FxCop and Code Analysis: Writing Your Own Custom Rules

Jason Kresowaty <jason@binarycoder.net>

Version 1.5 (8/20/2008)

Table of Contents

Introduction

Microsoft is taking the .NET development world by storm with its FxCop tool. The tool is designed to check .NET code for violations of a wide range of programming rules and conventions. The rules included with FxCop draw heavily upon Microsoft's Framework Design Guidelines as described in *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* by Krzysztof Cwalina and Brad Abrams (ISBN 0321246756).

The kinds of rules checked by FxCop include:

- Design
- Globalization
- Interoperability

- Mobility
- Naming
- Performance
- Portability
- Security
- Usage

Unlike traditional code analysis tools (such as lint for C), FxCop does not analyze source code. Instead, it analyzes the binary Common Intermediate Language (CIL) generated by .NET compilers and persisted in .NET assemblies (EXE and DLL files). Analysis is enabled by the rich metadata that is a part of CIL. By analyzing assemblies directly, FxCop avoids being tied to any particular .NET programming language: it will work without modification against C#, VB.NET, and potentially other .NET languages.

FxCop is a mature tool. The first version of FxCop was released to the public shortly after the release of the .NET Framework 1.0 itself in 2002. In Summer 2007, FxCop was awarded the Chairman's Award for Engineering Excellence by Microsoft chairman Bill Gates. This award reflected the substantial improvement FxCop has made to development and testing practices of managed code at Microsoft. A version of FxCop is integrated with Visual Studio as a part of the Visual Studio Team System code analysis features. Another version is available as a free download.

The best part about FxCop is that you are not limited to just the rule libraries provided by Microsoft. You can easily define your own rules. What makes this possible is the "FxCop SDK," two DLLs which host the rule API. As with most APIs, you might expect detailed and comprehensive documentation. However, this is an area in which the SDK is lacking. While it has encouraged developers to write custom rules and has provided some sample code, Microsoft has not yet provided public documentation for the FxCop SDK. One ramification is that the API is subject to change without notice. Indeed, there have been significant changes in the API from version 1.35 to the latest version 1.36.

This document provides only a cursory look at the basics of using FxCop. Instead of diving deeply into features that are well-documented elsewhere, our focus is on development of custom rules, a task that many developers are—or should be—interested in doing.

Visual Studio Team System: The Code Analysis features of Visual Studio Team System 2008 Development Edition provide the same API for custom rules as FxCop 1.36. Therefore, most of what is described in this document applies to Visual Studio Team System as well. Note that the Team System includes additional rules out of the box that are not included with the free FxCop.

Caution

This document is based on research and experimentation. The FxCop APIs are not yet documented by Microsoft. As a result, expect some changes in future releases of FxCop. Furthermore, some parts of this document are likely to be incomplete or even incorrect. This document is a tutorial; to improve readability, this document sometimes presents assumptions as if they were verified facts.

The Possibilities

FxCop's capabilities encompass a great breadth and depth of code analysis features. The breadth of elements that you can check spans everything from naming conventions and parameter values to spelling and XML documentation comments. The depth encompasses high-level assembly metadata down to control structures and, ultimately, individual opcodes. Nearly all of FxCop's intrinsic power is made available for you to use in your custom FxCop solutions. By developing FxCop customizations, you can:

- Ensure that the names of controls on forms and web pages follow your naming conventions.
- Check that your preferred controls, components, and classes are used instead of alternatives.
- Inspect literal argument values being passed to your methods.

- Examine control structures, such as conditions and loops, to evaluate code metrics.
- Determine the callers and callees of methods.
- Spell-check text elements such as identifiers, literals, and resource strings.
- Verify that elements are properly documented with XML documentation comments.
- Build standalone tools that take advantage of FxCop's code analysis APIs.
- ... and many more possibilities.

Obtaining FxCop

The production version of FxCop can be obtained through the Microsoft Download Center. As of this writing, the latest production release of FxCop is version 1.36 and is available at http://www.microsoft.com/downloads/details.aspx?FamilyID=9aeaa970-f281-4fb0-aba1-d59d7ed09772.

The code analysis features of Visual Studio Team System 2008 Development Edition most closely correspond to those available in FxCop 1.36. The API for developing custom rules is the same in both. However, Visual Studio includes additional rules out of the box that are not included in the free FxCop.

FxCop 1.36 will run under and analyze programs written for .NET Framework versions 2.0, 3.0, and 3.5.

Note

Everything that you need to develop custom rules is included with the main FxCop download. While the API is officially referred to as the FxCop SDK, there are no separate SDK files to download.

Using FxCop: A Short Tutorial

This section describes how to get up and running quickly with FxCop. While this is not the main focus of this document, new FxCop users may find it helpful. For additional resources on using the program, refer to FxCop's official documentation.

Recall that FxCop checks compiled assemblies. Prior to running FxCop, you need to compile the program that you want to check. FxCop supports assemblies written in C#, VB.NET, C++/CLI, and other managed .NET languages. This is the result of FxCop directly checking assemblies instead of source code files. It is best to do a Debug build of your application and be sure that you generate the program database (PDB) files. These are the defaults for builds performed using Visual Studio. The program database files will supply FxCop with the debugging information it needs to trace problems back to line numbers in your source code.

