

1. .NET Reflector 6 documentation	2
1.1 Getting started	3
1.2 Using .NET Reflector	4
1.2.1 Running .NET Reflector	5
1.2.2 About the graphical user interface	6
1.2.3 Configuring .NET Reflector	12
1.2.4 Working with code	13
1.2.5 Using assemblies	14
1.2.6 Managing the debug store	15
1.2.7 Navigating in .NET Reflector	16
1.2.8 Using bookmarks	18
1.2.9 Working with Add-ins	19
1.3 The Visual Studio add-in for .NET Reflector Pro	20
1.3.1 Installing the Visual Studio add-in	22
1.3.2 Using the Visual Studio add-in	25
1.3.3 Working with assemblies	26
1.3.4 Debugging an executable	28
1.3.5 Deactivating the serial number	29
1.4 Upgrading	30
1.5 Troubleshooting	31
1.5.1 Log files for .NET Reflector Pro	32
1.5.2 Methods showing no source code in the disassembly window	33
1.5.3 Permissions error on starting Visual Studio	34
1.5.4 The XP Bug Workaround	35
1.6 Tips and articles	36
1.6.1 .NET Reflector Tips - keyboard shortcuts	37
1.6.2 Debugging a SharePoint customization	39
1.6.3 Debugging into SharePoint and seeing the locals	42
1.6.4 Introduction to building .NET Reflector add-ins	49
1.7 Release notes and other versions	51
1.7.1 .NET Reflector 7.7 release notes	52
1.7.2 .NET Reflector 7.6 release notes	53
1.7.3 .NET Reflector 7.5 release notes	54
1.7.4 .NET Reflector 7.4 release notes	55
1.7.5 .NET Reflector 7.3 release notes	56
1.7.6 .NET Reflector 7.2 release notes	57
1.7.7 .NET Reflector 7.1 release notes	58
1.7.8 .NET Reflector 7.0 release notes	59
1.7.9 .NET Reflector 6.5 release notes	60
1.7.10 .NET Reflector Pro 6.5 release notes	61
1.7.11 .NET Reflector and .NET Reflector Pro 6.1 release notes	62
1.7.12 .NET Reflector 6.0 release notes	63
1.7.13 .NET Reflector Pro 6.0 release notes	64

.NET Reflector 6 documentation

About .NET Reflector

With .NET Reflector you can decompile and debug .NET components, such as assemblies, and disassemble the source code into your chosen .NET language, enabling you to see the contents of a .NET assembly.

For more information, see the [.NET Reflector product page](#).

Quick links

[Getting started](#)

[Running .NET Reflector](#)

[Walkthrough: Debugging a SharePoint customization](#)

[.NET Reflector forums](#)

Getting started

With .NET Reflector you can decompile and debug .NET components, such as assemblies, and disassemble the source code into your chosen .NET language, enabling you to see the contents of a .NET assembly.

You can use .NET Reflector to search for bugs and performance issues that may be affecting your system, and to review areas of code that were not previously accessible. For example, you may have an application that uses a third-party component, which is returning unexpected results. With .NET Reflector, you can disassemble and debug the third-party component, helping you to determine whether the component is responsible for the results.

.NET Reflector is compatible with:

- **.NET Framework** versions: 1.0*, 1.1*, 2.0, 3.0, 3.5, 4.0
 - * .NET Reflector runs under these framework versions, but the Visual Studio add-in and .NET Reflector Pro require .NET 2.0 or later.
- **Microsoft Windows** operating systems (32-bit and 64-bit): Windows 2000
 - *, Windows XP SP2 or later, Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7
 - * .NET Reflector only; not the Visual Studio add-in or .NET Reflector Pro.
- **Microsoft Visual Studio** versions: 2005, 2008, 2010

You should ensure that your system meets the minimum requirements for the version of Microsoft Visual Studio you want to run.

You can use .NET Reflector to:

- [Disassemble and analyze code](#)
- [Manage assembly lists](#)
- [Bookmark items within an assembly](#)

You can use .NET Reflector Pro to:

- [Select assemblies to debug within Visual Studio](#)
- [Generate debug information](#)
- [Debug executables](#)

Useful examples and links

Follow these links for more help information about .NET Reflector:

- [Walkthrough of .NET Reflector on Simple Talk](#).
- [Details on .NET Reflector add-ins \(CodePlex\)](#). These can be used to add functionality. Further information on these is available on [Simple Talk](#).
- [Frequently asked questions \(Simple Talk\) about .NET Reflector](#).
- [Hosting .NET Reflector \(Simple Talk\) in your own application](#).
- [Using the Methodist add-in \(Simple Talk\) to interact with your own code](#).

Case studies:

- [.NET Reflector Saved their Bacon: Chris Kapilla's Story \(Simple Talk\)](#)
- [.NET Reflector Saved their Bacon: The Gremlins Strike Back \(Simple Talk\)](#)

Using .NET Reflector

- [Running .NET Reflector](#)
- [About the graphical user interface](#)
- [Configuring .NET Reflector](#)
- [Working with code](#)
- [Using assemblies](#)
- [Managing the debug store](#)
- [Navigating in .NET Reflector](#)
- [Using bookmarks](#)
- [Working with Add-ins](#)

Running .NET Reflector

You can use .NET Reflector to open an assembly by doing one of the following:

- Open .NET Reflector, and then drag the assembly from Windows Explorer onto .NET Reflector. You can also open .NET Reflector directly from Windows Explorer.
See [Setting Integration Options](#) for details.

- Open .NET Reflector, and on the **File** menu, click **Open**; or on the toolbar, click

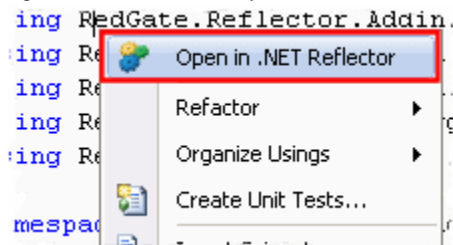


- Invoke .NET Reflector from the command line:
`Reflector.exe [options] [assemblies]`
You can use the following options:

<code>/register</code>	Register Visual Studio and Windows Explorer integration
<code>/unregister</code>	Unregister Visual Studio and Windows Explorer integration
<code>/select:<identifier></code>	Select item in browser
<code>/fontname:<name></code>	Use specified font name
<code>/fontsize:<size></code>	Use specified font size
<code>/configuration:<file></code>	Use configuration file name
<code>/help</code>	Show this help message

For example: `C:\>Reflector.exe /select:"System.Uri" System.dll`

- Right-click on the assembly in Visual Studio and select **Open in .NET Reflector**.

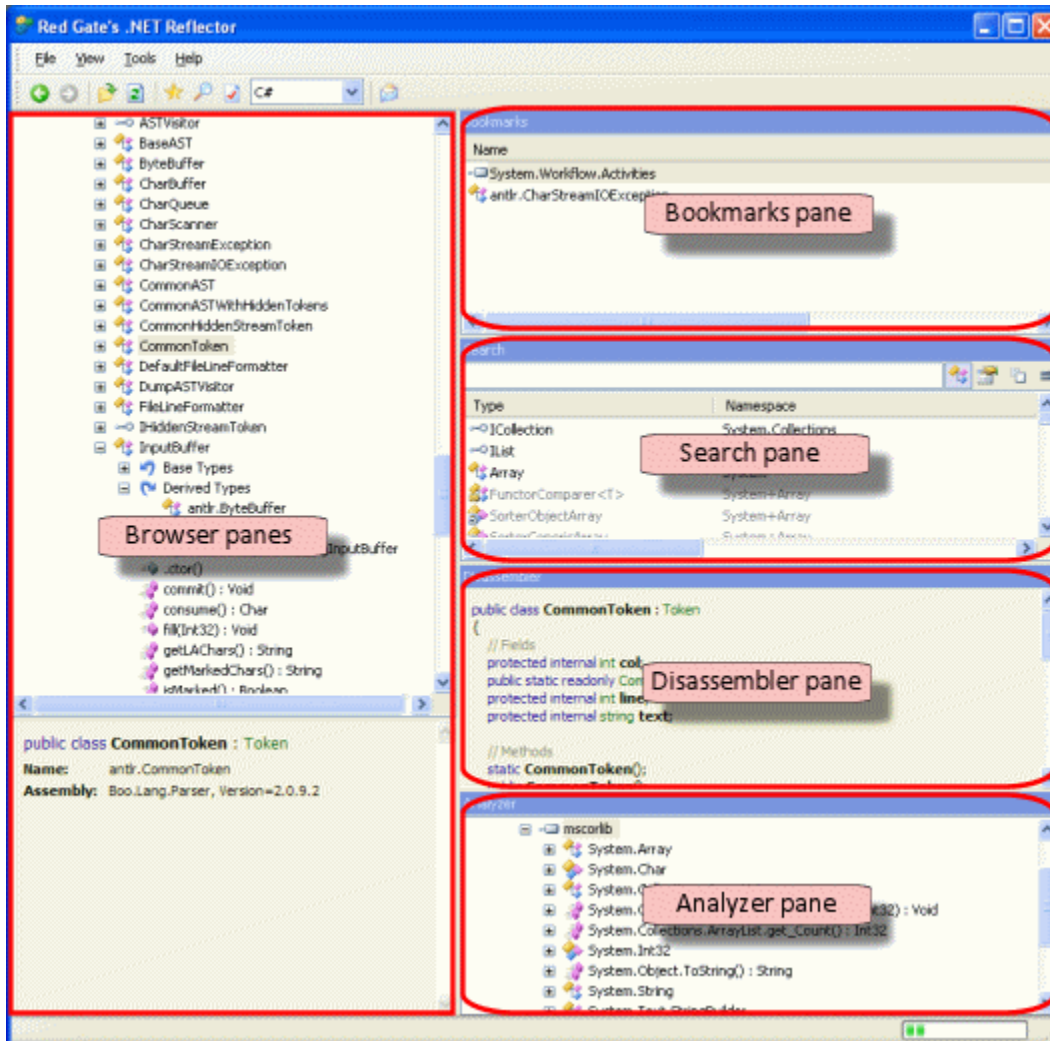


About the graphical user interface

The .NET Reflector graphical user interface enables you to navigate, decompile, and analyze assemblies. It also provides features and tools to help you manage your code.

To start the graphical user interface, see [Running .NET Reflector](#).

Once opened, .NET Reflector displays the main window:



Initially, only the **Browser** panes are shown, which are used to navigate to the assembly you are interested in. Other panes are opened when you perform a particular operation or action:

- **Bookmarks** pane - enables you to bookmark an assembly.
See [Using bookmarks](#) for details.
- **Search** pane - displays the search criteria and the results of any search.
See [Navigating in .NET Reflector](#) for details.
- **Disassembler** pane - shows the disassembled code for the selected assembly.
See [Disassembling code](#) for details.
- **Analyzer** pane - shows the code dependencies.
See [Analyzing code](#) for details.

You can choose to leave the panes open, that way they are refreshed automatically, when required. Alternatively, you can close them until the next time the operation is performed.

The various panes in .NET Reflector contain entities that are displayed in a tree structure. To expand an item, click

