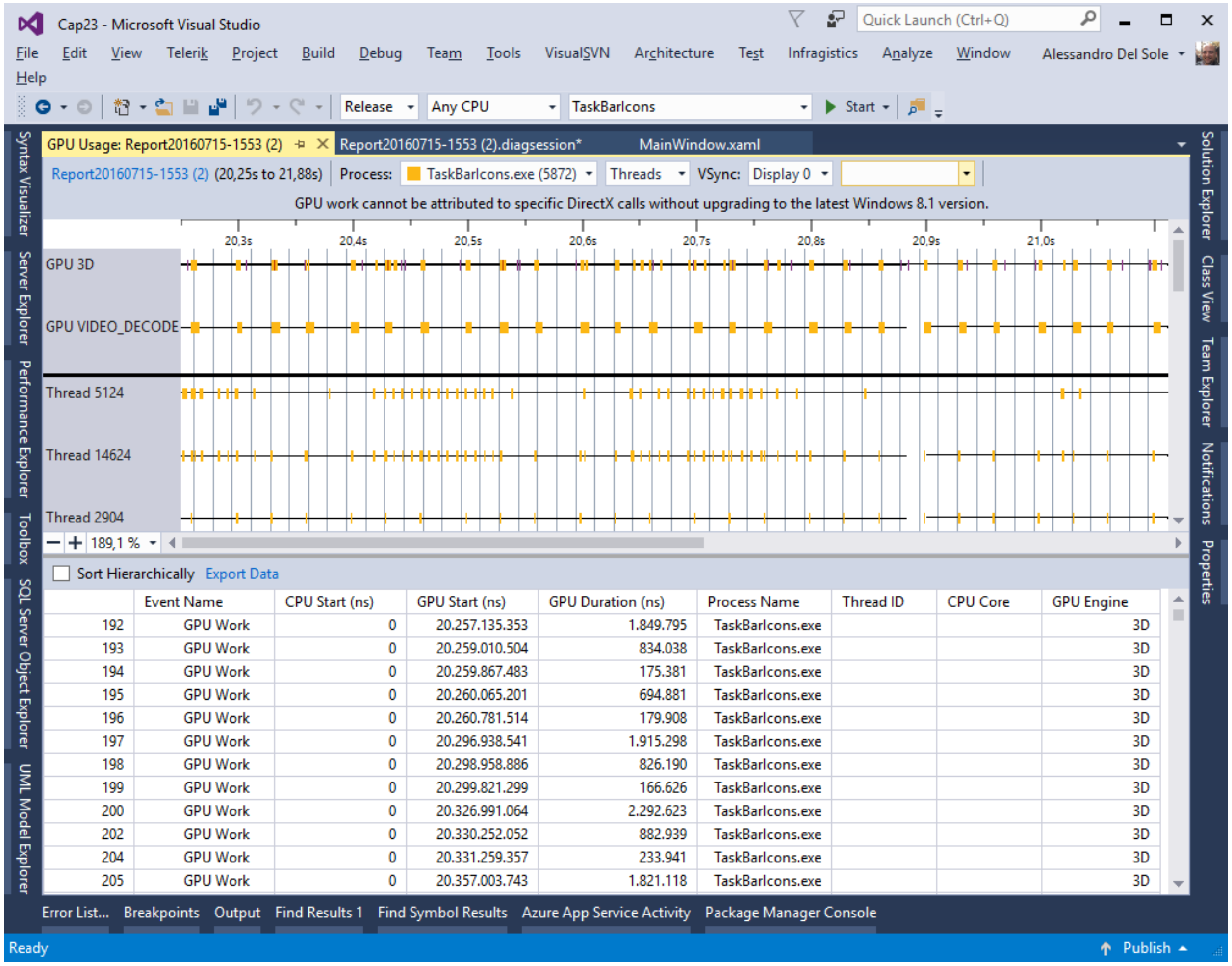


Figure 47: GPU Utilization in Action

The Diagnostic session area still reports the duration of the application lifecycle. With DirectX utilization, you can see the frame time in milliseconds and the frame rate per second at a given time. In this case, you only get information in the GPU Utilization area. The utilization is due to the video playing. Before you can analyze details for GPU utilization, you need to select a small interval of time, up to three seconds, then click the **view details** hyperlink. At this point, you will see a detailed report for the GPU utilization, as shown in Figure 48. If your code uses DirectX directly, you will get details for each marker. In this case, you can see how the GPU has been busy in decoding a video during the selected time frame, and you can see the threads involved in the work. You can hover over each marker to see additional details (if available). At the bottom, there is a list of events; in this case, all events are generically called GPU Work, but would be more detailed in the case of direct calls to the DirectX libraries.