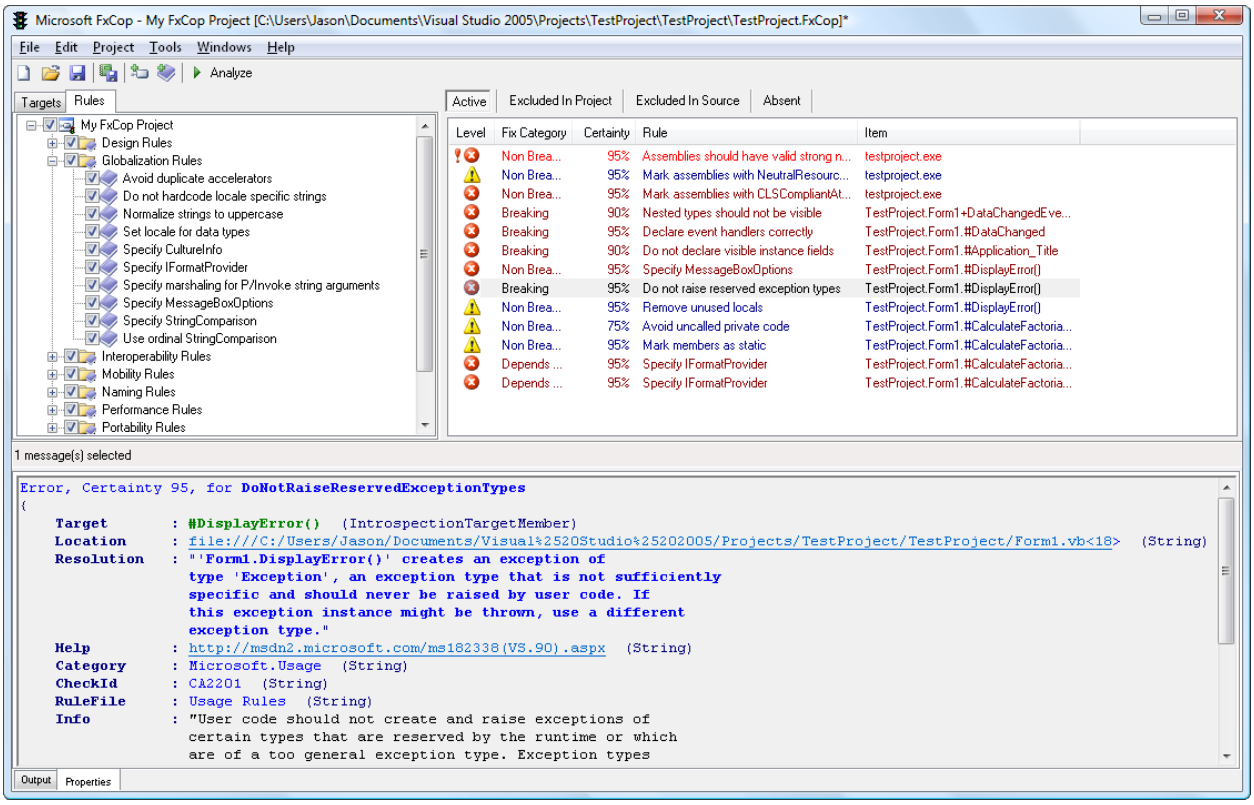
Once you have compiled your program, start FxCop. FxCop initially displays a blank project to which you can add one or more assemblies and choose which rules you want to enforce. Click the Project, Add Targets menu item to add your assembly (EXE or DLL) to the FxCop project. If your Visual Studio solution consists of several projects, you might want to add other assemblies at this time as well.

Next, you will probably want to shut off rule checking against automatically generated code. Examples of generated code include the designer parts of classes generated by the Visual Studio `Form` and `DataSet` designers. Such code violates some rules in minor ways. Since generated code is typically not modifiable, these violations are usually just noise. You can turn off checking generated code through the Project, Options menu item. Click the Spelling & Analysis tab and check the box for Suppress analysis results against generated code.

If you want to configure which rules are enabled, you can do so on the Rules tab in the main window. By default, all of the rules that ship with FxCop are enabled. Once you are finished configuring FxCop, you are ready to run it.

Click the Analyze button on the toolbar. FxCop will check the chosen assemblies against the chosen rules and output a report of any violations that were found.

Figure 1. FxCop Sample Screenshot



By clicking on a violation in the report, you can obtain detailed information about that violation. You may decide to:

1. Modify your source code to fix the violation. If possible, FxCop displays a hyperlink to your Visual Studio source file to assist you in locating the code that needs to be modified.

2. Enter an exclusion, which declares an exception to the rule. You can do this using the FxCop GUI which will record this in the FxCop project file. A better way is to use the `SuppressMessageAttribute` in your source code file as described below.

3. Ignore the violation for now. You can come back to it at a later time.

Although the FxCop GUI provides a way to record exclusions in the FxCop project file, a superior mechanism for declaring exceptions to rules is to use the `SuppressMessageAttribute` which comes with .NET in the `System.Diagnostics.CodeAnalysis` namespace. To use it, make note of the Category name (here, `Microsoft.Usage`) and the CheckId (here, `CA2201`) as displayed by FxCop.

Tip

You may find it a useful technique to temporarily mark exclusions using the FxCop GUI and later, at your convenience, replace them with `SuppressMessageAttribute` in your source code.

For C# code:

```
using System.Diagnostics.CodeAnalysis;
```

```
[SuppressMessage("Microsoft.Usage", "CA2201:DoNotRaiseReservedExceptionTypes")]
public void YourMethod()
{
     .
     .
     .
}
```

For VB.NET code:

```
Imports System.Diagnostics.CodeAnalysis

<SuppressMessage("Microsoft.Usage", "CA2201:DoNotRaiseReservedExceptionTypes")> _
Public Sub YourMethod()
{
     .
     .
     .
}
```

FxCop can automatically generate the correct `SuppressMessageAttribute` for you. Right-click on the problem in the FxCop GUI and choose Copy As, SuppressMessage. This will place the code declaring the `SuppressMessageAttribute` on the clipboard. If you are using C#, it will be exactly what you need. If you are using VB.NET, you will need to make some slight modifications by hand to convert it from C# to VB.NET. Usually, this involves changing the square brackets to angle brackets. Sometimes, you will also need to change `=` to `:=` if additional properties such as `MessageId` are included.