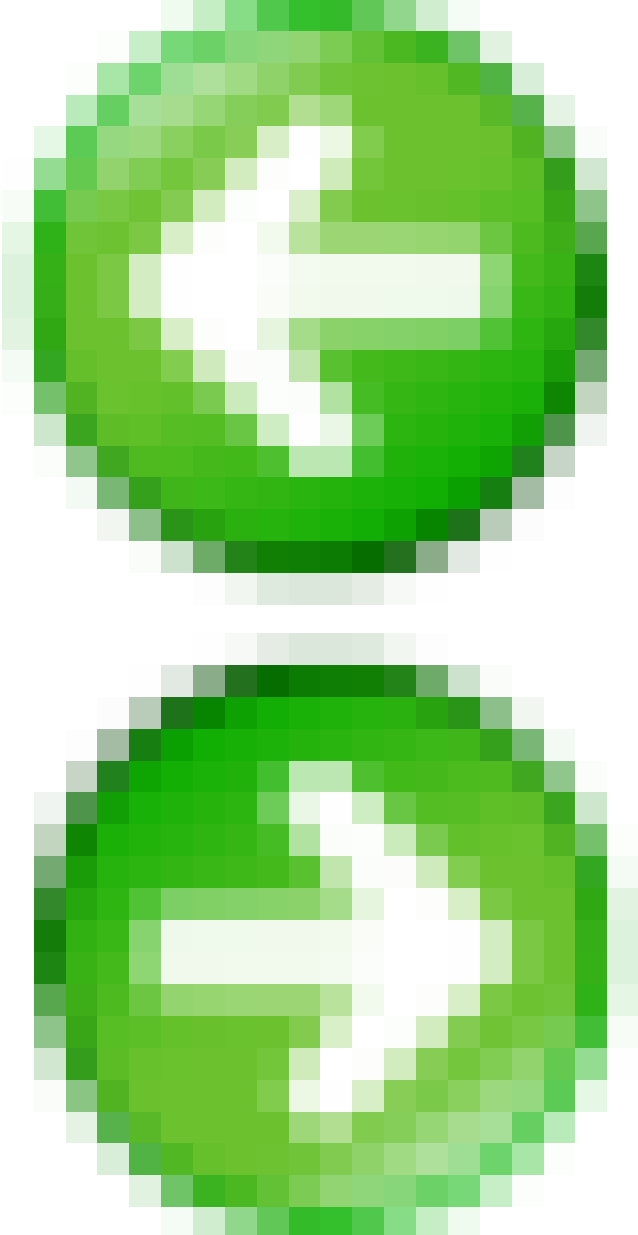


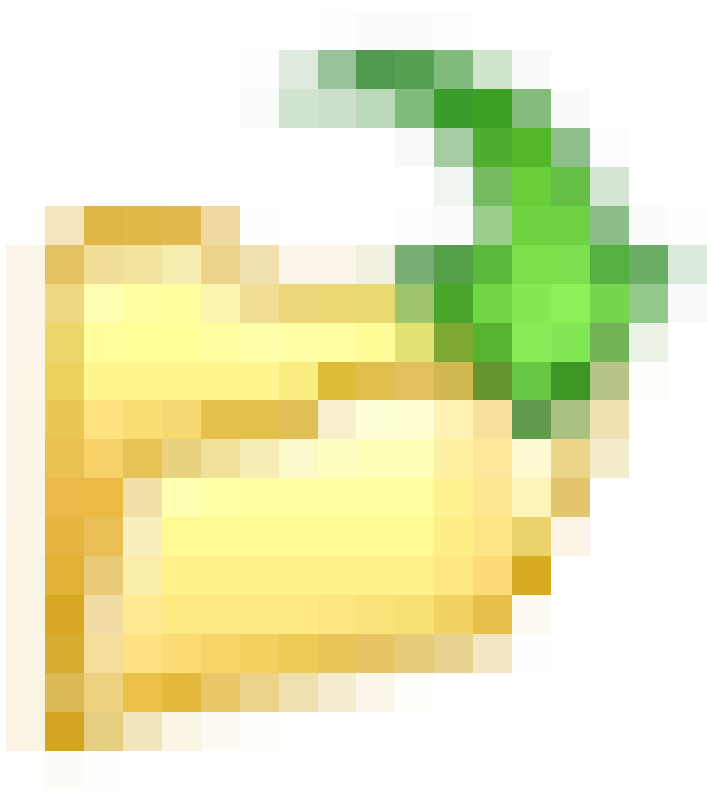
. To collapse an item, click



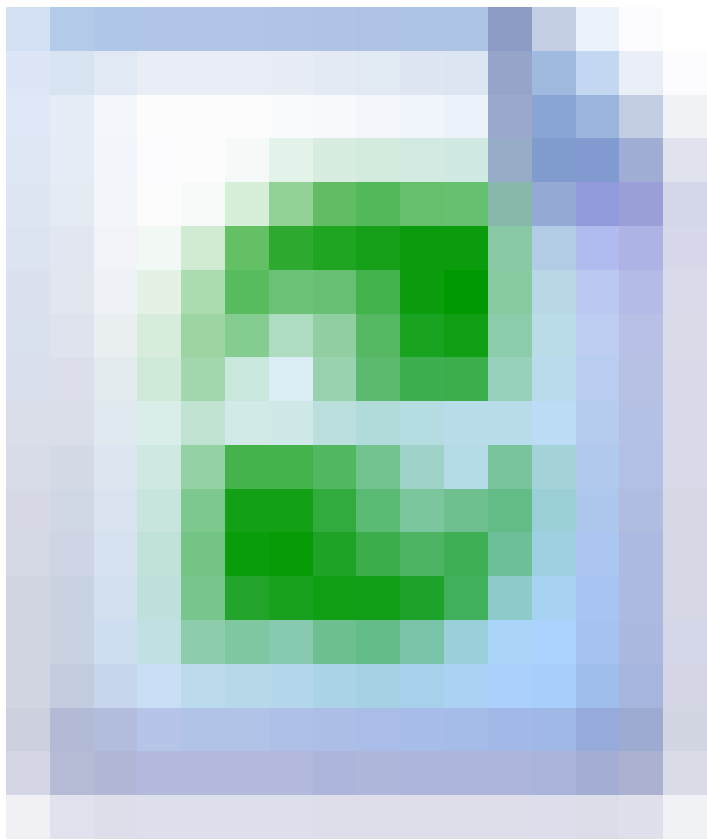
Using the toolbar

The following options are shown on the toolbar:

	<p>Navigate the history</p>
--	-----------------------------



Open an assembly



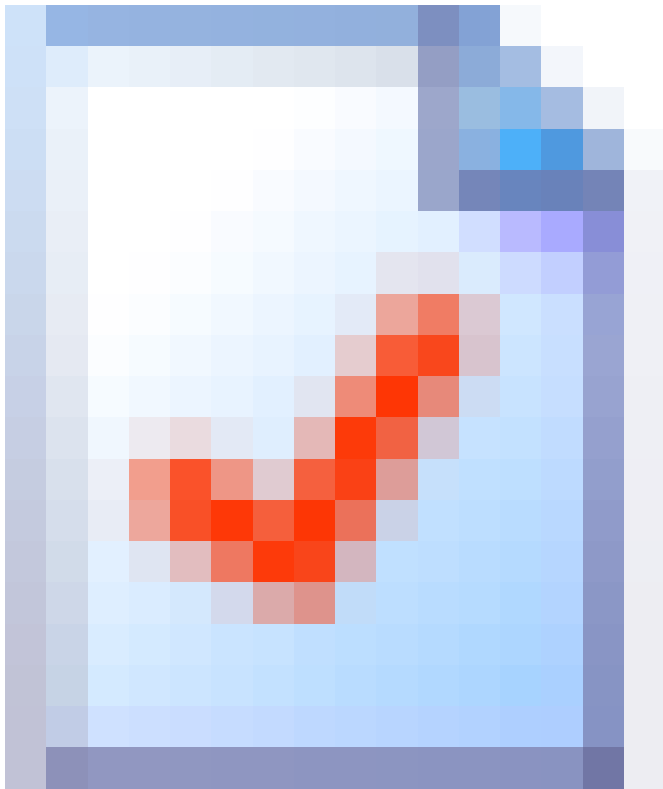
Refresh the assembly list. This updates the tree structure containing the assemblies, but also closes all other panes.



Open the **Bookmarks** pane



Open the **Search** pane



Open the **Options** dialog box

C#



Select the disassembly language



Provide feedback

Using the menu options

The following actions can only be performed using the menu options, and are not available as icons on the toolbar:

- From the **File** menu, you can open the assembly cache, manage the assembly list, and unload an assembly from the browser tree structure. You can also exit .NET Reflector.
- From the **View** menu, you can manage any .NET Reflector add-ins.
- From the **Tools** menu, you can open the the **Disassembler** and **Analyzer** panes, view the integration options, and search specialized websites (Bing and MSDN).
- From the **Help** menu, you can access and view help information on .NET Reflector. You can also check to make sure that you are using the latest version of .NET Reflector.

Configuring .NET Reflector

Setting integration options

The integration options are used to add extensions to .NET Reflector, enabling integration with Visual Studio and Windows Explorer.

To access the integration options, on the **Tools** menu, click **Integration Options**. The **Integration Options** dialog box is displayed.

Managing the Visual Studio add-in

You can use the **Integration Options** dialog box to control whether the Visual Studio add-in is used. This enables you to open .NET Reflector directly from Visual Studio. With this add-in you can right-click on an assembly within Visual Studio and select **Open in .NET Reflector**.

If you have installed .NET Reflector Pro, see [Installing the Visual Studio add-in](#) for further details.

Managing the Windows Explorer add-in

The Windows Explorer add-in enables you to open .NET Reflector directly from Windows Explorer. You can use Windows Explorer to navigate to a particular EXE or DLL file, then right-click and select **Browse with .NET Reflector**. You can also double-click on DLL files within Windows Explorer to open them directly within .NET Reflector.

Setting .NET Reflector options

You can use the .NET Reflector options to alter how .NET Reflector is used:

- **Disassembler** - controls the language and formatting that the code is disassembled into and which version of .NET is used for optimization. You can also use this option to control how the documentation is displayed.
- **Browser** - controls how the code is displayed in the browser and which items are visible. For example, you can select whether to view only public members and types, or everything.
- **Appearance** - controls the font used for the code and the browser. For example, you can view code with a fixed-width font, rather than the default variable-width font.
- **Proxy Server** - enables you to select a proxy server to use.

To access the .NET Reflector options, on the **View** menu, click **Options**; or on the toolbar, click



Working with code

Disassembling code

Using .NET Reflector, you can disassemble your assemblies back into the high-level language that produced them, or into a different language.

To disassemble code using .NET Reflector:

1. Open .NET Reflector and select the assembly you want to disassemble. See [Running .NET Reflector](#) for details on how to open an assembly.
2. On the toolbar, select the language you want to disassemble into.
3. On the **Tools** menu, click **Disassemble**. You can also do this by pressing SPACE.
The code for the assembly is shown in the **Disassembler** pane. You can now navigate through the code.

The disassembled code may not be perfect, because some symbolic information is often lost in the compilation. For further details on why some code may not have been decompiled correctly, see [this Simple Talk support article](#).

Analyzing code

The analyzer is used to show code dependencies and what is exposed by, instantiated by, and assigned by the class.

To analyze the code:

1. Select the class you want to analyze.
2. Right-click and choose **Analyze**. The **Analyzer** pane shows the relationship between the class and the rest of the code.

Using assemblies

.NET Reflector can disassemble process assemblies (EXE) and library assemblies (DLL).

Using assembly lists

You can create groups of assemblies and then open and browse them as a set. The list of assemblies is stored in the configuration file, **Reflector.cfg**, and is loaded the next time you open .NET Reflector. You can create as many assembly lists as you want.

To open an assembly list:

1. On the **File** menu, click **Open List**.
The **Assembly List Management** dialog box is displayed.
2. Choose the assembly list and click **Select**.

You can also use the **Assembly List Management** dialog box to add a new assembly list. This creates a copy of the current list, which can then be modified. You can also remove assembly lists using this dialog box.

To open an assembly from the Global Assembly Cache (GAC):

1. On the **File** menu, click **Open Cache**.
The **Assembly Cache** dialog box is displayed. You can then open a particular assembly directly from the cache.
2. Select the assembly then click **Open**.
The new assembly is added to the assembly list. You may need to minimize any currently expanded assemblies to see the new entry.

You might want to refresh the assembly list, for example, if you have made changes in an open assembly. To refresh the assemblies list, On the **View** menu, click **Refresh**; or on the toolbar, click



. All assemblies in the list are reloaded.

It is not possible to refresh an individual assembly. If you have a large number of assemblies in the list then you should only refresh when necessary.

Managing the debug store

.NET Reflector generates a store of decompiled code and debug symbols on your hard drive (a set of directories in your local settings), enabling you to step-through debug the selected assemblies. The store retains the decompiled code and symbols, so that you do not need to decompile every time.

To delete the contents of the store:

1. Open the **Debug and Decompile** dialog box.
See [Decompiling assemblies](#) for details on how to do this.
2. Click **Clear Store**.
Depending on the size of the store, it may take a few minutes to be clear the store.

You will need to reselect and decompile any assemblies that were previously debugged.

Navigating in .NET Reflector

There are several ways to navigate in .NET Reflector. You can use menu options to open and close panes, and then expand and collapse hierarchies, enabling you to drill down to a particular code element. From here you can click on the element to display more information about it, and view the source code in the **Disassembler** pane. If you click on a link within the source code you are immediately taken to the location that it references.

You can use



Back to return to a previous stage in your navigation. This is particularly useful if you have navigated via a series of links in the source code. Once you have traced your steps back, you can then click



Forward to advance through the navigation history.

Using the search tool

You can search for a type, member, or item across all the loaded assemblies. On the **View** menu, click **Search**; or on the toolbar, click



The **Search** pane is displayed, and if previously opened in the session, shows the last search criteria.

To search, first select the search preference:



	search a member
	search an item

You can then enter the search string. To specify an exact match, click



. The current list is filtered as you type the string. If the list contains a large number of entries then it may take several seconds to refresh.

If you select an alternative search preference, with the string field populated, the new list is automatically filtered.

Searching BING and MSDN

To provide full details on the assemblies, you can access [MSDN](#) and [Bing](#) directly from .NET Reflector.

You need to select the assembly, then on the **Tools** menu, click either **Search Bing** or **Search MSDN**. If your selection is not applicable to one of the sites, the option is unavailable on the **Tools** menu.

Using bookmarks

You can bookmark any item within an assembly. This is done by saving a custom URI for the item, which you can click to display the item in .NET Reflector.

To view existing bookmarks:

1. On the **View** menu, click **Bookmarks**; or on the toolbar, click



The **Bookmarks** pane is displayed.

2. Double-click the bookmark to see the disassembled code.

To add (toggle on) a bookmark:

1. On the **View** menu, click **Bookmarks**.
The Bookmarks pane is displayed. The list is empty if no bookmarks have previously been added (toggled on).
2. Highlight the element in the assembly, right-click and choose **Toggle Bookmark**.
The URI for the new bookmark is shown in the **Bookmarks** pane.

To remove a bookmark (toggle off), you can select the element in the list that has the bookmark associated with it, right-click and choose **Toggle Bookmark**. This removes the associated bookmark. You can also remove a bookmark by selecting the bookmark, right-clicking and selecting **Remove Bookmark**.

To copy an existing bookmark:

You can copy a bookmark directly from the **Bookmarks** pane. You can then paste the bookmark somewhere for future reference, or share with other developers; for example, by pasting the link into an e-mail. You can only share the URI with other developers who have .NET Reflector installed and have access to the associated DLL or EXE file.

Working with Add-ins

A number of add-ins have been written for .NET Reflector to extend its functionality. Most of these add-ins are open-source, and are available via the [Codeplex](#) website.

To view the current add-ins, on the **View** menu, click **Add-ins**. The **Add-ins** dialog box is displayed showing you a list of the installed add-ins. You can install new add-ins and remove existing add-ins from this dialog box.

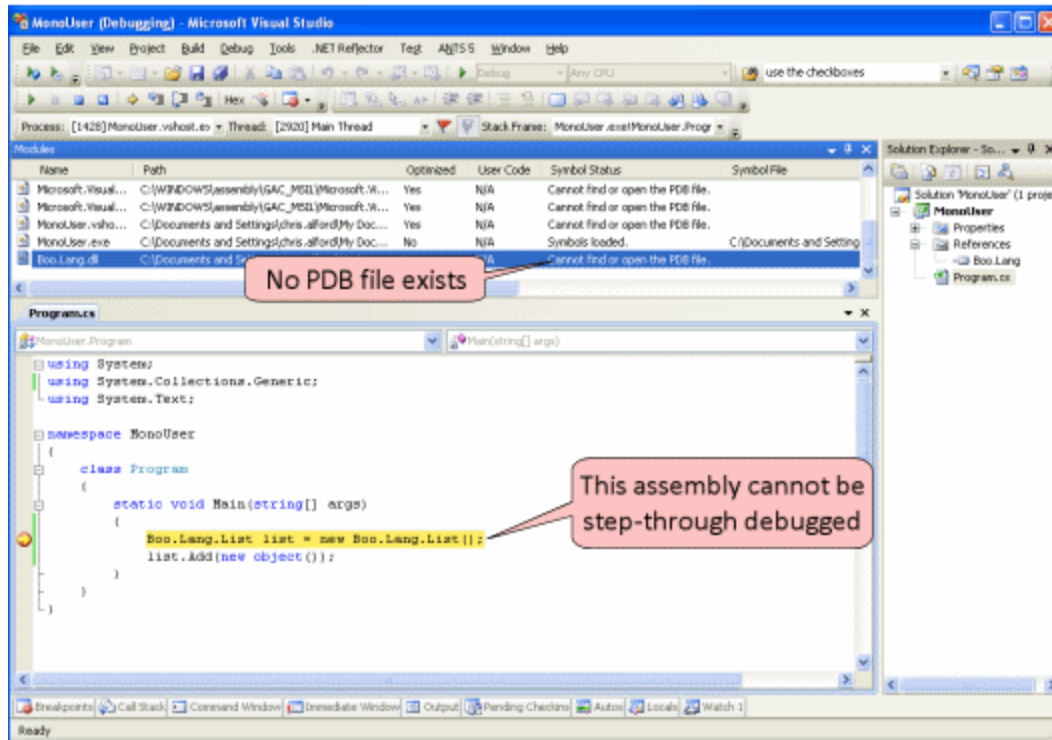
You can find information on how to install the add-ins on [Simple Talk](#).