Report20160715-1553 (2) (20,25s to 21,88s) Process: TaskBarIcons.exe (5872) Threads VSync: Display 0

GPU work cannot be attributed to specific DirectX calls without upgrading to the latest Windows 8.1 version.

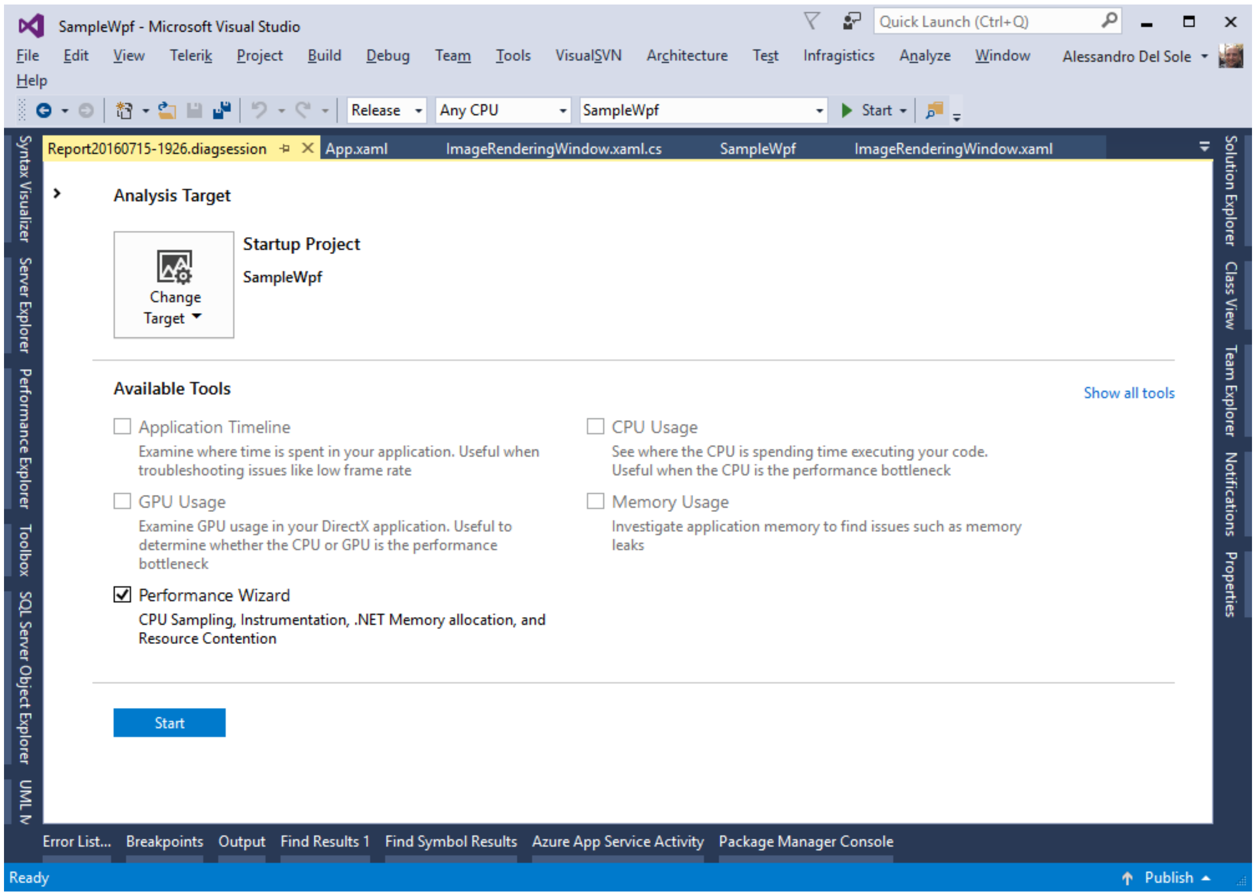
Sort Hierarchically Export Data

	Event Name	CPU Start (ns)	GPU Start (ns)	GPU Duration (ns)	Process Name	Thread ID	CPU Core	GPU Engine
192	GPU Work	0	20.257.135.353	1.849.795	TaskBarIcons.exe			3D
193	GPU Work	0	20.259.010.504	834.038	TaskBarIcons.exe			3D
194	GPU Work	0	20.259.867.483	175.381	TaskBarIcons.exe			3D
195	GPU Work	0	20.260.065.201	694.881	TaskBarIcons.exe			3D
196	GPU Work	0	20.260.781.514	179.908	TaskBarIcons.exe			3D
197	GPU Work	0	20.296.938.541	1.915.298	TaskBarIcons.exe			3D
198	GPU Work	0	20.298.958.886	826.190	TaskBarIcons.exe			3D
199	GPU Work	0	20.299.821.299	166.626	TaskBarIcons.exe			3D
200	GPU Work	0	20.326.991.064	2.292.623	TaskBarIcons.exe			3D
202	GPU Work	0	20.330.252.052	882.939	TaskBarIcons.exe			3D
204	GPU Work	0	20.331.259.357	233.941	TaskBarIcons.exe			3D
205	GPU Work	0	20.357.003.743	1.821.118	TaskBarIcons.exe			3D

Figure 48: Reporting the GPU Work

Hints about the Performance Wizard