Generally, the `SuppressMessageAttribute` needs to be placed right before the declaration for the member identified in the Item column in the FxCop GUI. Usually, this will be a class or a method. There are some cases where placing the attribute in the correct location may be inconvenient or impractical. For instance, the member may be declared in a code file that was automatically generated by a tool, such as Visual Studio or a custom standalone utility. You can consider setting the option described earlier to instruct FxCop to not check the generated code. Nonetheless, your code generator might not mark its output with a `System.CodeDom.Compiler.GeneratedCodeAttribute` and, as a result, it might be incompatible with this exclusion option. A second case is the individual fields that implement `Control` members in the Windows Forms designer because these are not considered to be generated code. A third case is problems are associated with the module itself instead of a specific member. In all of these cases you can tell FxCop to generate a suppression for the problem that you can place at the module-level.

A module-level suppression means that the `SuppressMessageAttribute` is placed at the module-level in your code, prefixed by `module:`. The name "module-level" can be confusing; such a suppression does not have global effects. Just like ordinary `SuppressMessageAttribute` declarations, a single problem instance is suppressed. You specify the `Scope` and `Target` properties (and potentially a `MessageId`) to identify the precise problem to suppress. To automatically generate a module-level `SuppressMessageAttribute`, right-click on the problem and choose Copy As, Module-level SuppressMessage.

The following accomplishes the same suppression as the previous example using a module-level `SuppressMessageAttribute` in C#:

```
using System.Diagnostics.CodeAnalysis;

[module: SuppressMessage("Microsoft.Usage", "CA2201:DoNotRaiseReservedExceptionTypes",
  Scope="member",
  Target="TestProject.Form1.#YourMethod()")]
```

Equivalently, in VB.NET:

```
Imports System.Diagnostics.CodeAnalysis

<Module: SuppressMessage("Microsoft.Usage", "CA2201:DoNotRaiseReservedExceptionTypes", _
  Scope:="member", _
  Target:="TestProject.Form1.#YourMethod()")>
```

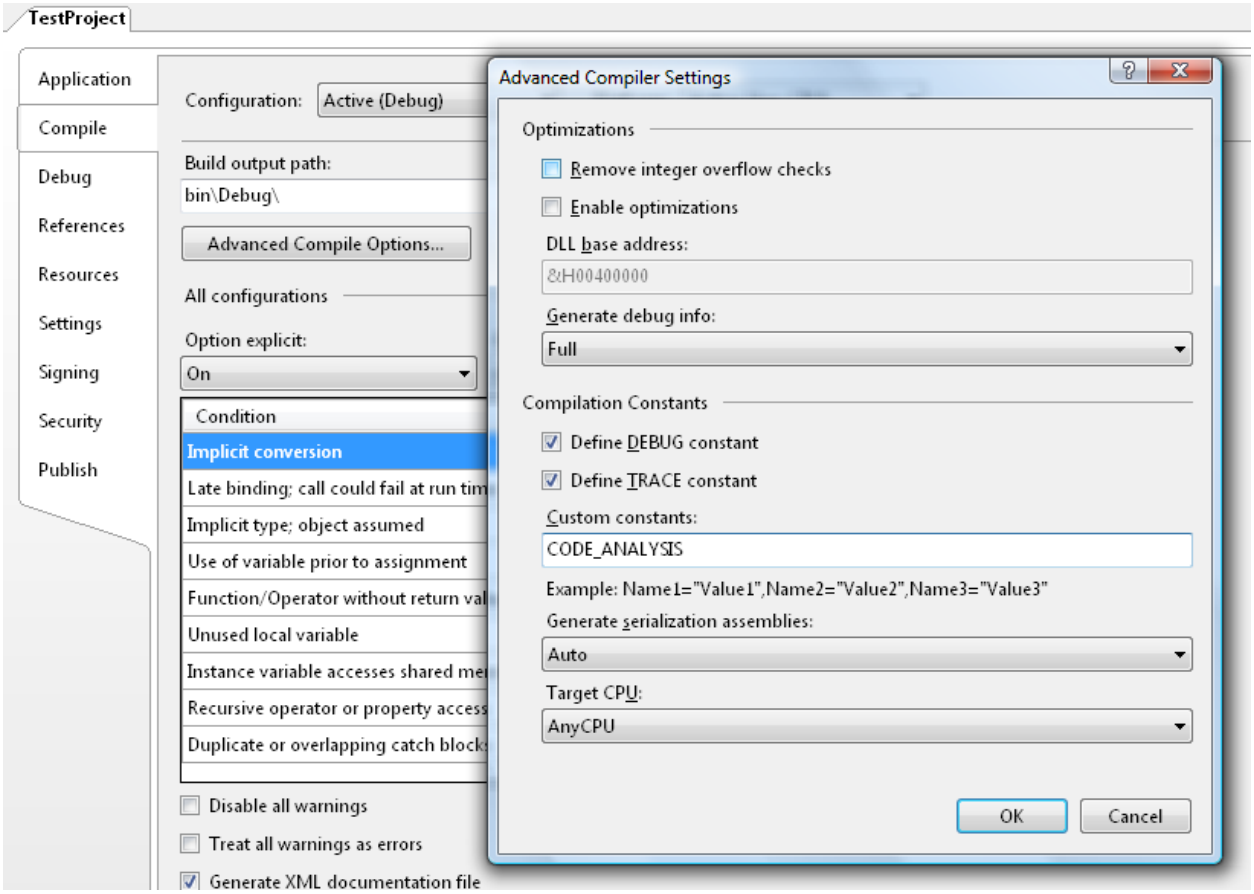
Module-level `SuppressMessageAttribute` declarations must come before most other code elements in a source file. In particular, it must come before `namespace` and `class` blocks.

Tip

You might choose to put all of your module-level `SuppressMessageAttribute` declarations in their own, separate source file. For example, a SuppressMessages.cs file. Optionally, you might decide that you want to always use module-level suppressions. In this case, you would be able to put all of your suppressions for the assembly in the same source file.

In order for the `SuppressMessageAttribute` declarations to take effect, you need to declare the `CODE_ANALYSIS` custom constant in your Visual Studio project's properties under Advanced Compile Options.

Figure 2. Custom Constant for SuppressMessageAttribute



FxCop 1.36 includes documentation that describes the FxCop user interface (and the command line interface as well) in detail. The documentation includes detailed information describing the reasoning behind all of the predefined rules and advice on modifying your code to resolve violations of them. This help is available within the program and online at http://msdn2.microsoft.com/en-us/library/bb429476.aspx.