The Visual Studio add-in for .NET Reflector Pro

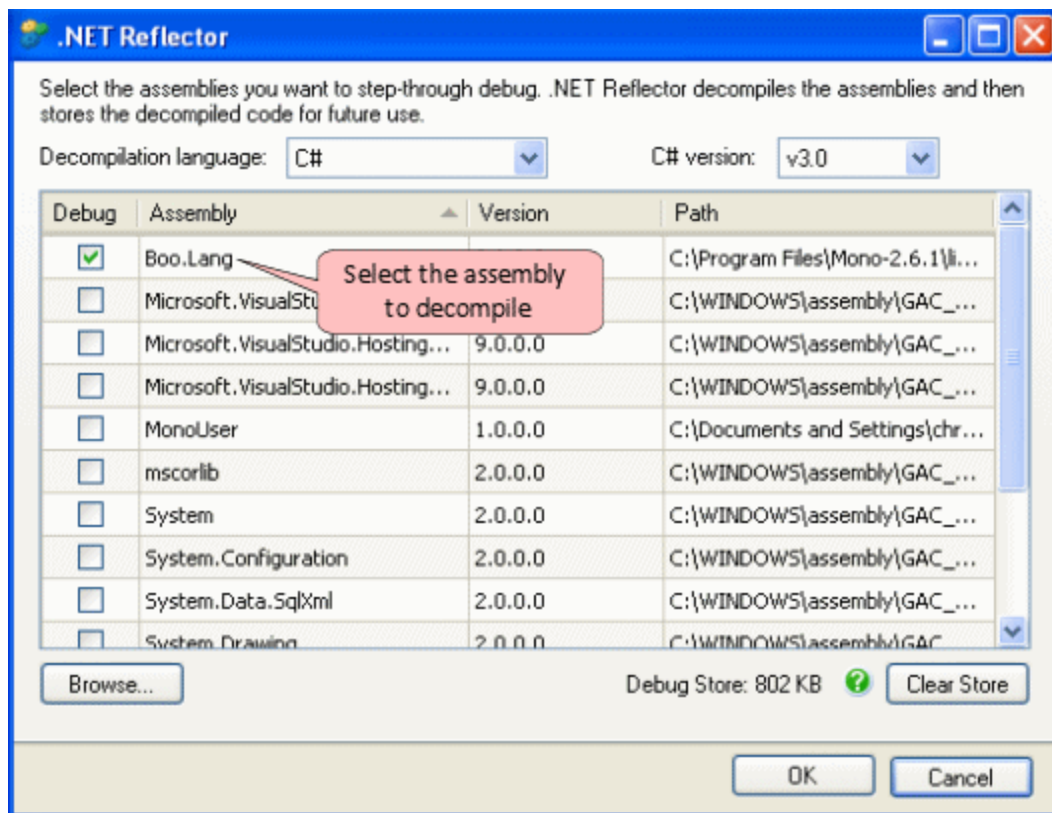
With .NET Reflector Pro, you'll get the latest version of .NET Reflector, plus a Visual Studio add-in containing the Pro features.

.NET Reflector Pro integrates .NET Reflector into Visual Studio to allow you to seamlessly debug into third-party code and assemblies, even if you don't have the source code for them.

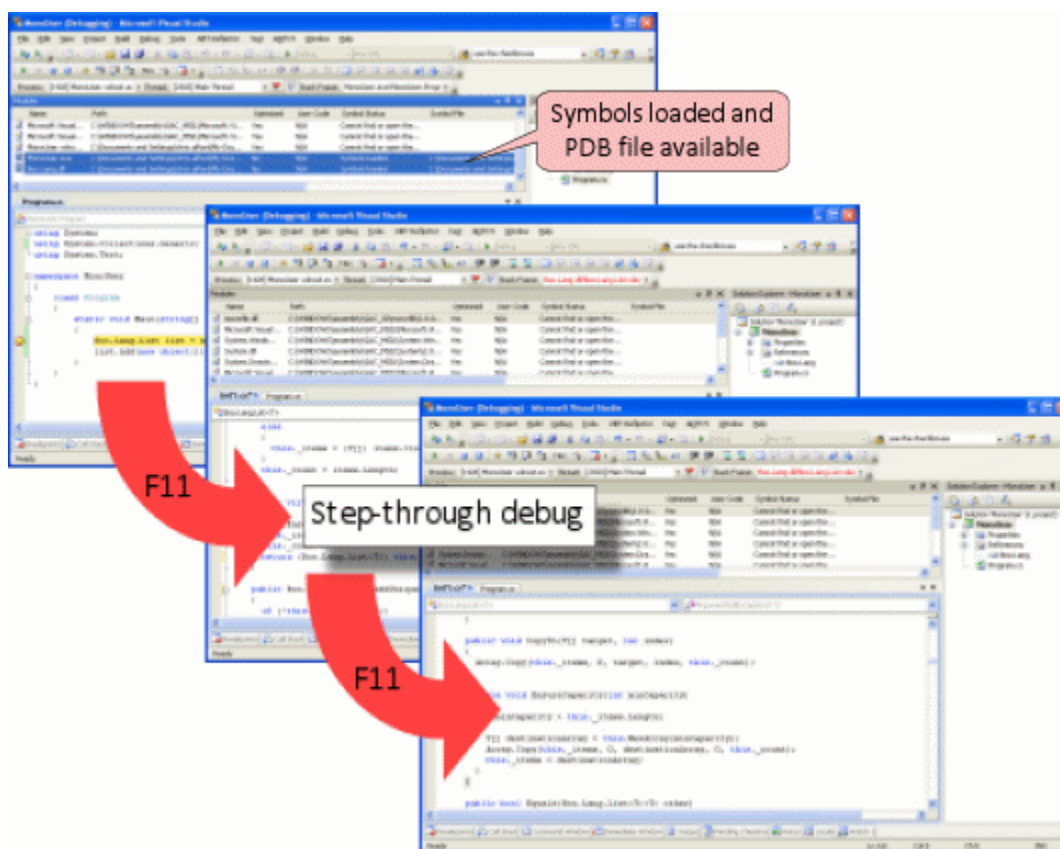
For example, you may have a project that contains a third-party assembly, but you cannot step into it because you do not have the source code or PDB file.



After you have installed .NET Reflector Pro, you can use the Visual Studio add-in to select and decompile the assembly that you are interested in.



You can then return to your source code in Visual Studio and treat the decompiled assembly much like your own code; you can step through the assemblies and use all the debugging techniques that you would use on your own code, including setting breakpoints.



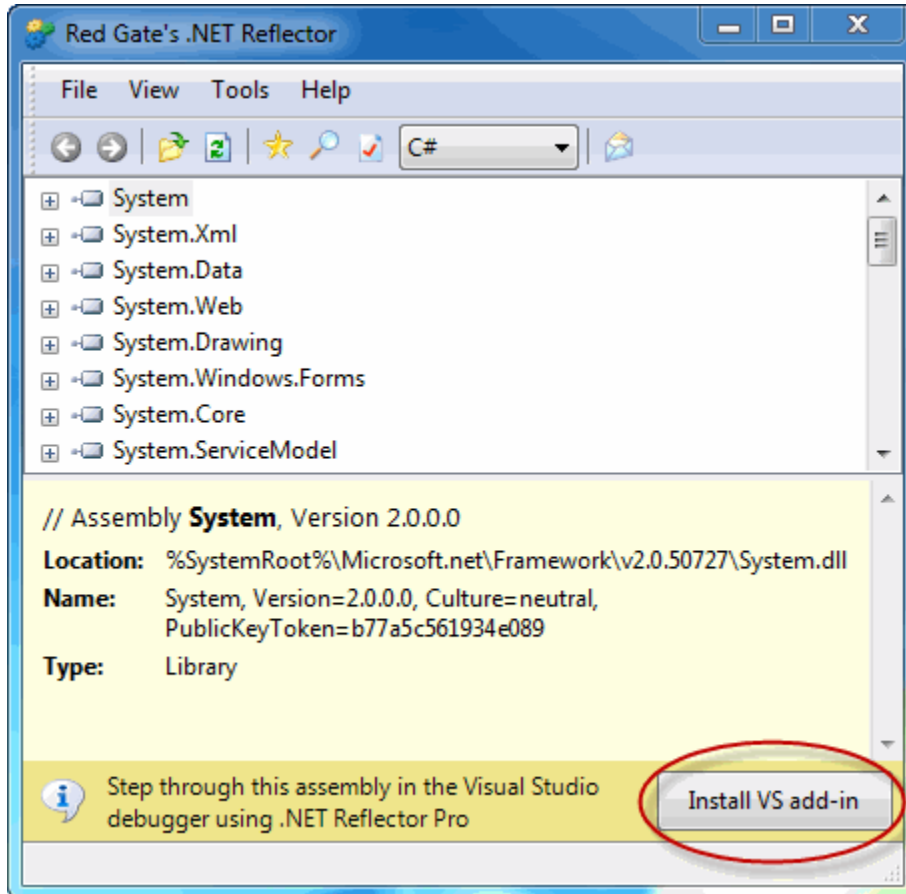
Installing the Visual Studio add-in

The .NET Reflector Pro Visual Studio add-in creates a new menu in Microsoft Visual Studio which allows you to disassemble third-party code from within the Visual Studio user interface.

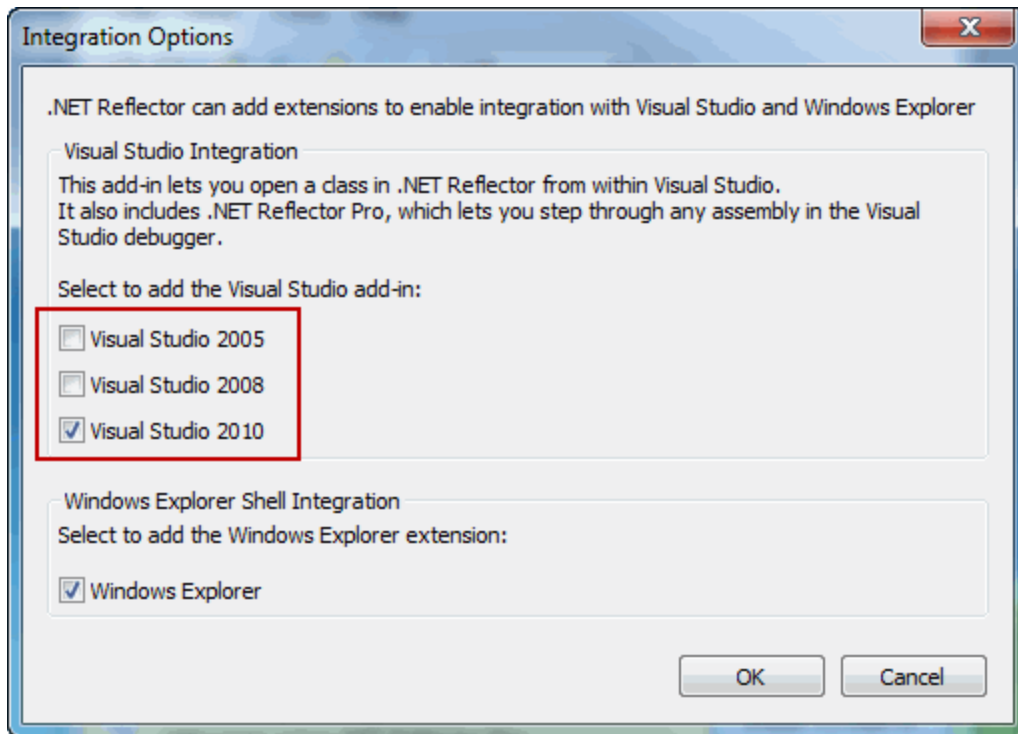
In previous versions of .NET Reflector Pro, the Visual Studio add-in was installed automatically when you first ran .NET Reflector Pro. In version 6.5, in response to user feedback, you must explicitly install the add-in.

To install the Visual Studio add-in the first time you run .NET Reflector

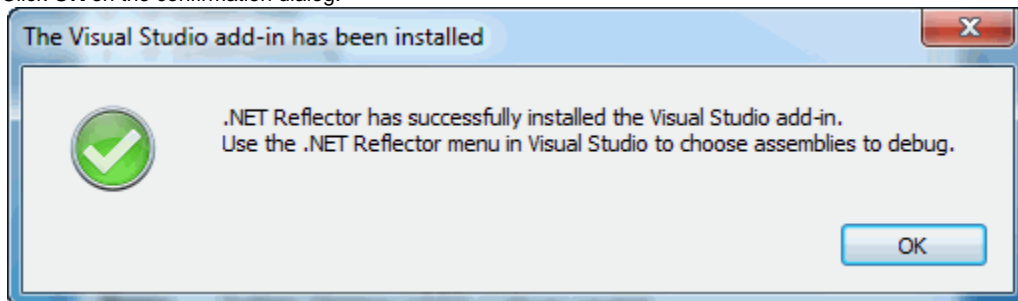
1. Run **Reflector.exe**.
2. Click **Install VS add-in**.



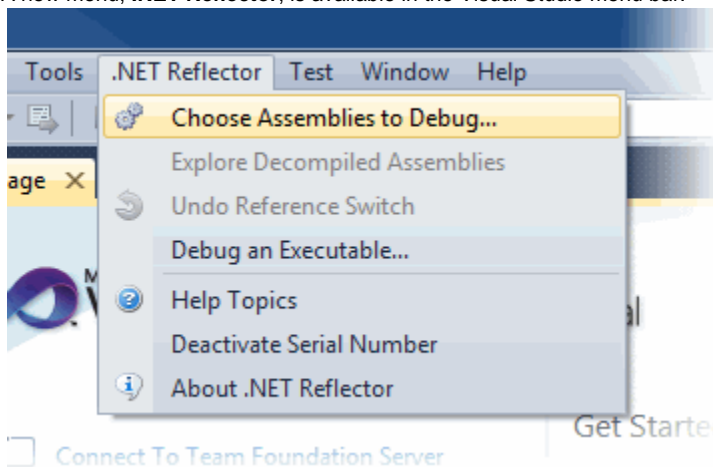
3. In the Integration Options dialog box, under Visual Studio Integration, __select the versions of Visual Studio where you want the add-in and click **OK**.



4. Click **OK** on the confirmation dialog.



5. Restart Visual Studio.
A new menu, **.NET Reflector**, is available in the Visual Studio menu bar.



To uninstall the Visual Studio add-in

1. In .NET Reflector on the **Tools** menu, click **Integration Options**.
2. Under Visual Studio Integration, clear the checkbox for the versions of Visual Studio where you want to remove the add-in.
3. Click **OK**.
4. Restart Visual Studio.

To reinstall the Visual Studio add-in

The Install VS add-in button is only shown if you have never installed the Visual Studio add-in. To install the Visual Studio add-in if you have previously installed it and then uninstalled it:

1. Run **Reflector.exe**.
2. On the **Tools** menu, click **Integration Options**
3. In the Integration Options dialog box, under Visual Studio Integration, select the versions of Visual Studio where you want the add-in and click **OK**.
4. Click **OK** on the confirmation dialog.
5. Restart Visual Studio.