Visual Studio 2015 inherits from its predecessors a profiling tool known as Performance Wizard. This tool has been updated with a modern user interface, but it performs analysis sessions that are essentially similar to the Memory Usage and CPU Utilization, plus has a couple of diagnostic tools that I will describe shortly. Behind the scenes, the Performance Wizard relies on the Visual Studio profiler called VsPerf.exe. This is a command-line tool that can run on machines on which Visual Studio is not installed, such as servers. Performance Wizard is enabled in the Diagnostic Hub, as shown in Figure 49.



Analysis Target

Startup Project

SampleWpf

Available Tools

☐ Application Timeline
Examine where time is spent in your application. Useful when troubleshooting issues like low frame rate

☐ CPU Usage
See where the CPU is spending time executing your code. Useful when the CPU is the performance bottleneck

☐ GPU Usage
Examine GPU usage in your DirectX application. Useful to determine whether the CPU or GPU is the performance bottleneck

☐ Memory Usage
Investigate application memory to find issues such as memory leaks

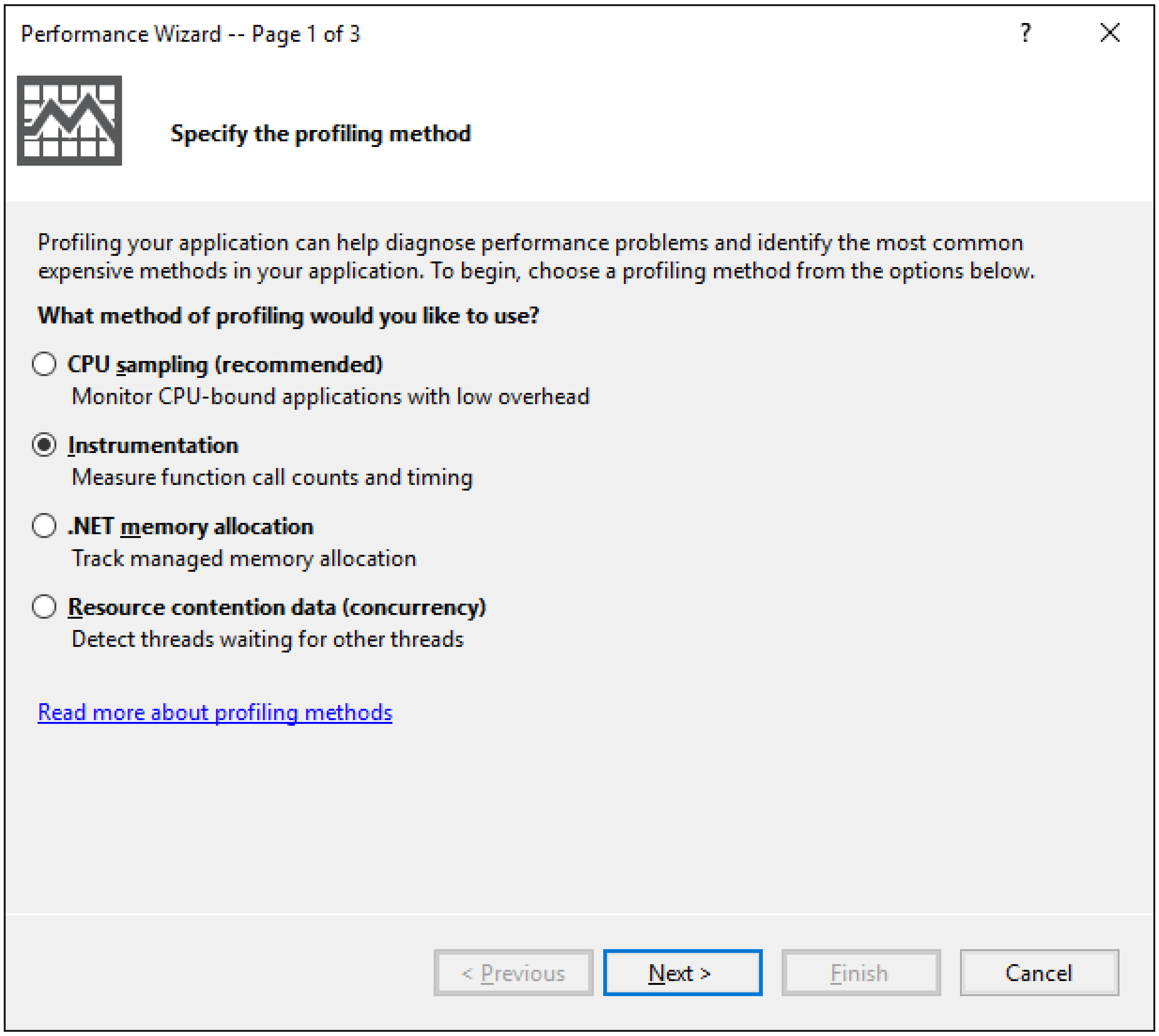
☑ Performance Wizard
CPU Sampling, Instrumentation, .NET Memory allocation, and Resource Contention