Assembly Structure

FxCop rules are organized into DLL assemblies (which can be created as Class Library projects in Visual Studio). Each assembly, when loaded into FxCop, displays its rules in the GUI as leaf nodes in a hierarchical tree.

Each rule assembly should directly reference two assemblies provided with FxCop. These two assemblies are located in the Program Files\Microsoft FxCop 1.36 folder:

- `FxCopSdk.dll`
- `Microsoft.Cci.dll`

The only namespace that needs to be imported to use the API is `Microsoft.FxCop.Sdk`.

Visual Studio Team System: To add a custom rule assembly to Visual Studio Code Analysis, copy the DLL to `c:\Program Files\Microsoft Visual Studio 9.0\Team Tools\Static Analysis Tools\FxCop\Rules`, where `c:\` reflects the drive where Visual Studio is installed.

Introspection Code Model

The .NET Framework includes an API for examining assembly metadata in the System.Reflection namespace. Rules in early versions of FxCop examined code directly through the reflection API. However, reflection has some limitations that become evident when used in a tool such as FxCop. These limitations primarily stem from the fact that the reflection API is designed to examine programs loaded by the CLR for execution whereas FxCop is charged with inspecting a program that is not about to be executed.

FxCop solves this problem by providing an alternate, similar technology called introspection. The API is similar to reflection but offers generally improved analysis features and is overall more suitable for FxCop than the reflection API. If you liked reflection, you'll love introspection.

The code model divides the metadata into a hierarchy of fundamental primitives known as nodes. Each node is represented by a .NET class in the introspection API that is used to access the relevant metadata attributes. In cases where the node class name would conflict with a .NET Framework class, the API appends the word "Node" to the class name. To keep this list concise, some of the less frequently used node types have been abridged where noted.

- `Node`
  - `AssemblyReference`
  - `AttributeNode`
  - `Expression`
    - (numerous)
  - `Instruction`
  - `Member`
    - `EventNode`
    - `Field`
    - `Method`
      - `InstanceInitializer`
      - `StaticInitializer`
    - `PropertyNode`
    - `TypeNode`
      - `ClassNode`
      - `DelegateNode`
      - `EnumNode`
      - `InterfaceNode`
      - `Struct`
      - (others)
  - `ModuleNode`
    - `AssemblyNode`
  - `ModuleReference`
  - `SecurityAttribute`
  - `Statement`

- (numerous)

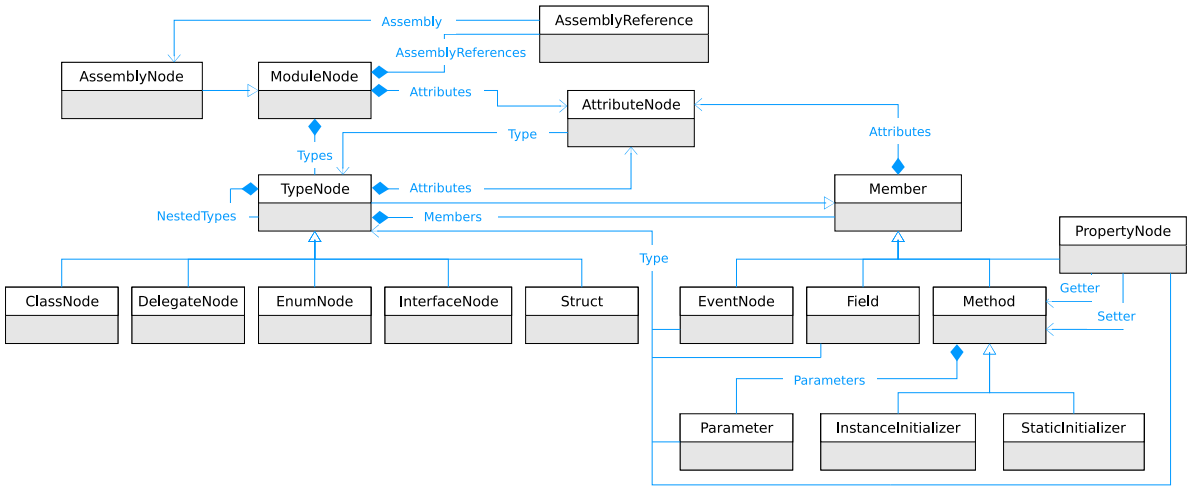
Note

Although .NET assemblies logically contain modules, `AssemblyNode` derives from `ModuleNode`, not the other way around. The inheritance is consistent with how the metadata is actually structured in assembly files. The main module is accessed directly through the `AssemblyNode`, which is convenient since most assemblies have only one module. This concept is explained in greater detail in a subsequent section.

The introspection model is much deeper than the reflection one. Introspection is capable of drilling down to the statement, expression, and CIL instruction levels, all with rich metadata along the way.

The above shows the "is a" relationships; it is also useful to know the "has a" relationships among the nodes so you can navigate from one node to another.

Figure 3. Object Model Relationships



Note

Although most of the node types are easily recognized, you might not be familiar with the terms "instance initializer" and "static initializer." An instance initializer is just an ordinary "constructor." A static initializer, also known as a "class constructor," contains code that initializes fields declared as `static` in C# or `Shared` in VB.NET.

Introspector

This author has developed a free utility, called Introspector, that lets you browse the introspection object model of assemblies on your system. You might find this tool useful when learning the introspection API and when developing custom rules. It eliminates some of the trial-and-error part of the development process by enabling you see beforehand the contents of the collections and properties made available by the introspection API.

If you are familiar with Lutz Roeder's .NET Reflector tool (http://www.aisto.com/roeder/dotnet), Introspector provides far less analysis and navigation features (and no decompiler). Instead, it aims at presenting the introspection objects with minimal transformation so that you can readily use what you discover in the tool to write custom rules against the introspection API.