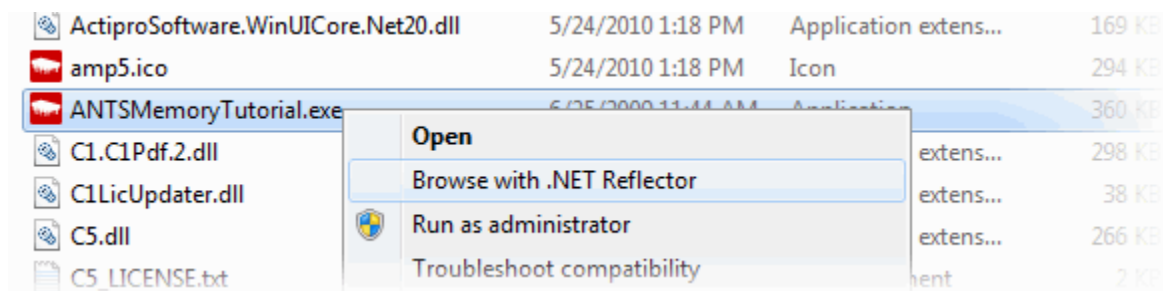
Shell integration

You can also use the Integration Options dialog to integrate .NET Reflector with the Windows shell. This feature:

- allows you to double-click a DLL in Windows Explorer to open it in .NET Reflector Pro
- adds a Browse with .NET Reflector command to the context menu when right-clicking an executable file

To enable shell integration:

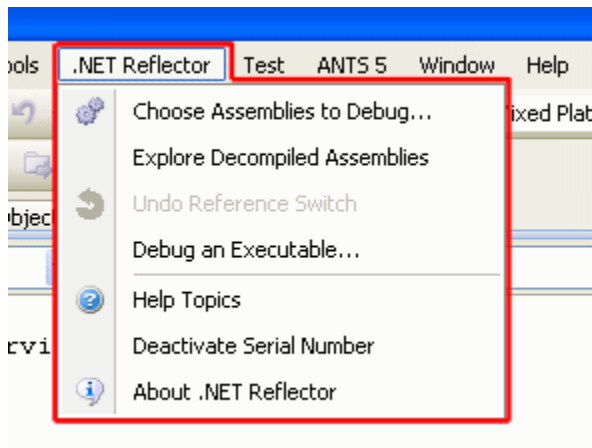
1. Run **Reflector.exe**.
2. On the **Tools** menu, click **Integration Options**
3. In the Integration Options dialog box, under Windows Explorer Shell Integration, __select Windows Explorer and click **OK**.



Browse with .NET Reflector is added for all executables, even if they are not built on the .NET framework. .NET Reflector Pro cannot open executables which do not use the .NET CLR.

Using the Visual Studio add-in

.NET Reflector Pro provides an add-in to Visual Studio. Once [installed](#), this add-in is accessed via the **.NET Reflector** menu option.



From the **.NET Reflector** menu option you can do the following:

- [Choose assemblies to debug](#)
- [Explore decompiled assemblies](#)
- [Undo reference switching](#)
- [Debug an executable](#)
- Access help topics and information on .NET Reflector (Pro)
- [Deactivate a serial number](#)

Getting help

To display the .NET Reflector help from within Visual Studio, on the **.NET Reflector** menu, click



Help Topics.

Working with assemblies

This page describes how to load .NET assemblies into .NET Reflector for decompilation, how to regenerate assemblies, and how to explore the assemblies that have been decompiled. It also explains how to use breakpoints.

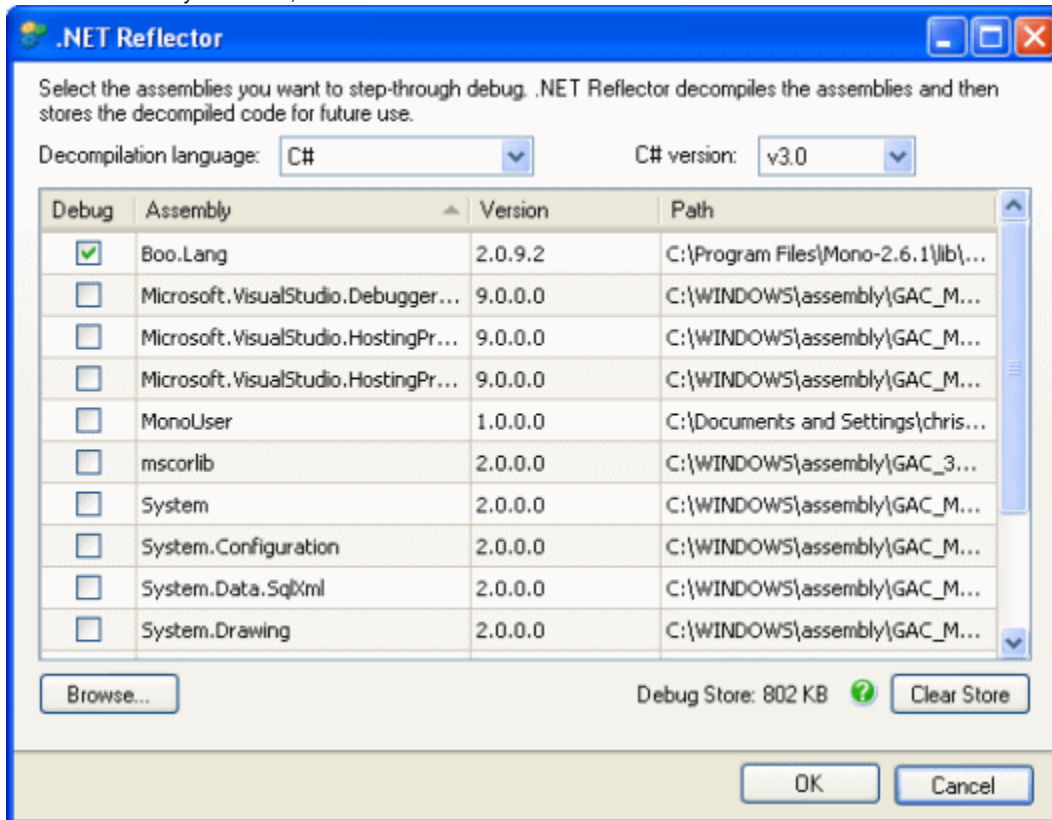
Decompiling assemblies

.NET Reflector decompiles the assemblies and then stores them for future use.

To decompile an assembly:

Open your project in Visual Studio.

1. On the **.NET Reflector** menu, click **Choose Assemblies to Debug**. The **Debug and Decompile** dialog box is displayed. Your project is scanned and all the assemblies that are referenced by your project are displayed alphabetically. This also includes all the assemblies that they reference, and so on.



2. Select the assemblies to decompile and the decompilation language. If an assembly is not shown in the list, click **Browse** and select the assembly.
3. Click **OK** to decompile the assemblies.
This may take a few minutes, depending on the assemblies selected. The decompilation status for each assembly is shown.

Once you have decompiled an assembly, you can debug the decompiled code as if it were your own, and navigate directly into the decompiled code.

Decompilation language

The decompilation language is the language that the code is decompiled into. This is automatically selected to match the language and version of the startup project. Selecting an alternative language may cause your code to be displayed in a different way to what you expect. The language version refers to the coding language and should not be confused with the .NET version.

Why is a PDB file produced?

During decompilation of the assembly, a PDB file is produced. The PDB file is a mapping file that maps back to the source code and contains debugging information.

Regenerating assemblies

Some assemblies require a debugger signature to be added before they can be decompiled. The debugger signature is used by Visual Studio to identify which PDB file is associated with a particular assembly.

If the decompiled assemblies require a debugger signature, then regenerated copies of the assemblies need to be created containing the signature.

To regenerate an assembly:

1. Follow the procedure for [Decompiling assemblies](#).
If the decompiled assemblies require a debugger signature to be added then a dialog box is displayed.
2. Click **Browse** in the dialog box to specify where you are going to save the copies of the assemblies.
You must not save the regenerated copies in the same location as the original assemblies.
3. Click **Regenerate**.
The assemblies are regenerated with the debugger signature added.
After the regeneration has completed, you are asked whether you want your solution to reference the regenerated copies of your assemblies; this is called reference switching, and is only offered immediately after regeneration.
4. Click **Yes**.
The assemblies can now be decompiled as normal.

Occasionally, your solution cannot be pointed to the regenerated copies. An information message is displayed providing further information.

Using the original assemblies

Reference switching is used to point your solution at regenerated copies of assemblies. The presence of debugger signatures allows the assemblies to be decompiled as normal. However, you may want to revert back to the original versions of the assemblies before regeneration. To do this, on the **.NET Reflector** menu, click **Undo Reference Switch**. Once clicked, **Undo Reference Switch** is disabled.

If you have reverted back to an original assembly, you can no longer debug that assembly without decompiling again.

Exploring decompiled assemblies

You can use Visual Studio's Object Browser to explore the decompiled assemblies. You can treat the decompiled assemblies in the Object Browser just like live code.

To navigate around the code, use the **Go To Decompiled Definition** menu option. This takes you into the decompiled source code, enabling you to set break points to help you in debugging the code.

To explore an assembly:

On the **.NET Reflector** menu, click **Explore Decompiled Assemblies**.

You are taken to Visual Studio's Object Browser, where you can navigate into the decompiled assemblies just like any other source code. The Object Browser only shows public methods. To show private methods, open the class and navigate to the definition.

Debugging an executable

If you do not have the source code for a particular application and you are interested in seeing how it works, you can use .NET Reflector.

To debug an executable:

1. Open your project in Visual Studio.
2. On the **.NET Reflector** menu, click **Debug an Executable**.
3. Select the executable you want to debug and open.
4. On the **.NET Reflector** menu, click **Choose Assemblies to Debug**.
The **Debug and Decompile** dialog box is displayed.

You can now follow the steps outlined in [Decompiling assemblies](#).

Setting command line arguments

If you require particular command line arguments to be set when running your executable, you need to create a "dummy" project in Visual Studio. The original executable can then be run with the command line arguments:

1. On the **Project** menu, click **[Project name] Properties**, and then click the **Debug** tab.
2. In **Start external program**, type the executable name.
3. In **Command line arguments**, type the arguments used by your executable.

You can now start debugging in Visual Studio.

Deactivating the serial number

You can deactivate all products associated with the current serial number. This allows you to install the products on a different machine.

To deactivate the serial number:

1. On the **.NET Reflector** menu, click **Deactivate Serial Number**.
The **Deactivate Serial Numbers** dialog box is displayed.
2. Select the serial number and associated products.
3. Click **Deactivate**.

The serial number can now be used for .NET Reflector on a different machine.

Upgrading

Minor releases are free for all users. For example, if you have a license for version 7.0 of a product, you can upgrade to version 7.1 at no cost. When you download and install a minor release, the product is licensed with your existing serial number automatically.

Major releases are free for users with a current Support and Upgrades contract. For example, if you have a license for version 7 of a product, you can upgrade to version 8 at no cost. When you download and install a major release, the product is licensed with your existing serial number automatically.

If you don't have a current Support and Upgrades contract, installing a major release will start a free 14-day trial. You'll need to buy a new license and activate the product with your new serial number.

To check whether you have a current Support and Upgrades contract or see the cost of upgrading to the latest major version of a product:

- visit the [Upgrade Center](#)
- email sales@red-gate.com
- call:
 - 1 866 733 4283 (toll free USA and Canada)
 - 0800 169 7433 (UK freephone)
 - +44 (0)870 160 0037 (rest of world)

To check the latest version of a product, see [Current versions](#).

How to upgrade

You can download the latest version of a product using [Check for Updates](#), the [Upgrade Center](#), or the [Redgate website](#).

- If you download the latest version from the Upgrade Center or our website, you need to run the installer to upgrade the product.

Some Redgate products are available as part of bundle. You can select which products you want to upgrade when you run the installer.

- If you use Check for Updates, the installer runs automatically.

You can install the latest *major* version of any product (other than SQL Backup Pro) on the same machine as the previous version. For example, you can run version 9 and version 10 in parallel. However, installing a *minor* release will upgrade the existing installation.

To revert to an earlier version, uninstall the later version, then download and install the version you want from the Release notes and other versions page. You can use a serial number for a later version to activate an earlier version.

Troubleshooting

How do I remove the Visual Studio add-in?

For details on how to remove the Visual Studio add-in see [Installing the Visual Studio add-in](#).

How do I provide feedback to Redgate?

You can provide feedback by visiting the [.NET Reflector Forum](#). Alternatively, within .NET Reflector, on the toolbar, click



How do I access the help?

To display the .NET Reflector help from the graphical user interface:

- press F1
- or, on the **Help** menu, click



Help Topics

Log files for .NET Reflector Pro

The information on this page applies to .NET Reflector Pro only. .NET Reflector does not create log files.

Log files collect information about the application while you are using it. These files are useful to us if you have encountered a problem.

Enabling logging

In .NET Reflector Pro's default state, very little information is recorded in the log. To record more information:

Using an XML editor, open

%USERPROFILE%\AppData\Local\Red Gate\NET Reflector 6\LoggingConfiguration.xml

Change the value for the recording level to 'WARN' for both the `<root>` and `<logger>` nodes, as in the example below:

```
<root>
  <level value="WARN" />
  <appender-ref ref="DefaultRGFile" />
</root>

<logger name="RedGate.Reflector.Addin">
  <level value="WARN" />
</logger>
```

Opening the log file

Log files are stored in:

%USERPROFILE%\AppData\Local\Red Gate\NET Reflector 6

If the log files in this directory are all empty, you may need to increase the recording level in *LoggingConfiguration.xml* to 'ERROR' or 'FATAL', following the instructions above.

Methods showing no source code in the disassembly window

When disassembling a method using .NET Reflector, the method may appear with no code inside, for instance:

```
[C#]  
    public static void Method()  
    {  
    }  
[/C#]
```

In some .NET assemblies, it is possible to have "stub" methods which contain no IL code. Without any Intermediate Language code in the method, there is nothing to disassemble. The ILDASM tool from Microsoft, which comes with the .NET SDK, can be used to verify that the method contains no IL code.

Permissions error on starting Visual Studio

Upon starting Visual Studio with the .NET Reflector add-in configured, the following or similar error may occur:

Request for the permission of type 'System.Security.Permissions.FileIOPermission, mscorlib, version=4.0.0.0, Culture = neutral, PublicKeyToken b77a5c561934e089' failed.

This may happen when you had already used the Visual Studio integration option in the stand-alone distribution of .NET Reflector version 6.0, and subsequently upgraded to the Professional edition and configured VS integration using the product installer.

In order to fix this problem, it may be necessary to start the stand-alone version of Reflector, and upgrade using **Help->Check for Updates** on the menu.

After the upgrade, start Visual Studio and go to **Tools->Add-in Manager** on the menu, and enable the .Net Reflector addin, ensuring it is also set to run on startup, then close and re-open Visual Studio.

The XP Bug Workaround

When attempting to update .NET Reflector's Visual Studio extension, the installation can fail with a message:

The certificate for a digital signature in this extension is not valid.

This will affect people trying to use check for updates on XP / Windows Server 2003 to upgrade the Visual Studio extension in Visual Studio 2010

This is a Microsoft bug and is detailed [on the Microsoft Knowledge Base](#).

The issue occurs because Windows XP and Windows Server 2003 handle the Certificate Revocation Lists (CRL) differently than other operating systems.

When Extension Manager handles the Certificate Revocation Lists (CRL), Extension Manager uses the same method for all operating systems. However, Windows XP and Windows Server 2003 handle the Certificate Revocation Lists (CRL) differently than other operating systems. Therefore, issue 2 occurs.