Figure 49: Enabling the Performance Wizard

The Performance Wizard offers the following diagnostic tools:

- **CPU Sampling**, which allows for analyzing the CPU usage.
- **Instrumentation**, which measures function call counts and timing.
- **.NET memory allocation**, which tracks managed memory allocation.
- **Resource contention data**, which is useful for detecting threads waiting for other threads.

I will not discuss **CPU Sampling** and **.NET memory allocation** here; you have seen how to use the Memory Usage and CPU Usage, which are more recent tools that target Windows Store apps as well, while the Performance Wizard does not. When you click Start, Visual Studio starts the Performance Wizard by asking you to select the profiling method (see Figure 50).



Performance Wizard -- Page 1 of 3

Specify the profiling method

Profiling your application can help diagnose performance problems and identify the most common expensive methods in your application. To begin, choose a profiling method from the options below.

What method of profiling would you like to use?

○ **CPU sampling (recommended)**
Monitor CPU-bound applications with low overhead

● **Instrumentation**
Measure function call counts and timing

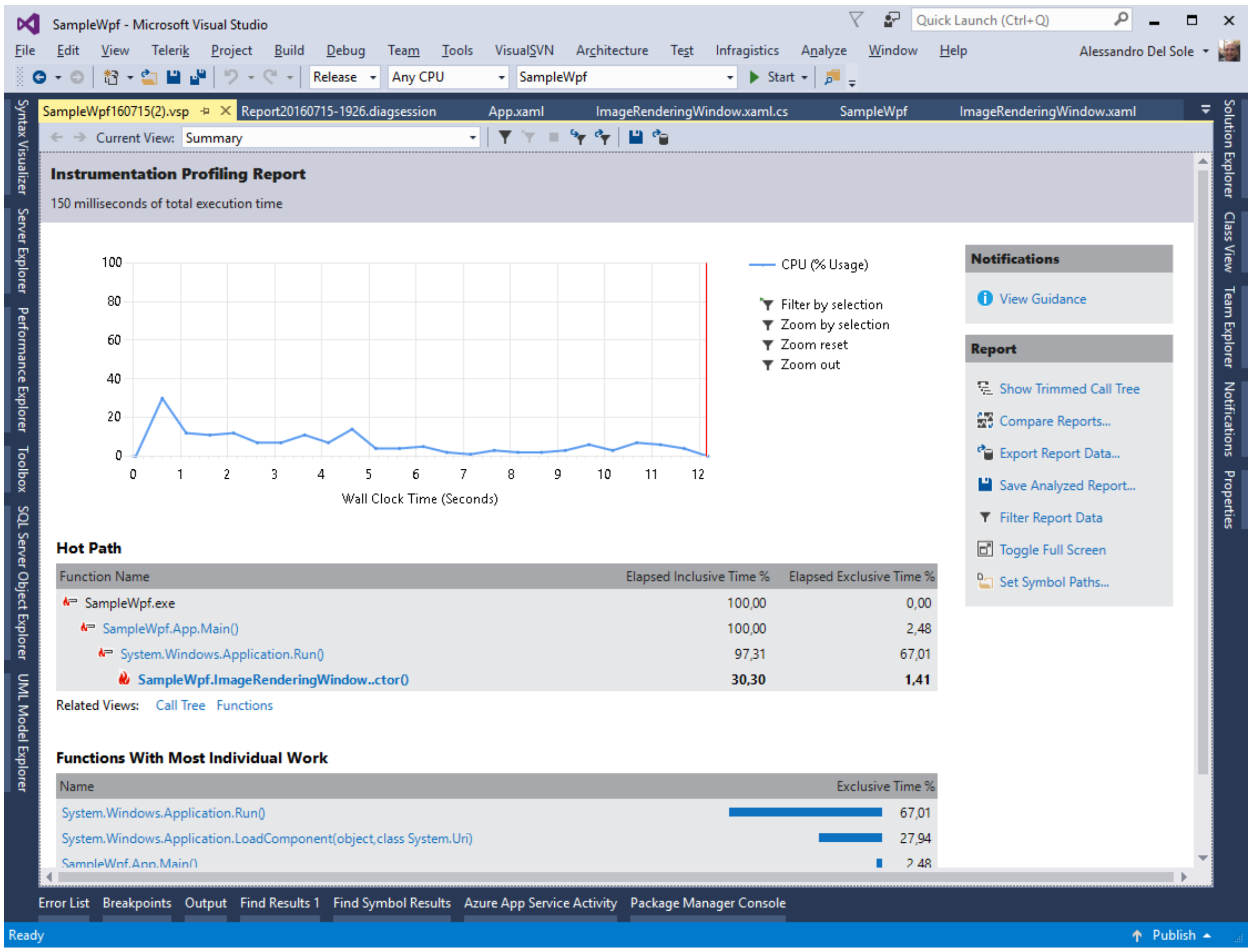
○ **.NET memory allocation**
Track managed memory allocation

○ **Resource contention data (concurrency)**
Detect threads waiting for other threads

Read more about profiling methods

Figure 50: Specifying the Profiling Method