The user interface consists of:

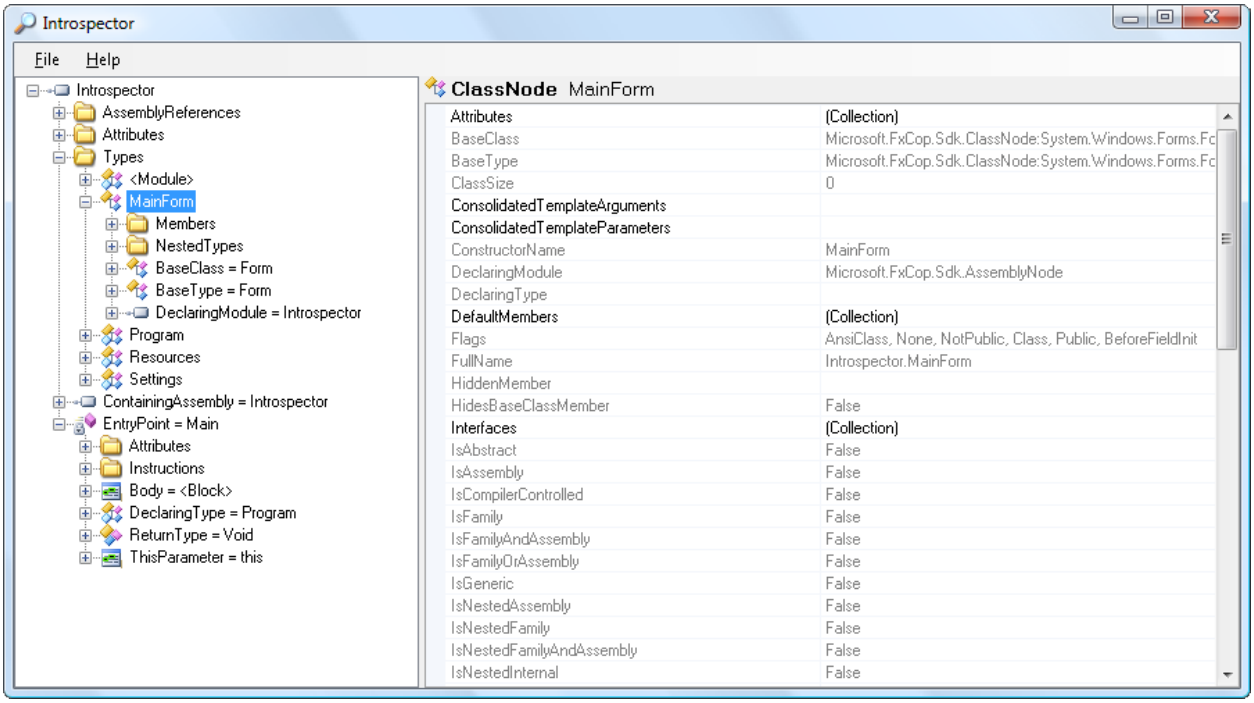
- A menu bar, containing a File menu that lets you browse for and open assemblies (.NET EXEs and DLLs).
- A tree view which contains a hierarchy of items representing the introspection `MetadataCollection` and `Node` objects for each open assembly.
- A property grid which displays the properties of the object selected in the tree view.

Introspector is available for download at http://www.binarycoder.net/fxcop.

Tip

As you read the next sections on the introspection API, try using Introspector to explore the corresponding objects firsthand.

Figure 4. Introspector Screenshot



Identifiers

Most framework metadata objects have names associated with them. In the case of data types (classes, structures, and so on), these are two part names. The first part is the namespace name and the second part is the unqualified name (simply *the* "name"). The introspection API represents names using the `Identifier` class. The `Identifier` representing the unqualified name can be obtained through the `Name` property. The `Identifier` representing the namespace name, if present, can be obtained through the Namespace property. This `Identifier` class contains a `Name` property which is the `String` form of the name.

Important

The `Identifier` class does not override the `Object.Equals` method or implement equality operators. To compare if two identifiers are the same name, check that `id1.Name == id2.Name`.

The `Identifier` class also provides a `UniqueIdKey` integer property that is useful if you need to make many comparisons for a specific name. It might provide a performance benefit over comparing entire strings.

Many nodes also provide a `FullName` property of type `String` that contains the fully qualified name which is the dotted join of the namespace, type name, and (if applicable) member name.

Sometimes, you will need to acquire instances of `Identifier` to pass to API methods. You can obtain an `Identifier` given a `String` using the static `Identifier.For` method.

Working With Assemblies, Modules, and Types

A .NET assembly is a .NET EXE or DLL that includes of one or more modules whose types are aggregated as one logical package. Modules are the highest level containers of all `TypeNode` objects. All non-nested types in the assembly are exposed directly via the module. Typically, an assembly has only one module. There are two exceptions: The first is when the assembly references .netmodule files that span the assembly over multiple physical files. This is typically—if there is anything typical about this—accomplished using the C# or VB.NET compiler's `/addmodule` option. The second is when the assembly linker **al.exe** or a similar tool is used to produce a single EXE or DLL that combines multiple modules into a single physical assembly file.

In the introspection model, `AssemblyNode` inherits from `ModuleNode`. The main module in the physical assembly is accessed directly through the `AssemblyNode`. The `ModuleNode` objects for other modules can be obtained through the `AssemblyNode.ModuleReferences` property or more explicitly with the `GetNestedModule` method. The `ModuleNode.Types` property provides the `TypeNode` objects for all of the types (classes, structures, and so on) contained in the module. The `GetType` method that takes two `Identifier` parameters can be used to obtain a type given its namespace name and type name.

Important

When the `Types` property and the `GetType` method are invoked on an `AssemblyNode`, they only return types in the main module. To properly support assemblies containing multiple modules, you may need to check the main module (the `AssemblyNode`) and all of the modules in the assembly's `ModuleReferences` collection when looking for a type.