The Workaround:

To work around this bug:

1. Go to Visual Studio > Tools > Extension Manager and uninstall the old .NET reflector package.
2. Download the new installer from the Reflector website
3. Run the installer.

Let us know if you still have any trouble with this bug, and we'll investigate further.

Tips and articles

- [.NET Reflector Tips - keyboard shortcuts](#)
- [Debugging a SharePoint customization](#)
- [Debugging into SharePoint and seeing the locals](#)
- [Introduction to building .NET Reflector add-ins](#)

.NET Reflector Tips - keyboard shortcuts

.NET Reflector has a number of keyboard shortcuts. This article details most of them.

Open assembly Ctrl+O

Opens a dialog which allows you to browse to an assembly and open it.

Open assembly list Ctrl+L

This brings up the Assembly list management dialog. You can have multiple assembly lists so that you can easily switch back and forth between different versions of the framework or any other set of assemblies you choose to define:

Export Assembly Source Code Ctrl+S

This is only available at the top level of an assembly. Takes you to the export dialog which allows you to generate a visual studio project for an assembly. While not necessarily fully compilable it does go some way to recovering the source code you may have lost.

We should probably assign this to a key rather than "S" in future, and an issue has been raised.

Open new tab Ctrl+left mouse button click (or mouse 3)

Opens a new tab containing the code for the thing you were clicking on.

Close Current Tab Ctrl+F4

Closes the tab which currently has focus.

Open Bookmarks Pane F2

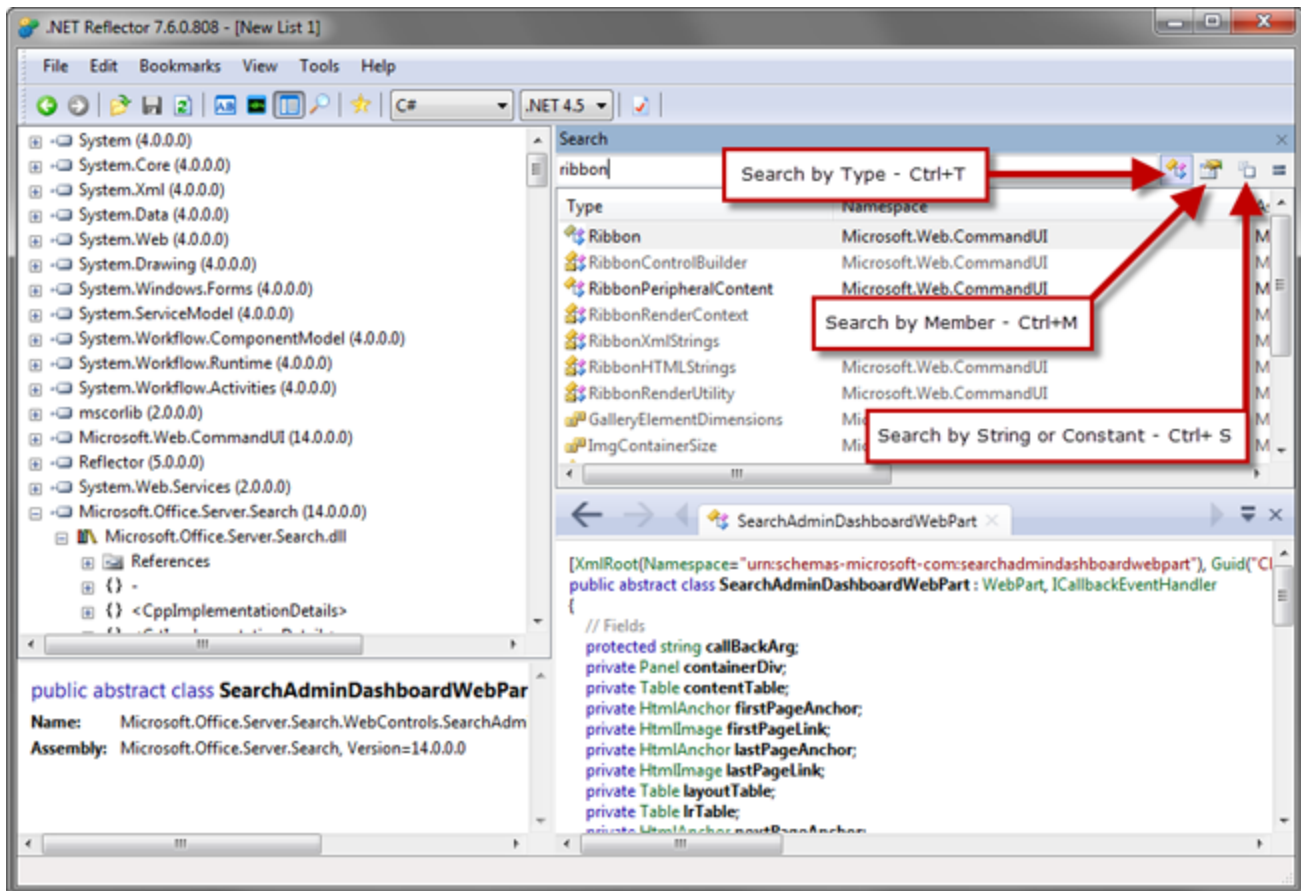
Opens the bookmark pane and displays links to any code that you've previously bookmarked:

Toggle bookmark Ctrl+K

Allows you to set up a bookmark for a particular method, type or assembly.

Open Search F3

Opens the search panel. It's also worth noting that you can change what you search by when this pane has focus:



Decompile Assembly Space or mouse click

This displays code in the currently active tab. The click part is reasonably intuitive, but you might not know about Space.

Open Analyze Pane Ctrl+R

Opens the Analyze pane so that you can do some analytical investigation on whatever was highlighted in the assembly tree:

Close assembly Delete

Closes the currently highlighted assembly in the assembly tree.

Search MSDN Ctrl+M

Available at the namespace level for framework classes Searches MSDN for documentation on this namespace.

Debugging a SharePoint customization

SharePoint has a large and complex code base, and some of the details aren't as transparent or well-documented as developers need. This is a short overview showing how bugs can arise when working with incomplete documentation, and how we can use .NET Reflector to fix those bugs.

In this scenario we're working on a web portal, and we want to add a set of controls to the ribbon. This is simple enough, but we run into an unexpected error. We use .NET Reflector to explore the SharePoint API that is throwing the exception, so we can understand the problem and troubleshoot the control.

Defining custom controls

We're looking at an issue in SharePoint 2010, where XML comments in definition files cause an exception. The issue can arise when creating custom fields, content types, and other custom features that require XML definition files. The XML schema for SharePoint Features is documented, but the documentation doesn't always give rich implementation information.

The custom action behind the control is defined in the XML file, and it's part of a group of controls being used in the ribbon. Because these files quickly become long and complex, we want to comment the code.

This is a simplified "hello world" example of a custom action definition file:

```

<?xml version="1.0" encoding="utf-8"?>
<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
<CustomAction
  Id="Ribbon.WikiPageTab.CustomGroup"
  Location="CommandUI.Ribbon">
  <CommandUIExtension>
  <CommandUIDefinitions>
  <CommandUIDefinition
    Location="Ribbon.WikiPageTab.Groups._children">
    <groups>
    <!-- Group containing the new "hello world" controls -->
    <Group
      Id="Ribbon.WikiPageTab.CustomGroup"
      Sequence="55"
      Description="Custom Group"
      Title="Custom"
      Command="EnableCustomGroup"
      Template="Ribbon.Templates.Flexible2">
      <Controls Id="Ribbon.WikiPageTab.CustomGroup.Controls">
      <Button
        Id="Ribbon.WikiPageTab.CustomGroup.CustomGroupHello"
        Command="CustomGroupHelloWorld"
        LabelText="Hello, World"
        TemplateAlias="o2"
        Sequence="15" />
      <Button
        Id="Ribbon.WikiPageTab.CustomGroup.CustomGroupGoodbye"
        Command="CustomGroupGoodbyeWorld"
        LabelText="Good-bye, World"
        TemplateAlias="o2"
        Sequence="18" />
      </Controls>
      </Group>
      </groups>
      </CommandUIDefinition>
      </CommandUIDefinitions>
      <CommandUIHandlers>
      <CommandUIHandler
        Command="EnableCustomGroup"
        CommandAction="javascript:return true;" />
      <CommandUIHandler
        Command="CustomGroupHelloWorld"
        CommandAction="javascript:alert('Hello, world!');" />
      <CommandUIHandler
        Command="CustomGroupGoodbyeWorld"
        CommandAction="javascript:alert('Good-bye, world!');" />
      </CommandUIHandlers>
      </CommandUIExtension>
    </CustomAction>
  </Elements>

```

(In practice, we could be dealing with hundreds of lines of XML)

When the portal page renders, the controls appear. But when the control is actually used, SharePoint displays an error.

Debugging the error

SharePoint error messages typically have a Correlation ID, a GUID that lets us track the error through the logs. From these logs, we see that the error is a null reference exception.

This is surprising, because in building our control, we've followed the SharePoint documentation pretty thoroughly. At first glance, there appears to be nothing wrong with any of our own code, and the custom action definitions match the documented schema.

To solve the problem, we look into the code with .NET Reflector.

We see that the exception is thrown by Microsoft.Web.CommandUI.Ribbon.CreateRenderContext, so we search for that library using .NET Reflector desktop, and decompile it to view the underlying source code.

The code .NET Reflector shows us is:

```
}
DataNode node4 =
uiproc.GetResultDocument().SelectSingleNode("/spui:CommandUI/spui:Ribbon/spui:Tabs/spui:Tab[@Id='" + this.InitialTabId + "']/spui:Groups");
if (node4 == null)
{
node4 =
uiproc.GetResultDocument().SelectSingleNode("/spui:CommandUI/spui:Ribbon/spui:ContextualTabs/spui:ContextualGroup/spui:Tab[@Id='" + this.InitialTabId + "']/spui:Groups");
}
Hashtable hashtable = new Hashtable();
if (node4 != null)
{
foreach (DataNode node5 in node4.ChildNodes)
{
XmlAttribute attribute2 = node5.Attributes["Id"];
if (attribute2 != null)
{
hashtable[attribute2.Value] = node5;
}
}
}
}
```

If a node is a comment, rather than a conventional element, the value returned is null. This is because XmlNode.Attributes returns as null for all nodes that are not of the XmlNodeType.Element type. This null result is unexpected, so an exception is thrown.

Luckily, the solution is simple, and removing the comments prevents the exception. This may not be an ideal workaround, but because the limitation lies in a third-party assembly, it can't be fixed directly.

Improving on documentation

The XML comments issue is a small example of an undocumented limitation. It's simple to fix with decompilation tools, but would be a substantial debugging task without the ability to look inside the underlying code.

Unfortunately, incomplete (or entirely absent) documentation is quite common for 3rd party tools libraries, and frameworks. SharePoint itself is elaborate, complex, and difficult to debug and understand using only the published documentation. In practice, you can encounter unusual errors, or require customizations that depend on entirely undocumented behavior.

In these cases .NET Reflector saves troubleshooting time, and lets you explore 3rd party libraries in context, to better understand how to integrate with them and build upon them.

Debugging into SharePoint and seeing the locals

The debugging functionality in .NET Reflector is based in Visual Studio. It lets you use the Visual Studio debugger with decompiled code, so you can step through it, set breakpoints, and so on. However, for debugging scenarios like working with SharePoint, the assemblies are hosted and loaded outside Visual Studio. Reflector generates PDB files and allows you to step through the code, but the optimizations made by the CLR mean that you cannot see the values of the local variables.

This limits debugging because you cannot watch these values as they change, and properly follow the data flow. In this article, our technical lead Clive discusses a method for enabling locals for debugging sessions attached to the SharePoint `w3wp` processes.

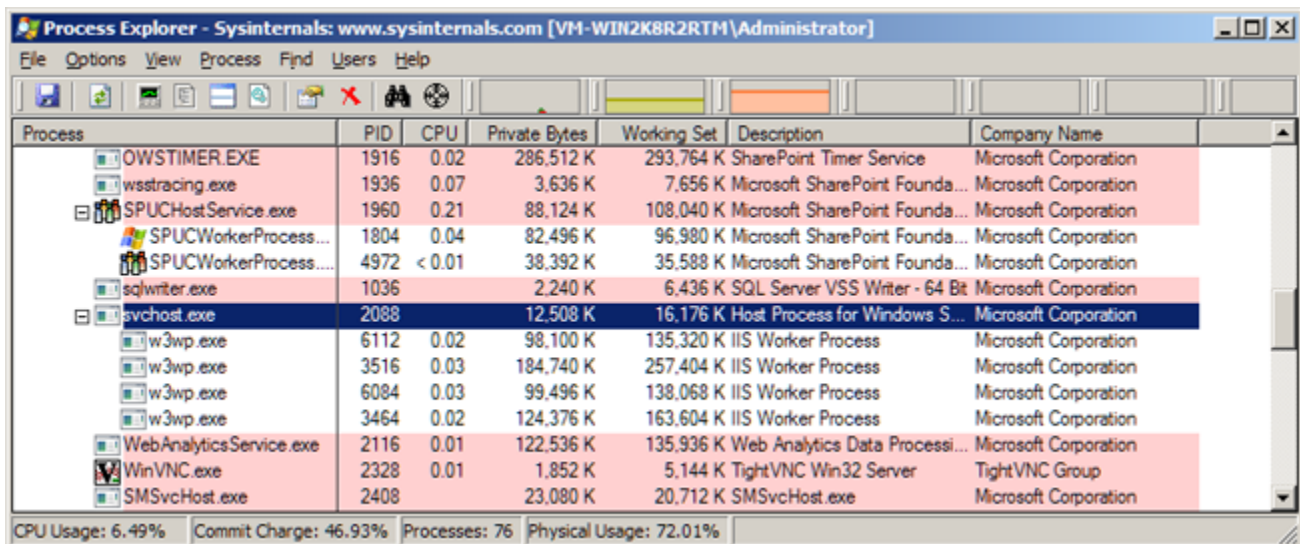
Clive's investigation

The CLR is fairly keen on generating efficient code, and so, if you don't have a debugger attached at the time the code is JIT-ed, it's going to do its very best to generate fast (and therefore undebuggable) code. If you later attach a debugger, the CLR as it currently stands generally won't re-JIT methods, so the debugger is not going to give you a very good debugging experience.

This became clear recently when I did some experimenting on debugging SharePoint using Reflector VSPro. I'm no expert on SharePoint, and it took quite a while to get a virtual machine together. Moreover, running SharePoint inside a VM on my work machine brings the machine to a crawl. But we learned a lot from the debugging experience, and I'm going to walk you through it.