Select **Instrumentation**, which is useful for checking for functions doing the most individual work. When you click **Next**, you will be asked to specify the analysis target. You can leave the default selection unchanged on the current project, but you could also specify a different .exe file, an ASP.NET application, and even a .dll library. Complete the Wizard and start profiling. While the application is running, Visual Studio collects information about function calls. You can end the diagnostic session simply by closing the application, and you will get a detailed report of the functions doing the most individual work, as shown in Figure 51.


SampleWpf - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Telerik Project Build Debug Team Tools VisualSVN Architecture Test Infragistics Analyze Window Help
Alessandro Del Sole
Release
Any CPU
SampleWpf
Start
SampleWpf160715(2).vsp Report20160715-1926.diagsession App.xaml ImageRenderingWindow.xaml.cs SampleWpf ImageRenderingWindow.xaml
Current View: Summary
CPU (% Usage)
Filter by selection
Zoom by selection
Zoom reset
Zoom out
Wall Clock Time (Seconds)
Notifications
View Guidance
Report
Show Trimmed Call Tree
Compare Reports...
Export Report Data...
Save Analyzed Report...
Filter Report Data
Toggle Full Screen
Set Symbol Paths...
Error List Breakpoints Output Find Results 1 Find Symbol Results Azure App Service Activity Package Manager Console
Ready
Publish
Syntax Visualizer Server Explorer Performance Explorer Toolbox SQL Server Object Explorer UML Model Explorer
Solution Explorer Class View Team Explorer Notifications Properties