There are many ways to obtain a reference to an AssemblyNode. These include:

1. Navigate to the `AssemblyNode` from a child `Node` that you already have. For example, through a parameter passed to you by FxCop. Often, this means using the `TypeNode.DeclaringModule` property. Once you have the `ModuleNode`, you can use its `ContainingAssembly` property to get the `AssemblyNode`.

2. Use the `FrameworkAssemblies` class to obtain references to a small number of frequently used assemblies through its `Mscorlib`, `System`, `SystemConfiguration`, `SystemData`, `SystemWeb`, and `SystemWindowsForms` static properties.

3. Use the `ModuleNode.ReferencedAssemblies` to get to an assembly referenced by a module that you already have.

4. Use the static `AssemblyNode.GetAssembly` method to obtain an `AssemblyNode` based on the assembly's path and filename on disk. Since other methods are typically more efficient and easier to use, this method is somewhat of a last resort.

The `FrameworkTypes` class provides commonly used `TypeNode` objects analogous to how `FrameworkAssemblies` provides commonly used `AssemblyNode` objects. The types are not limited to the .NET Base Class Library; they also include types from higher layers of the .NET Framework such as Windows Forms and ASP.NET. It includes over a hundred `TypeNode` objects, among them are `Component`, `Enum`, `GenericList` (which corresponds to `List<T>`), `DataSet`, `Form`, and `String`.

Sometimes, you want to know if a type implements an interface or has a specific certain base class in its inheritance hierarchy. This is equivalent to asking if the type is assignable to the interface or base class in question. For example:

```
bool descendsFromControl = myType.IsAssignableTo(FrameworkTypes.Control)
```

A special type, `FrameworkTypes.Void`, is used for return values of methods that do not return any value such as `void` methods in C# and `Sub` methods in VB.NET.

Unconstructed generic types, such as `List<T>`, have an `IsGeneric` property of `true`. The type parameters (here, *T*) are available through the `TemplateParameters` property. Constructed generic types, such as `List<String>` are considered to be "structural types" and are discussed in the next section.

Structural Types

.NET programs can make use of some special types that are not directly defined in assemblies but are closely related to types defined in assemblies. These types, which often arise as parameters or local variables, have `TypeNode.IsStructural` set to true.

Arrays	Array types, such as `byte[]`, are represented by the `ArrayType` node. The `ElementType` property reflects the underlying type of the elements in the array.
References	Parameters that pass a value by reference (`ref` and `out` in C#, `ByRef` in VB.NET) are represented by the `Reference` node. The `BaseType` property reflects the underlying type of the referent.
Pointers	Pointers, which arise in `unsafe` C# and C++ code, are represented by the `Pointer` node. The `BaseType` property reflects the type of the pointer's target.
Function Pointers	Function pointers, which arise in C++ code, are represented by the `FunctionPointer` node. It has `ParameterType` and `ReturnType` properties that describe the function signature.
Generic Type Parameters	Generic type parameters (such as *T*, *K*, and *V*) are represented by `TypeParameter` and `ClassParameter` nodes. If the type parameter has a constrained base type, `ClassParameter` is used and the base type is available through the `BaseType` property. Otherwise, `TypeParameter` is used and `BaseType` is `null`. If the type parameter is constrained to implement one or more interfaces, these interfaces are available through the `Interfaces` property.
Constructed Generic Types	So-called "constructed generic types", or generic types with their type parameters bound to specific types such as `List<String>`, are also considered structural. These are represented by an ordinary type node, such as `ClassNode` or `Struct`. The `Template` property identifies the related unconstructed generic type. The `TemplateArguments` property provides the specific type arguments that were bound against this template (in the `List<String>` example, these would be a `ClassNode` for `String`).
Type Modifiers	Type modifiers tend to appear when using C++. There are two kinds of type modifier nodes, `OptionalModifier` and `RequiredModifier`, both of which inherit from `TypeModifier`. These nodes give the type of the modifier as the `Modifier` property and the type being modified as the `ModifiedType` property. For example, in C++, a variable declared as `const int x` would have a `Modifier` that corresponds to

the `System.Runtime.CompilerServices.IsConst` class and a `ModifiedType` that corresponds to the `System.Int32` structure.

Access Modifiers

Member nodes such as data types, fields, properties, methods, and events have access modifiers associated with them. These define the level of object-oriented encapsulation implemented by the member. The introspection API provides properties to determine which access modifier is applied. You are familiar with these; however each .NET language has slightly different names for the modifiers. The following table shows you how to convert between the language-specific modifiers and the properties provided by `Member` nodes.

Table 1. Access Modifiers

Introspection Property	C#	VB.NET	C++/CLI
`IsPublic`	`public`	`Public`	`public`
`IsFamilyOrAssembly`	`protected internal`	`Protected Friend`	`public protected`
`IsFamily`	`protected`	`Protected`	`protected`
`IsFamilyAndAssembly`	n/a	n/a	`protected private`
`IsAssembly`	`internal`	`Friend`	`internal`
`IsPrivate`	`private`	`Private`	`private`

Important

On any member node, exactly one of these properties is `true` and all of the others are `false`. For example, to determine if a member is visible in inherited types, you need to check using the expression `(node.IsFamilyOrAssembly || node.IsFamily || node.IsFamilyOrAssembly)` in C# or `(node.IsFamilyOrAssembly OrElse node.IsFamily OrElse node.IsFamilyOrAssembly)` in VB.NET. It is not sufficient to check just for `IsFamily`.

Sometimes, instead of using the above properties, you only care whether or not the member is ultimately exposed for use outside the assembly. You may want to enforce stricter FxCop rules for the members that constitute your published API versus members that are just internal implementation details. Introspection considers such members "externally visible" and provides the `IsVisibleOutsideAssembly` property. For example, a `public` method in a `public` class is considered externally visible as is a `protected` method in a `public` class. However, a `public` method in a `private` class is not externally visible.