Setting up, and having problems

Initially, I kicked off a web site and watched the `w3wp.exe` instances being created using Process Explorer:

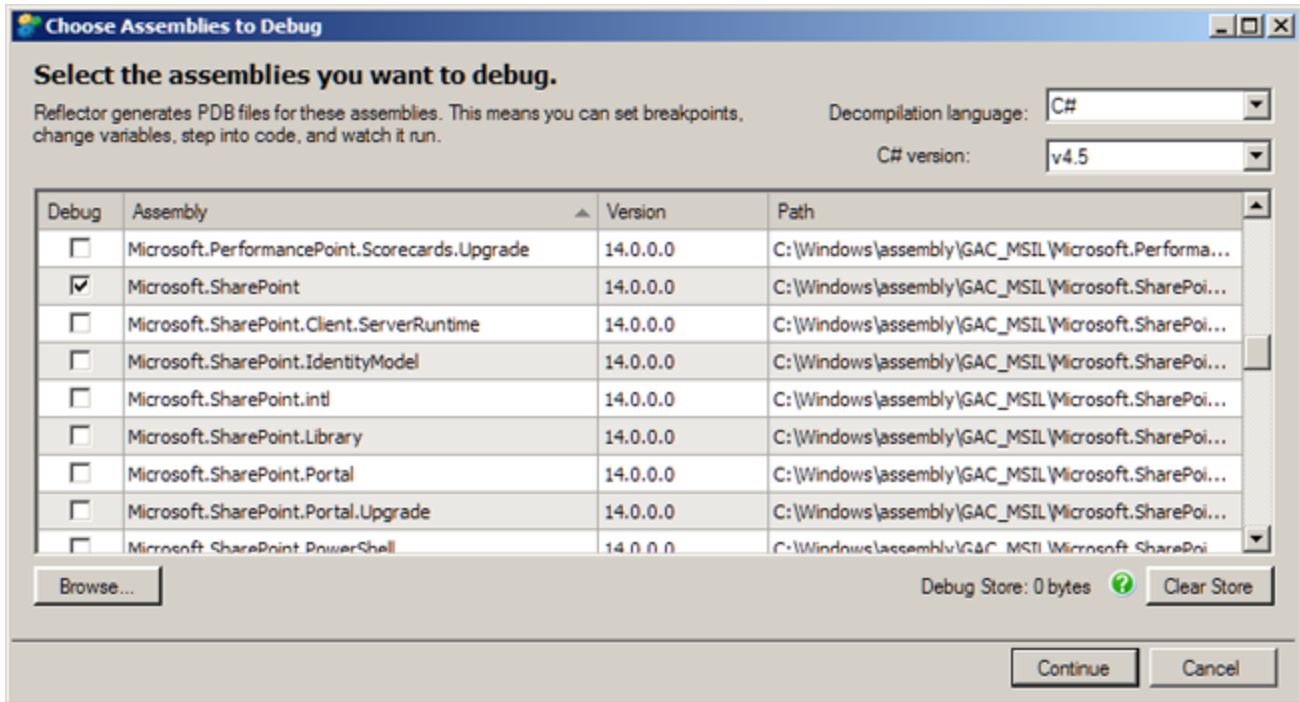


The screenshot shows the Process Explorer window from Sysinternals. The title bar reads 'Process Explorer - Sysinternals: www.sysinternals.com [VM-WIN2K8R2RTM\Administrator]'. The menu bar includes File, Options, View, Process, Find, Users, and Help. The process list table is as follows:

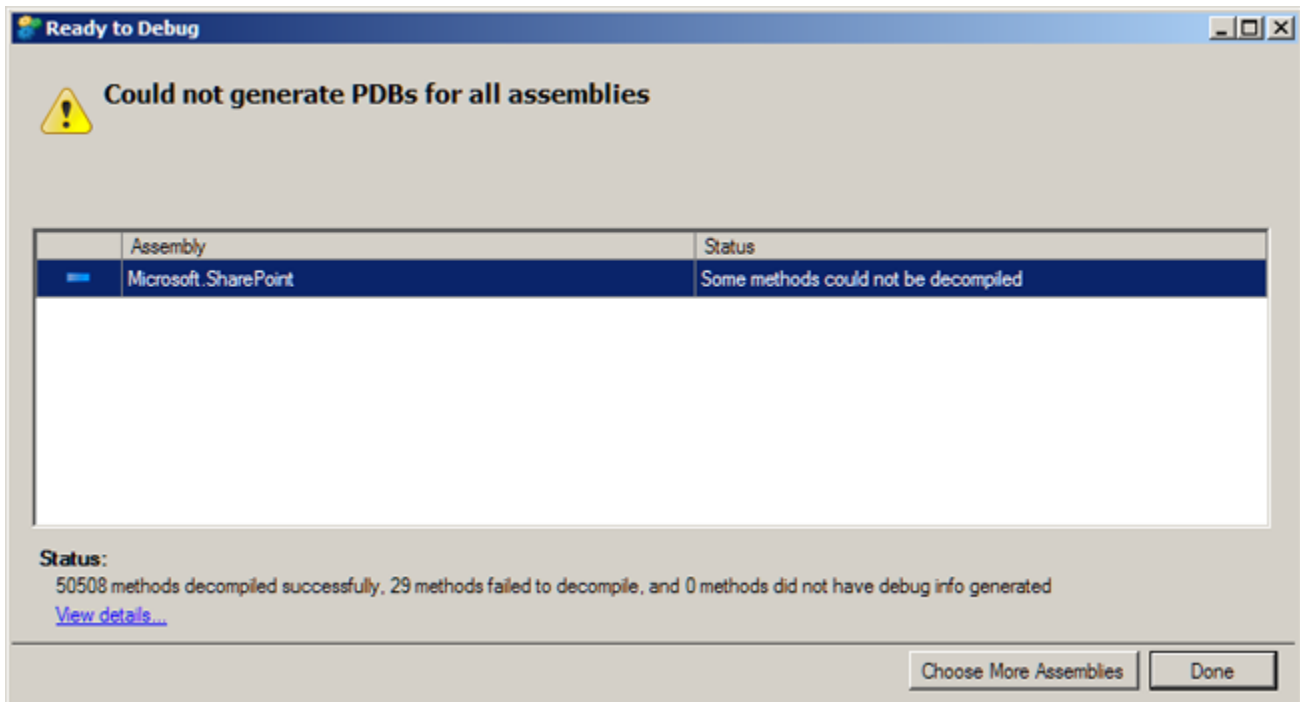
Process	PID	CPU	Private Bytes	Working Set	Description	Company Name
OWSTIMER.EXE	1916	0.02	286,512 K	293,764 K	SharePoint Timer Service	Microsoft Corporation
wsstracing.exe	1936	0.07	3,636 K	7,656 K	Microsoft SharePoint Founda...	Microsoft Corporation
SPUHostService.exe	1960	0.21	88,124 K	108,040 K	Microsoft SharePoint Founda...	Microsoft Corporation
SPUCWorkerProcess...	1804	0.04	82,496 K	96,980 K	Microsoft SharePoint Founda...	Microsoft Corporation
SPUCWorkerProcess...	4972	< 0.01	38,392 K	35,588 K	Microsoft SharePoint Founda...	Microsoft Corporation
sqlwriter.exe	1036		2,240 K	6,436 K	SQL Server VSS Writer - 64 Bit	Microsoft Corporation
svchost.exe	2088		12,508 K	16,176 K	Host Process for Windows S...	Microsoft Corporation
w3wp.exe	6112	0.02	98,100 K	135,320 K	IIS Worker Process	Microsoft Corporation
w3wp.exe	3516	0.03	184,740 K	257,404 K	IIS Worker Process	Microsoft Corporation
w3wp.exe	6084	0.03	99,496 K	138,068 K	IIS Worker Process	Microsoft Corporation
w3wp.exe	3464	0.02	124,376 K	163,604 K	IIS Worker Process	Microsoft Corporation
WebAnalyticsService.exe	2116	0.01	122,536 K	135,936 K	Web Analytics Data Processi...	Microsoft Corporation
WinVNC.exe	2328	0.01	1,852 K	5,144 K	TightVNC Win32 Server	TightVNC Group
SMSSvcHost.exe	2408		23,080 K	20,712 K	SMSSvcHost.exe	Microsoft Corporation

At the bottom, the status bar shows: CPU Usage: 6.49%, Commit Charge: 46.93%, Processes: 76, Physical Usage: 72.01%.

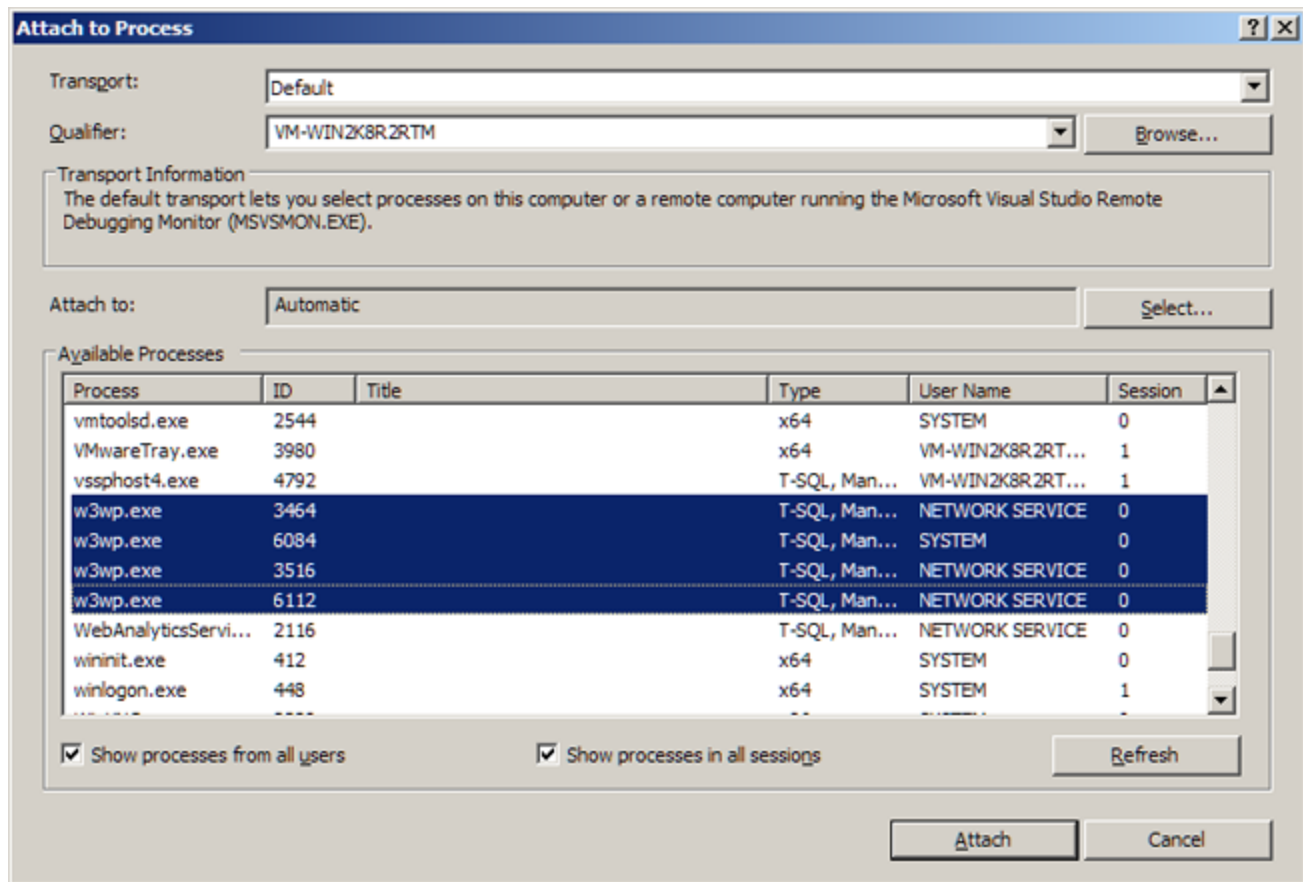
To enable debugging, I then decompiled the SharePoint assembly using Reflector VSPro:



There were a couple of methods that couldn't be decompiled:

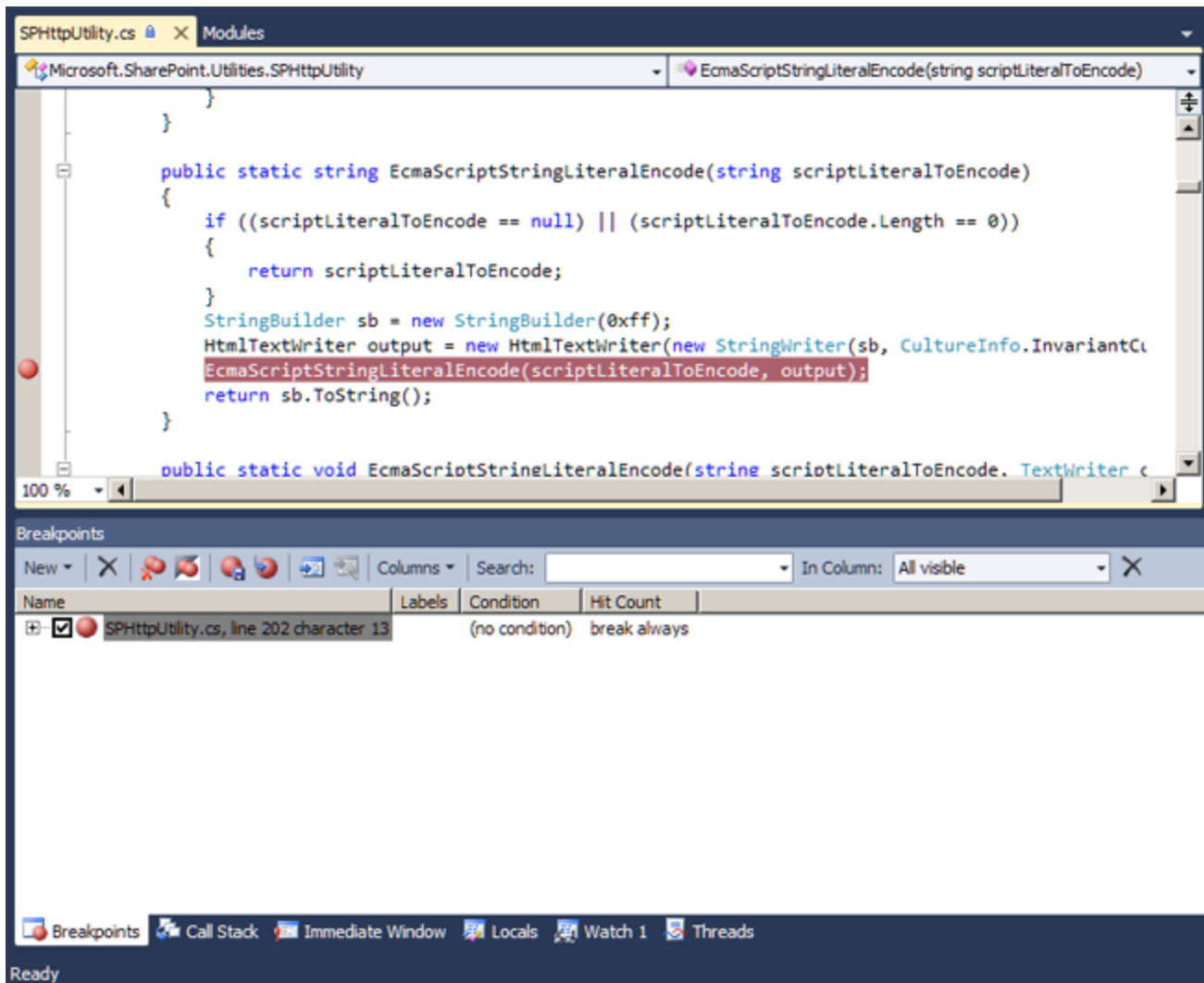


But it's still possible to attach the debugger to the four worker processes, and start to look at what's going on:

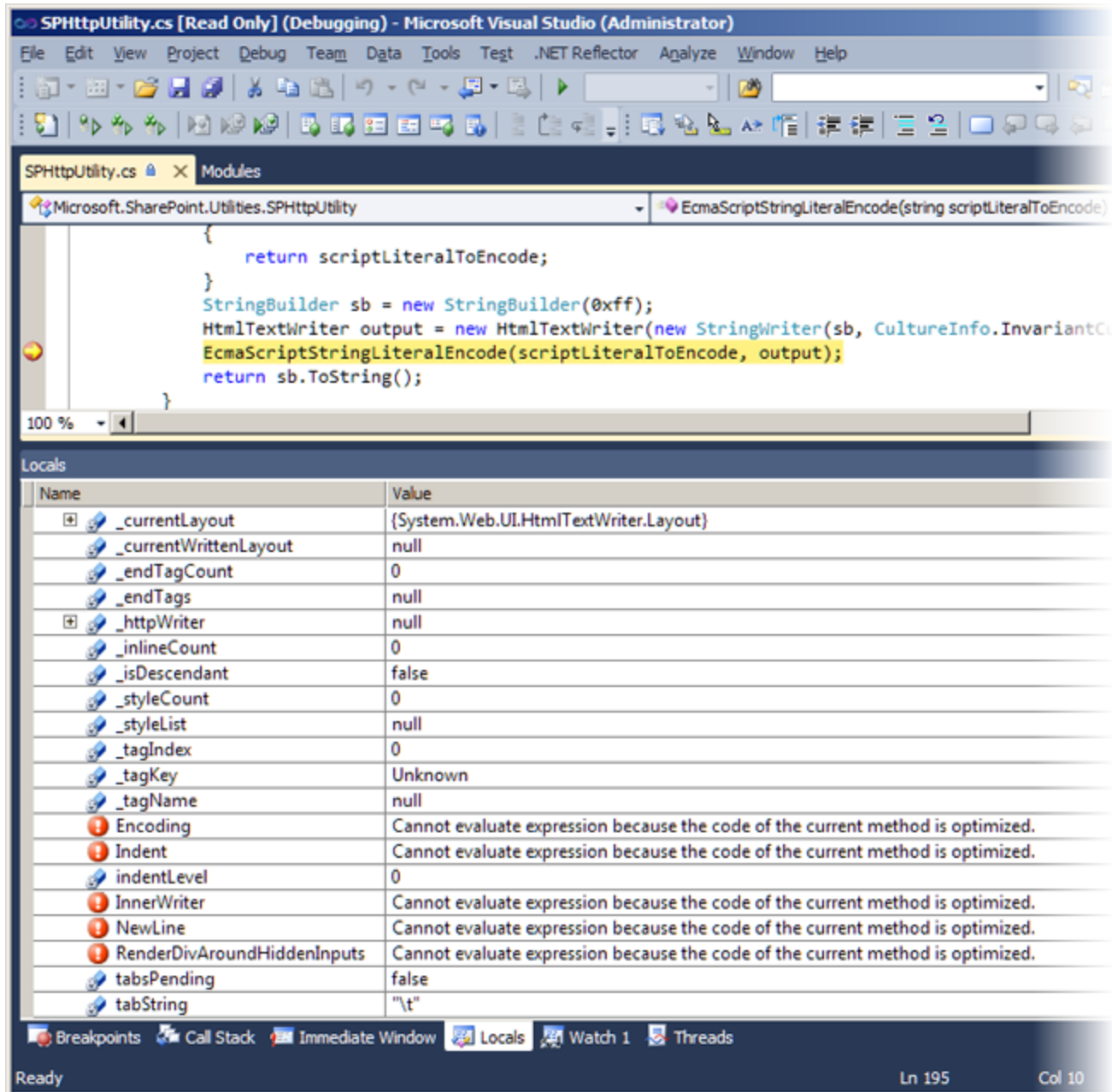


In order to start debugging, I needed to locate a source file corresponding to a class that I knew was going to be called. So I navigated into the Reflector cache directory and found the file `SPHttpUtility`, which I knew would contain the code for the class of the same name.

Finding this in Visual Studio, I set a breakpoint on one of its methods:



I then used a web browser to get the SharePoint to execute the code. Imagine my dissatisfaction when the debugger couldn't display the local variable values:



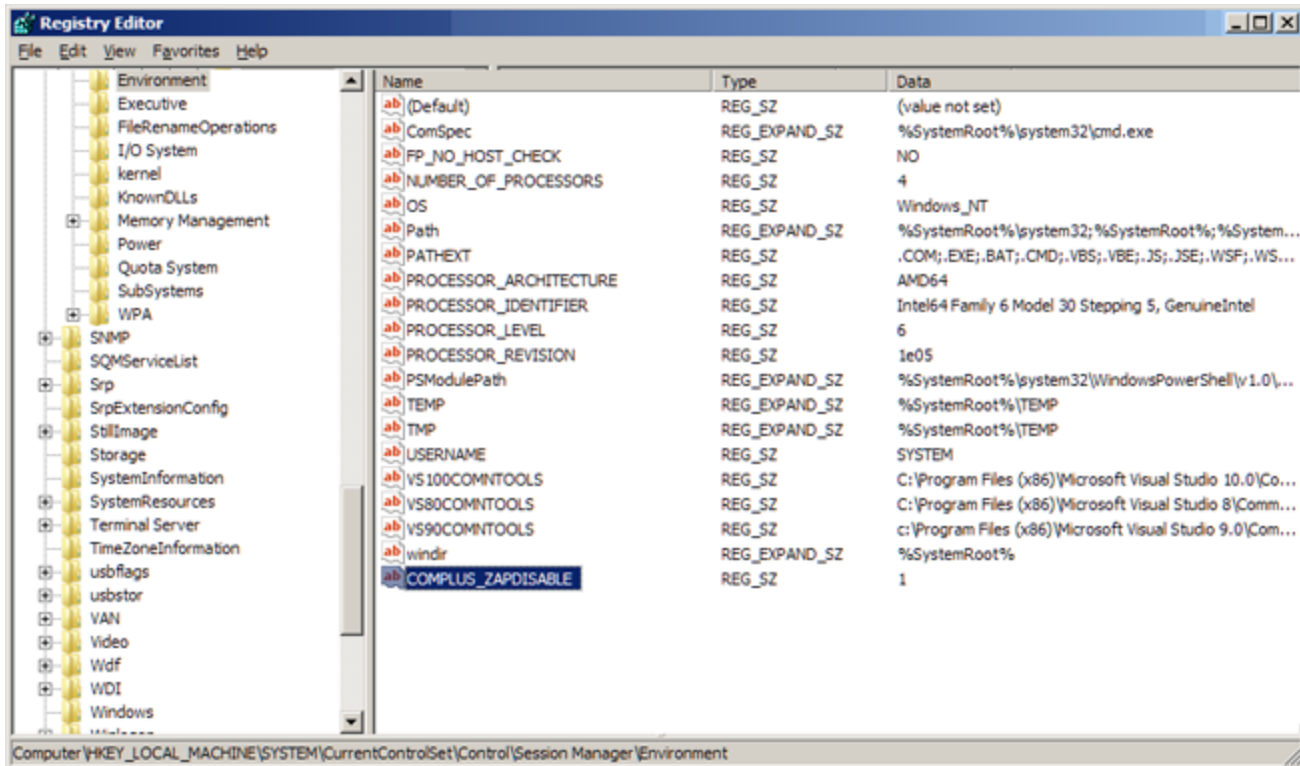
Without the local values, we can't follow the flow of data through the code, and we can't debug as accurately as we would like.

Enabling debugging for SharePoint locals

One problem is that SharePoint is made up of ngen'd assemblies, and you can't see what's going on in that code. The assemblies are loaded automatically by the CLR, and so prevent us debugging.

Disabling optimizations with `COMPLUS_ZAPDISABLE`

To see the locals, we need to prevent the CLR loading the ngen'd assemblies. Fortunately, this can be done by setting the `COMPLUS_ZAPDISABLE` environment variable in the process that loads the CLR itself.



This issue is also documented in the MSDN blog post: [How to disable optimizations when debugging Reference Source](#)

With IIS, I find this easiest to do using the registry entry described in this article on improving the debugging experience.

The environment variable prevents the CLR loading the precompiled version of an assembly. If you set this entry, you'll need to restart the various worker processes, and a useful trick is to use process explorer to check that the environment variable is set in the process, by using the properties tab in the context menu when the process is selected

Preventing optimization with .ini files

The second problem is that the methods were JIT-ed before the debugger was attached, and hence the code is optimized to some extent. The trick now is to use a .ini file, which the JIT will detect and which can be used to override the optimization level specified in the assembly itself.

I went into the GAC, using the Modules window inside Visual Studio to determine where the assembly was actually loaded from. I then made a .ini file, named just as the assembly but with the extension ini instead of dll, and containing the following three lines:

```

Administrator: Visual Studio Command Prompt (2010)

Directory of C:\Windows\assembly\GAC_MSIL\Microsoft.SharePoint\14.0.0.0__71e9bce111e9429c
e111e9429c
12/10/2012  15:39    <DIR>          .
12/10/2012  15:39    <DIR>          ..
25/01/2012  14:49           16,516,968  Microsoft.SharePoint.dll
12/10/2012  11:25              ??  Microsoft.Sharepoint.ini
                2 File(s)      16,517,045 bytes
                2 Dir(s)      7,979,155,456 bytes free

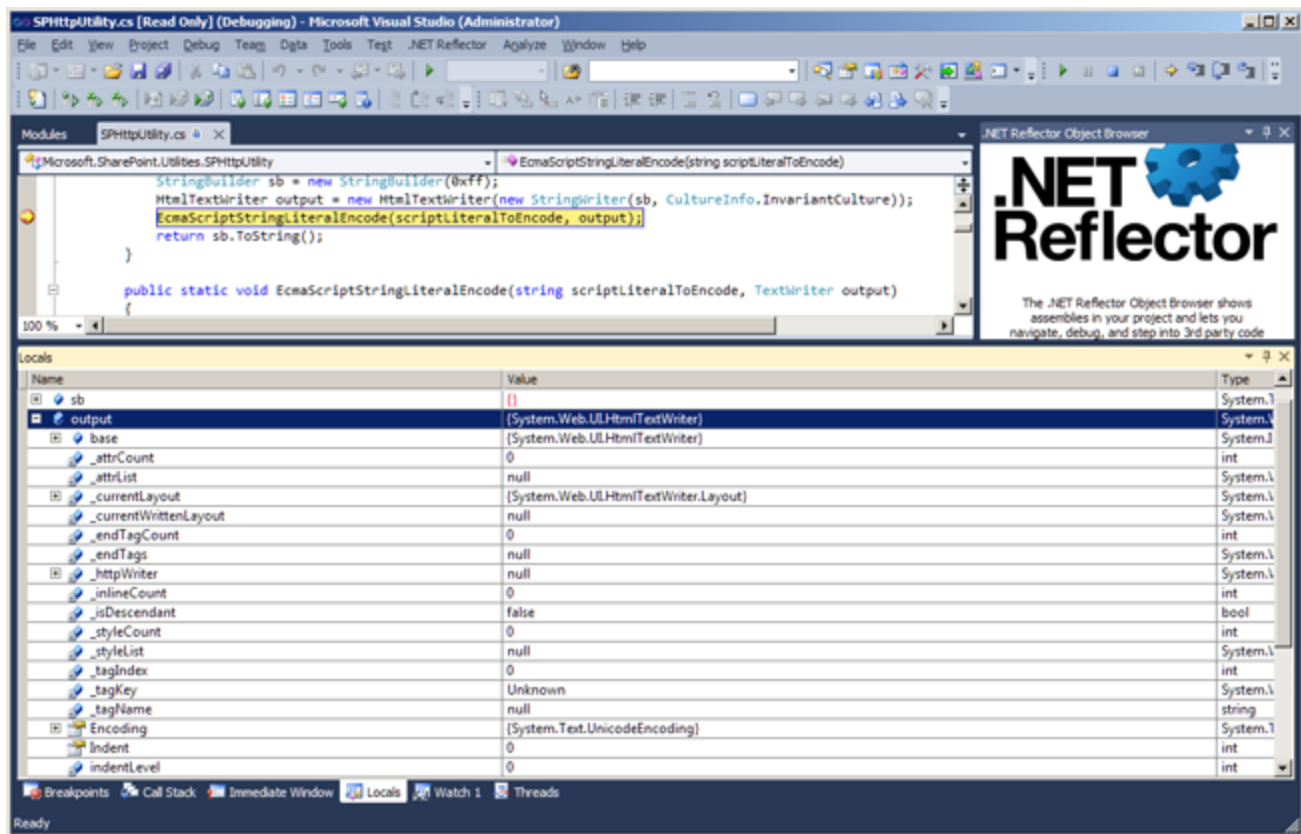
C:\Windows\assembly\GAC_MSIL\Microsoft.SharePoint\14.0.0.0__71e9bce111e9429c>type Microsoft.Sharepoint.ini
[.NET Framework Debugging Control]
GenerateTrackingInfo=1
AllowOptimize=0

C:\Windows\assembly\GAC_MSIL\Microsoft.SharePoint\14.0.0.0__71e9bce111e9429c>

```

In order to try this again, I recycled the `w3wp.exe` processes in my case by using Process Explorer to kill them, although recycling IIS might have been a slightly tidier way to do it. I then hit the web page, let them start up and attached again.

This time, at the same breakpoint, we can see all of the variable values because the code is now unoptimised:



Conclusions

There are two things to take away, if you want to debug into the SharePoint assemblies, with all the locals visible:

1. Prevent the loading of precompiled assemblies
Set the `COMPLUS_ZAPDISABLE` environment variable in the registry
2. Prevent the optimization of loaded assemblies using the JIT
Create .ini files in the same locations as the .dll files you're looking at.

This will give you a much better debugging experience, even if you attach the debugger to the process after it has started.

In my tests there was no impact on system stability, and it's easy to remove the files and re-set the variables when you're done.

Introduction to building .NET Reflector add-ins

.NET Reflector has an extensive add-in framework, and there are plenty of add-ins already available to use as examples of what can be done.

A .NET Reflector add-in is fundamentally a dll/exe assembly file that contains packages. A package is a class that implements the `IPackage` interface, which defines a `Load` and `Unload` method. An `IServiceProvider` interface is passed during loading, and gives access to a set of services which are part of the .NET Reflector object model (the most common of which we'll see below).