Instrumentation Profiling Report

150 milliseconds of total execution time

Hot Path

Function Name	Elapsed Inclusive Time %	Elapsed Exclusive Time %
SampleWpf.exe	100,00	0,00
SampleWpf.App.Main()	100,00	2,48
System.Windows.Application.Run()	97,31	67,01
SampleWpf.ImageRenderingWindow..ctor()	**30,30**	**1,41**

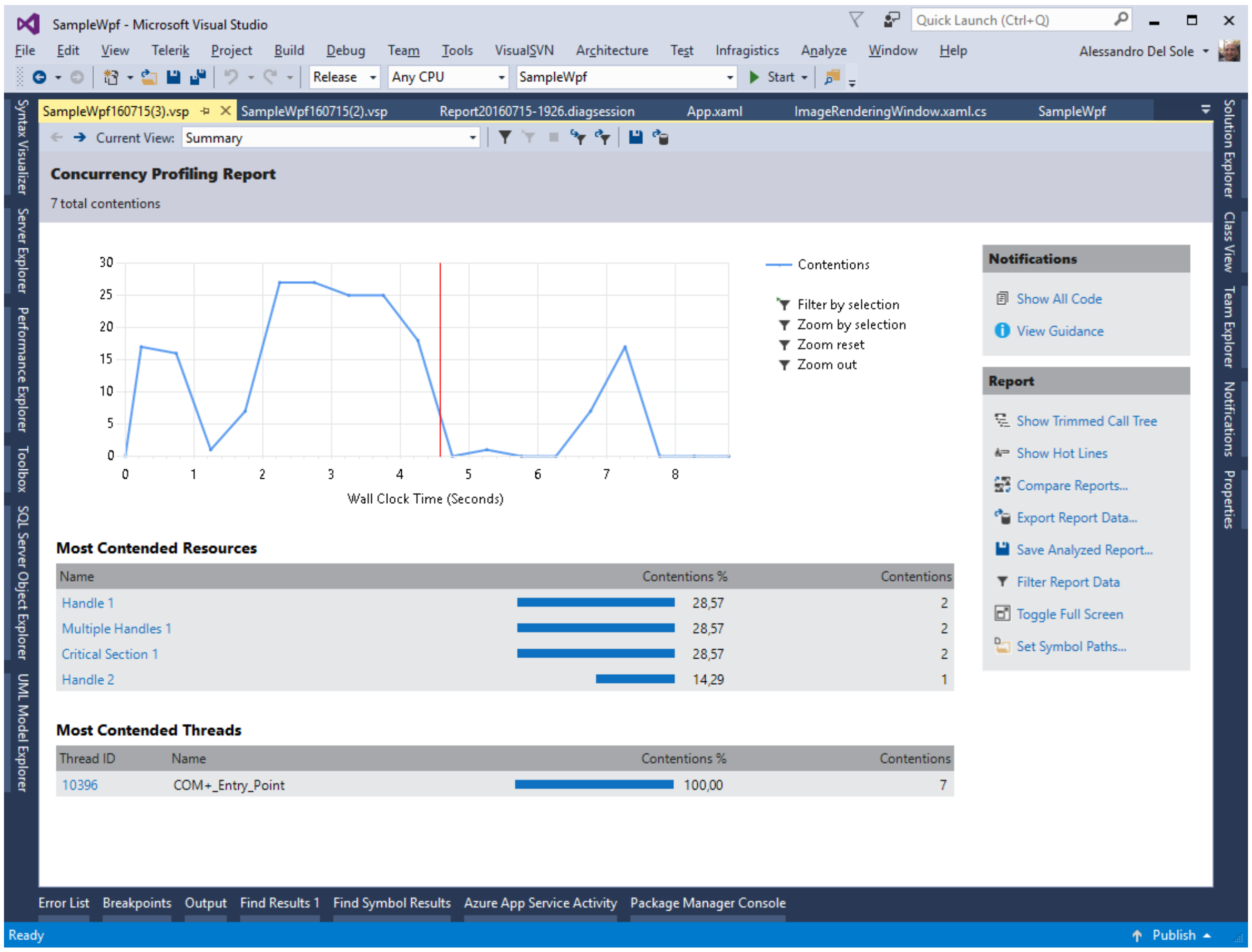
Related Views: Call Tree Functions

Functions With Most Individual Work

Name	Exclusive Time %
System.Windows.Application.Run()	67,01
System.Windows.Application.LoadComponent(object,class System.Uri)	27,94
SampleWpf.App.Main()	2,48

Figure 51: Functions Doing the Most Individual Work

The **Resource contention data** tool shows information about thread concurrency—threads waiting for other threads—with most contended resources and most contended threads, as represented in Figure 52.