Rule Metadata XML

Each rule assembly contains an XML resource defining metadata for all of the rules in the assembly. In Visual Studio, the addition of this resource to your project can be accomplished by adding an XML file and then marking the file as an "Embedded Resource" in the Visual Studio properties window.

```
<?xml version="1.0" encoding="utf-8"?>
<Rules FriendlyName="">
  <Rule TypeName="" Category="" CheckId="">
    <Name></Name>
    <Description></Description>
    <Url></Url>
    <Resolution></Resolution>
    <MessageLevel Certainty=""></MessageLevel>
    <Email></Email>
    <FixCategories></FixCategories>
    <Owner></Owner>
```

```
  </Rule>
</Rules>
```

The `Rule` tag may be repeated as many times as necessary. None of these tags should be interpreted as optional. In the event that a value is not required, the tag should still be present with its content left empty.

`FriendlyName`	The display name of the rule library as it will appear in the FxCop GUI rules tree.
`TypeName`	The name of the .NET class that implements the rule. Do not qualify with a namespace.
`Category`	The category of the rule. This should generally be a programmatic version of `FriendlyName`. It will typically be the same for all rules in the same assembly. The `Category` is important to end users when they declare exceptions to your rule (more on that later).
`CheckId`	An identifier that distinguishes the rule from all others in the same `Category`. Traditionally, this is two letters followed by a four digit number. Like `Category`, this is important when users declare exceptions to your rule.
`Name`	The display name of your rule as it will appear in the leaf level of the FxCop GUI rules tree.
`Description`	A description of your rule. Traditionally, these are detailed explanations providing background on why the rule exists and how users should go about remedying violations of the rule. FxCop will display these in the bottom panel of its GUI when a violation of the rule is highlighted.
`Url`	A URL associated with the rule that can be used to provide even more information about the rule than you would ordinarily include in the `Description` element. `Url` can be left empty if no URL is desired.
`Resolution`	A short suggestion on how to fix the violation. It may contain `String.Format` placeholders whose values are to be supplied programmatically by the rule when it detects the violation. FxCop will show the resolution to the user when the rule is violated. (Multiple resolutions for a rule may be specified if a `Name` attribute is included on the `Resolution` tag. The rule selects one of these resolutions when it detects the violation.)
`MessageLevel`	The severity of the error: `CriticalError`, `Error`, `CriticalWarning`, or `Warning`.
`Certainty`	The `certainty` is an integer from `0` to `100` that indicates how often the rule typically detects a real problem in the code. This tag is purely informational.
`Email`	An email address to be associated with the rule for display to the user. Often left empty.
`FixCategories`	Characterizes the impact of fixing the rule: `Breaking`, `NonBreaking`, or `DependsOnFix`. This is purely informational.
`Owner`	The person who "owns" the rule for display to the user. Often left empty.

Important

Be sure to observe the proper element/attribute capitalization and structure when declaring your rules. If the XML is not formatted correctly, FxCop might not be able to load your rules. Often, FxCop will ignore rules without displaying any error messages. If this happens to you, carefully review the XML.

All classes in your custom rule assembly that implement `IRule` (typically by inheriting from `BaseIntrospectionRule`) must have a corresponding `Rule` tag that matches on `TypeName`.

Note

Rule classes do not need to be declared `public` because FxCop instantiates them using reflection.

Figure 5. Project With Rule Metadata XML Embedded Resource

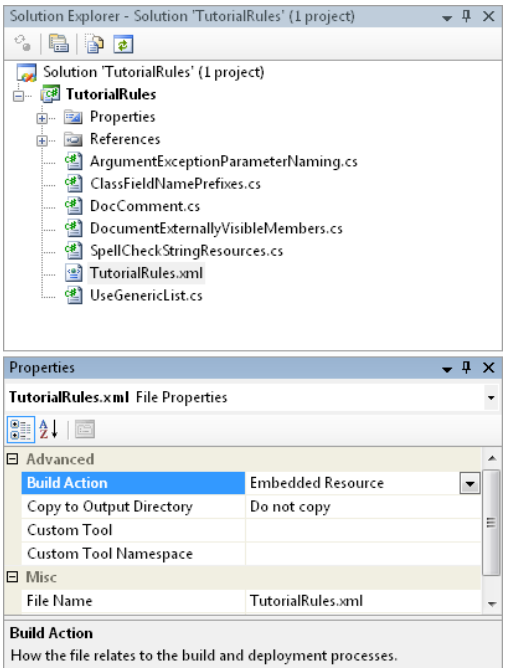


Figure 6. Sample Metadata XML

```
<Rules FriendlyName="Tutorial Rules">
  <Rule TypeName="ClassFieldNamePrefixes" Category="Tutorial" CheckId="TT1010">
    <Name>Class field names should use the 'm_' and 'g_' suffixes.</Name>
    <Description>
      Instance fields in classes should begin with the 'm_' prefix.
      Static fields in classes should begin with the 'g_' prefix.
    </Description>
    <Url></Url>
    <Resolution Name="Static">Prefix the static field '{0}' with 'm_'.</Resolution>
    <Resolution Name="Instance">Prefix the instance field '{0}' with 'g_'.</Resolution>
    <MessageLevel Certainty="95">Warning</MessageLevel>
    <Email></Email>
    <FixCategories>Breaking</FixCategories>
    <Owner></Owner>
  </Rule>
</Rules>
```

Rule Initialization

Custom rules inherit from `BaseIntrospectionRule`. Its constructor takes three arguments:

- The name of the rule, which must match the `TypeName` specified in the rule metadata XML.
- The name of the metadata XML resource. For assemblies compiled using C#, this starts with the assembly name followed by a period and then the embedded resource file name with the `.xml` extension is omitted.
- The assembly containing the metadata XML resource.

FxCop creates one instance of your rule class for each analysis thread. You may optionally override the `BeforeAnalysis` and `AfterAnalysis` methods. These methods are called once during the lifetime of each rule object (in other words, once per analysis thread).

Warning

FxCop API objects are generally *not* thread-safe. Do not store them in globals declared using the `static` (C#) or `Shared` (VB.NET) keywords.