Available services

The following table lists the most commonly-used services that can be accessed through the `GetService` method on `IServiceProvider`

Service	Description
IAssemblyBrowser	Maintains the currently selected Code Model object in the <code>ActiveItem</code> property. You can assign a Code Model object like <code>IMethodDeclaration</code> to the <code>ActiveItem</code> to programatically change the currently selected item in the browser window. <code>ActiveItemChanged</code> notifies that the selected item has changed.
IWindowManager	Manages the application window and pane windows. You can add your own pane windows to the <code>Windows</code> collection which will create an <code>IWindow</code> hosting frame. <code>ShowMessage</code> can be used to show notification messages to the user.
ICommandBarManager	Manages the Reflector menu bar, tool bar and context menus. You can lookup a context menu by its identifier and add items to it.
IConfigurationManager	Manages the sections from the Reflector configuration file as a set of <code>IConfiguration</code> objects. Lists of items are represented as properties named "0?", "1?", "2?", and so on.
IAssemblyManager	Maintains the list of currently loaded assemblies. <code>LoadFile</code> can be used to load an assembly file from disk. <code>Unload</code> allows you to unload an assembly from memory. The <code>Assemblies</code> collection holds all the currently loaded assemblies.
ILanguageManager	Manages formatting modules for different programming languages. The <code>ActiveLanguage</code> property exposes the <code>ILanguage</code> object currently used for rendering. You can add your own language rendering code by implementing the <code>ILanguage</code> interface. Use <code>RegisterLanguage</code> to add your add-in to <code>ILanguageManager</code> .

Although the .NET Reflector API exposes more interfaces than this, these are the most commonly used ones.

Building a HelloWorld add-in

A simple "HelloWorld" add-in can be created by implementing the `IPackage` interface.

The `Load` method is implemented to ask the `IServiceProvider` for the `IWindowManager` service, which allows you to communicate with .NET Reflector's windowing system. Finally, the `ShowMessage` method is used to show a message to the user:

```
using System; using Reflector; internal class HelloWorldPackage : IPackage
{ private IWindowManager windowManager; public void Load(IServiceProvider
serviceProvider)
{ this.windowManager = (IWindowManager)
serviceProvider.GetService(typeof(IWindowManager));
this.windowManager.ShowMessage("Loading HelloWorld!");
} public void Unload()
{ this.windowManager.ShowMessage("Unloading HelloWorld!");
}
}
```

The code can be compiled into an add-in dll, which is referencing `Reflector.exe` as a library:

```
csc.exe /target:library /out:HelloWorld.dll *.cs /r:Reflector.exe
```

The add-in can then be copied to your Reflector directory and loaded using the `View Add-Ins` menu. While this is a very basic add-in, the fundamentals of the construction and implementation don't change.

Adding items to command bars and context menus

The `ICommandBarManager` service allows you to add menu items to the .NET Reflector main menu and context menus. Each sub-menu and context menu is registered in the `CommandBars` collection with an identifier name, and the following table lists the most commonly used identifiers:

Identifier	Description
Tools	The tools menu shown as part of the main menu.
Browser.Assembly	The context menu for the currently selected assembly.
Browser.Namespace	The context menu for the currently selected namespace.
Browser.TypeDeclaration	The context menu for the currently selected type declaration.
Browser.MethodDeclaration	The context menu for the currently selected method declaration.

Learning more

Thoroughly documenting the .NET Reflector API is something we hope to improve in future.

We currently recommend this series of articles by Jason Haley:

- [Getting Started with .NET Reflector add-ins](#)
- [Create your own add-in : The basics](#)
- [Create your own add-in : More details](#)
- [Wrapping .NET Reflector](#)

For more examples, see:

- [.NET Reflector Add-in Tutorial \(Peli de Halleux\)](#)
- [Building the .NET Reflector Add-in \(Jamie Cansdale\)](#)

Release notes and other versions

Version 9.0 (current)	November 16th, 2015 (latest)	Release notes	Documentation
Version 8.5	March 4th, 2015	Release notes	Documentation
Version 8.4	October 27th, 2014	Release notes	
Version 8.3	January 20th, 2013	Release notes	
Version 8.2	June 16th, 2013	Release notes	
Version 8.1	May 7th, 2013	Release notes	
Version 8.0	February 20th, 2013	Release notes	
Version 7.7	October 12th, 2012	Release notes	<i>Documentation not available</i>
Version 7.6	July 16th, 2012	Release notes	
Version 7.5	February 13th, 2012	Release notes	
Version 7.4	November 12th, 2011	Release notes	
Version 7.3	July 15th, 2011	Release notes	
Version 7.2	July 7th, 2011	Release notes	<i>Documentation not available</i>
Version 7.1	May 4th, 2011	Release notes	
Version 7.0	March 7th, 2011	Release notes	
Version 6.5 - Pro	July 14th, 2010	Release notes	Documentation
Version 6.5	July 14th, 2010	Release notes	
Version 6.1	February 24th, 2010	Release notes	
Version 6.0 - Pro	February 9th, 2010	Release notes	
Version 6.0	February 9th, 2010	Release notes	

.NET Reflector 7.7 release notes

- Collapse All Assemblies powercommand has now moved to file menu
- Collapse All Assemblies powercommand now has updated icon
- Collapse Assembly option added to assembly root node context menu
- Copy As powercommand moved to context menu
- Import/Export Assembly list powercommand moved to file menu
- Open file location moved to to assembly root node context menu
- Open with Powercommand added to context menu
- Open Zip by dragging zip into Reflector
- Referenced by Powercommand added to analyzer
- Removed older non-functional powercommands
- Archived less commonly used powercommands into powercommands addin
- Re-ordered Reflector context menu
- Added new icons for bookmark/remove bookmark
- Refresh icons in Visual Studio Extension
- Reflector intercept of f12 functionality in visual studio is now toggleable.

.NET Reflector 7.6 release notes

- Installer now "per-user" again, making management via VS smoother
- Coupled with work in Build 7, gives Reflector a true installer
Installation process now attempts to elevate by default
 - Known issue: installing alongside Reflector 7.6.0.604 causes duplicate menus in Visual Studio
- Updating Core tool splash screen
- Adding an installer
- Better support for SharePoint DLLs
- Further improvements and bug fixes for Async support
- Adding a dynamic welcome screen
- Extending our C#5 async functionality support
- Continued re-plumbing Reflector UI to better support Dev11 theming
ROB & context menus now accept theming correctly
 - Known bug: some text in the ROB is rendered too light
- Continued improvements of C#5 async support.
Support for most new async methods in System.IO
 - Known issue: nested Try blocks aren't currently handled very well
 - If we're not confident about async decompilation, Reflector emits code for .NET 4.0
- Updated "about" dialog in standalone tool
- Control over whether Delphi & Oxygene are available decompilation options
- WinMD files are now discoverable on both 32-bit and 64-bit machines.
- Continued work on theming support for Dev 11
- Further development on C#5 async functionality support
- Added early theming support for Dev 11
- Added early support C#5 async functionality
- Added functional support for Dev 11
- Added support for .NET Framework 4.5 and its assemblies
- Added support for WinRT winmd libraries.
- Removed VS Addin support for VS 2005 / 2008

.NET Reflector 7.5 release notes

V7.5.2.1

- Reflector automatically detects the winmd directory, and can load the contained files into the assembly list

V7.5.2

- The ROB now remembers its state each time you close & re-open Visual Studio
- Check For Updates will automatically update the Visual Studio Integration from older add-ins and packages

V7.5.1.13

- Fixed several com exceptions, mostly relating to race conditions
- Updated the trial dialog string

V7.5

- Many string changes to make things easier to understand
- Assorted design changes to dialogs and screens
- Improvements to the installation experience of the package
- Improved the speed of type by type decompilation
- Improved handling of the case where the add-in is superseded by the package (we remove the old menu items)
- Error reporting experience is more fine-tuned. Users have options to be notified of work-arounds and fixes
- Hide menu items such as "enable debugging", instead of just disabling them
- Make "enable debugging" work on all items of the tree, rather than just the top level assembly item
- "Go to decompiled definition" more commonly enabled.
- Setting a break point causes the necessary PDB file to be automatically generated
- Decompiled code can now be stepped-into like any other code
- Attempting to step into inaccessible code triggers instructions on how to debug it correctly
- Multiple PDBs can now be generated in parallel
 - PDB generation can currently take a little time, especially if you're generating several or the assemblies are particularly large
- The trial dialog screen has been updated to be more informative and clear
- Re-introduced "Go to Decompiled Definition" right-click menu item
- Improved support for 'Go to Definition'(F12) in code without source
- A number of simple usability enhancements
- A number of licensing and installation bug fixes
- Added support for 'Go to Definition'(F12) in code without source
- Improved path to decompiled code
- Improved path to pdb generation for code you want to debug
- Completed move to a VS Package
- Started transition from a VS Add-in to a VS Package
- Added support for VS11
- Turned on SmartAssembly feature usage reporting on by default for all EA builds
- Added loaded project references to the Reflector Object Browser (ROB)
- Double click on any type in the 'ROB' to decompile in a new VS editor pane
- Added a Reflector 'Go to Definition' context menu item to navigate through code without source
- Added a new 'Reflector Object Browser'(ROB) into Visual Studio (will eventually offer decompilation by type inside VS)
- Changed the way Reflector shows the version number to display the correct build number
- Bug fixes:
 - "Open in Reflector" context menu not working.
 - XAML Source Code is wrong delimited.
 - Missing end tag in XAML Translation.
 - Code generation for different versions of the same assembly
 - 'Flatten Namespaces' dialog
 - Various decompilation and assembly-loading problems have been fixed drop us a note if you'd like to know more.

Note: This is not an exhaustive list. This has been compiled from blog and forum posts, but will much more representative from V7.5 onwards.

.NET Reflector 7.4 release notes

- Improve translation of hitherto less common decompilation cases (e.g. Lambda expressions).
- Improved handling of exceptions.
- Better handling of decompilation cases involving obfuscation.
- A sprinkling of UX improvements that you won't overtly notice, but which will make life more pain-free.

.NET Reflector 7.3 release notes

- BAML to XAML decompilation in the core product
- Fixed a bug that cause Reflector to hang (on some systems) when opened from the Windows Explorer shell integration.

.NET Reflector 7.2 release notes

- Fixed the start-up performance problem that was Plaguing V7.2.
- Many decompilation improvements, particularly when dealing with expressions and LINQ queries.
- Backwards compatibility for add-ins build for previous versions of .NET Reflector.
- Many additional bug fixes to cure stability problems, along with a number of UI glitches.

.NET Reflector 7.1 release notes

- Improved VB support
- Better handling of resource types
- Lots of improves to C# decompilation
- addition of command line activation /deactivation
- Fixes for 105 other bugs, including a number of stability improvements:
 - VB ByRef argument handled incorrectly.
 - Incorrect do-while-continue generated in C#.
 - Export Assembly Source Code creates an invalid VB project file.
 - "A break point could not be inserted at this location" error.

.NET Reflector 7.0 release notes

Features and Enhancements

.NET Reflector is a class browser, analyzer, and decompiler for .NET assemblies.

Highlights in this new release include:

- Better decompilation, including the addition of iterator block (yield) support, as well as improvements to the .NET 4.0 decompilation,
- Tabbed decompilation,
- Ability to decompile and explore source code for referenced assemblies in Visual Studio,
- Support for Silverlight XAP files
- PowerCommands integration

How to Install and Use the Visual Studio Add-in

To install the Visual Studio add-in:

1. Unzip everything into a suitable location.
2. Run **Reflector.exe**. Then click **Tools > Visual Studio and Windows Explorer Integration** on the main menu, and select the versions of Visual Studio in which you want to install the add-in.

You can then open Visual Studio, and use the **.NET Reflector** menu to select referenced assemblies you would like to debug into. These assemblies are then decompiled to C# or VB. You can then step into them whilst debugging, set breakpoints, see the values of variables and parameters, etc. In fact, the only debugger functionality that will not work is edit and continue.

You can also decompile and explore source if you right-click on a project reference, and then click **Decompile and Explore** on the context menu.

To remove the Visual Studio add-in:

1. On .NET Reflector's **Tools** menu, click **Visual Studio and Windows Explorer** and clear the appropriate **Visual Studio Integration** checkbox. The next time you open Visual Studio the add-in will be removed.

Supported .NET Framework Versions

.NET Reflector requires Microsoft .NET Framework 3.5 or later to run, but can open assemblies compiled against any version of the .NET framework, or Mono.

Supported OS Versions

- @Windows XP SP2 or later, @Windows Server 2003, @Windows Vista, @Windows Server 2008 & 2008 R2, @Windows 7

Supports 32-bit and 64-bit versions of all listed operating systems, where applicable.

Supported Visual Studio Versions

The following versions of Microsoft Visual Studio are supported by the .NET Reflector add-in:

- Visual Studio 2005, 2008, and 2010

Improvements

- Introduction of a tabbed browsing model.
- UI improvements to make features more discoverable.
- Inclusion of Jason Haley's PowerCommands add-in.
- Improves to decompilation, such as handling iterator blocks.

.NET Reflector 6.5 release notes

This release fixes a number of bugs in version 6.1. It also adds support for some of the new language features in .NET 4:

- The dynamic type in C# 4
- Co- and Contra- variance markers in interfaces and delegates
- The new expansion of the lock() { } statement
- Optional parameters

.NET Reflector Pro 6.5 release notes

In response to customer feedback, this release changes the installation method for the .NET Reflector Visual Studio add-in so that the add-in is no longer installed automatically. This version also includes the bug fixes and support for new features in .NET Reflector version 6.5.

.NET Reflector and .NET Reflector Pro 6.1 release notes

These releases fix several problems that were present in the 6.0 release:

- Support for using a copy of Reflector.cfg stored alongside Reflector.exe has been re-enabled so users upgrading from 5.x releases will not lose their settings.
- Fixed unhandled exception on exit of Visual Studio when .NET Reflector add-in used in conjunction with TestDriven.NET add-in.
- Added better support for dealing with framework assemblies, which only contain meta-data, in the "Referenced Assemblies" folder.
- Fixed problem where attempted decompilation with CppCliLanguage add-in would lead to display of a page on the Redgate website.
- Added option to activate .NET Reflector Pro to .NET Reflector menu in Visual Studio after receiving feedback from a number of users that it was hard to figure out how to activate the product.

.NET Reflector 6.0 release notes

.NET Reflector is a class browser, analyzer, and decompiler for .NET assemblies.

.NET Reflector 6 includes a new Visual Studio add-in that allows developers to jump into Reflector from your source code.

This version of .NET Reflector supports:

- .NET 4.0 assemblies
- Opening assemblies from the Global Assembly Cache
- Improved graphics and usability

.NET Reflector Pro 6.0 release notes

The new .NET Reflector Pro is an add-in to Visual Studio that lets you debug third-party code and assemblies. It decompiles the assembly to either C# or VB code, and lets you step through it in Visual Studio 2008, just as you would do with your own code. This is all controlled from a Visual Studio add-in, so you don't need to leave your coding environment.

- Full support for .NET 1.0, 1.1, 2.0, 3.0, 3.5 and 4.0
- Decompile an entire assembly to either C# or VB to view and debug in Visual Studio
- Step-through debugging of any assembly in Visual Studio (as long as it's not obfuscated):
 - Step into and set breakpoints anywhere in any assembly
 - Watch variables in the decompiled code
 - Use Visual Studio's advanced debugging features in decompiled code: Set Next Statement, modify variable values, and dynamic expression evaluation in the immediate window