Concurrency Profiling Report

7 total contentions

Most Contended Resources

Name	Contentions %	Contentions
Handle 1	28,57	2
Multiple Handles 1	28,57	2
Critical Section 1	28,57	2
Handle 2	14,29	1

Most Contended Threads

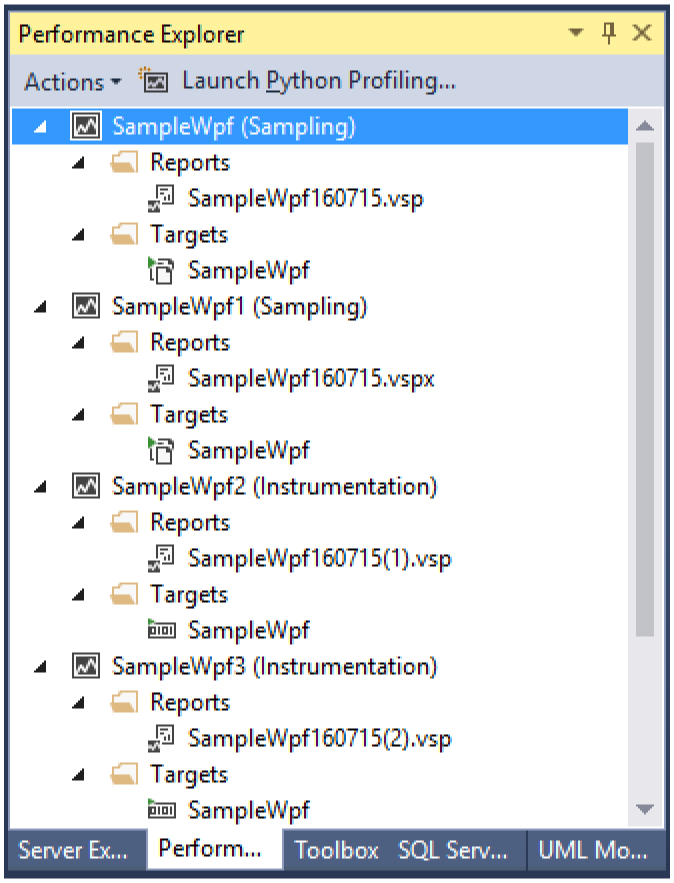
Thread ID	Name	Contentions %	Contentions
10396	COM+_Entry_Point	100,00	7

Figure 52: Understanding Thread Concurrency and Contended Resources

Of course, this tool will be particularly useful with applications that create multiple threads to do some of their work.

The Performance Explorer window

Each time you start a profiling session with the Performance Wizard, Visual Studio collects and organizes reports into a convenient view offered by the Performance Explorer tool window, which should be visible automatically and that you can also enable by selecting Debug, Profiler, Performance Explorer, Show Performance Explorer (see an example in Figure 53).



```
SampleWpf (Sampling)
    Reports
        SampleWpf160715.vsp
    Targets
        SampleWpf
SampleWpf1 (Sampling)
    Reports
        SampleWpf160715.vspx
    Targets
        SampleWpf
SampleWpf2 (Instrumentation)
    Reports
        SampleWpf160715(1).vsp
    Targets
        SampleWpf
SampleWpf3 (Instrumentation)
    Reports
        SampleWpf160715(2).vsp
    Targets
        SampleWpf
```

Figure 53: The Performance Explorer Tool Window

As you can see, Performance Explorer provides a categorized view of the profiling sessions based on their target and profile. You can simply double-click a report file to open the corresponding viewer. You can also compare reports by right-clicking a report name, then selecting Compare Performance Reports.

Chapter summary

Analyzing a WPF application performance is an important task, and Visual Studio demonstrates once again how powerful it is by offering a number of diagnostic tools. With the Memory Usage tool, you can investigate how your application uses memory and have an easier time of discovering memory leaks. With the CPU Usage tool, you can check if your application is performing unexpected CPU-intensive work. With the GPU Usage tool, you can analyze how your application is consuming GPU resources. With the Performance Wizard, you have specialized tools to investigate function calls and thread concurrency.