`BaseIntrospectionRule` has an overridable `TargetVisibility` property. If you want your rule to check only those elements that are exposed to third parties (for example, `public` methods in `public` classes) override this property to return `TargetVisibilities.ExternallyVisible`. If you do not override this property, FxCop uses `TargetVisibilities.All` by default. The following is a complete listing of all of the possible visibilities:

- `All`
- `AnonymousMethods`
- `ExternallyVisible`
- `None`
- `NotExternallyVisible`
- `Obsolete`
- `Overridable`

Note

It can be more efficient and convenient to override `TargetVisibility` on your rule instead of writing a lot of checks against `TypeNode.IsVisibleOutsideAssembly` properties.

Figure 7. Rule Boilerplate Code

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.FxCop.Sdk;

namespace TutorialRules
{
  class MyCustomRule : BaseIntrospectionRule
  {
    public MyCustomRule() :
      base("MyCustomRule", "TutorialRules.TutorialRules",
      typeof(MyCustomRule).Assembly)
    {
    }

    public override TargetVisibilities TargetVisibility
    {
      get
      {
        return TargetVisibilities.All;
      }
    }

    public override void BeforeAnalysis()
    {
      // Do once-per-thread initialization.
    }

    public override void AfterAnalysis()
    {
      // Do once-per-thread cleanup. Rarely needed.
    }
  }
}
```

Check and Visit

During analysis, FxCop recursively traverses through the nodes that make up your assemblies. The most interesting nodes are `ModuleNode` objects, `TypeNode` objects (such as `ClassNode`), and `Member` objects (such as `PropertyNode` and `Method`).

When FxCop encounters such a node and the `TargetVisibility` is consistent with any visibility filtering that your rule has specified, it calls a `Check` method. You can override any (or all) of the following methods inherited from `BaseIntrospectionRule`. By default, these methods perform no operation unless you override them to implement your rule:

- `Check`(`ModuleNode` *`module`*) — Checks a .NET module (an assembly consists of one or more modules).
- `Check`(`Resource` *`resource`*) — Checks a resource (such as a `Bitmap`) defined in a module.
- `Check`(`String` *`namespaceName`*, `TypeNodeCollection` *`types`*) — Checks types in a namespace.
- `Check`(`TypeNode` *`type`*) — Checks a type (such as `ClassNode`, `Struct`, `EnumNode`, or others).
- `Check`(`Member` *`member`*) — Checks a member (such as a `PropertyNode`, `Method`, `Field`, or others).
- `Check`(`Parameter` *`parameter`*) — Checks a parameter for a member.

Note

In general, use the most specific override possible when implementing your rule. For example, if checking that classes follow a naming convention, override `Check`(`TypeNode` type). If you override a method

that is too general and drill down through the nodes yourself, FxCop may have trouble identifying which node you are actually checking. This will affect how the results are displayed to the user.

When you override one of these methods, you should always return `this.Problems` in C# or `Me.Problems` in VB.NET. It is okay to return this way even if no problems were detected by your rule.

In some cases, you may want to check nodes that are deeper than the `Member` level such as individual `Statement` nodes. In this case, you can call a method that starts with "Visit" to initiate a drilldown. All child nodes will then be recursively visited. You can override `Visit` methods logically deeper in the node tree to execute code once nodes of those types are visited by the recursion. In such an override, you must call the base class implementation if you want recursion to continue deeper. For example, the following processes all `AssignmentStatement` nodes in a `Method`:

```
public override ProblemCollection Check(Member member)
{
  Method methd = member as Method;
  if (methd != null)
  {
    VisitStatements(methd.Body.Statements);
  }
  return this.Problems;
}

public override void VisitAssignmentStatement(AssignmentStatement assignment)
{
  // Process assignment statement. Add to this.Problems if problems are found.

  // Uncomment the following if you need to recurse deeper (for example, to the Expression level).
  // base.VisitAssignmentStatement(assignment);
}
```

Problems and Resolutions

Each rule has a `ProblemCollection` inherited from `BaseIntrospectionRule`. The rule is expected to add objects of type `Problem` to this collection as violations of the rule are detected during the traversal described in the preceding section.

When you construct a `Problem`, you select the `Resolution` string in your XML resource to be displayed to the user to assist the user with investigating and correcting the problem. If your rule only has one `Resolution` defined in its XML, use the rule's inherited `GetResolution` method to retrieve it. Otherwise, use `GetNamedResolution`, passing the value of the `Name` attribute in your XML that distinguishes the particular resolution. The methods for obtaining the resolution from the XML also take an optional paramarray of values to fill positional placeholders (such as `{0}`, `{1}`, etc.) in the resolution. The syntax is the same one used by `String.Format`.

Now for something a little tricky. Some overloads of the `Problem` constructor have a `String` *`id`* parameter. Usually you do not need to use these overloads but can instead stick to the simple one that just takes a `Resolution` object. The case where you need to use *`id`* is when a single `Check` method has the potential to add more than one `Problem` to the `ProblemCollection`. For example, consider the case where you might examine all of a method's local variables during `Check`(`Member` *`member`*) and potentially add anywhere from zero `Problem` objects up to the total number of local variables in the method. The *`id`* parameter lets you uniquely identify each of the `Problem` objects you add within the scope of the same `Check` method. In this local variable example, you might choose to use the name of the local variable as the *`id`*.

Note

If you notice that only one `Problem` is displayed in the FxCop GUI even though your rule is outputting more than one `Problem`, your implementation is likely not assigning unique identifiers in a case where they are required.

Debugging FxCop Rules

FxCop rules can be debugged using the debugger that is built in to Visual Studio. The key to debugging rules is to understand that you need to run the FxCop GUI application in the debugger since that application in turn loads your rules assembly. There are two basic ways to perform the debugging: by attaching to a running FxCop executable or by starting FxCop directly from your Visual Studio project.

To attach to a running FxCop executable, start FxCop normally. Next, ensure that your rules project is open in Visual Studio. Click the Tools, Attach to Process... menu item. Highlight the `FxCop.exe` process and click Attach. Now, perform an analysis using FxCop. Any breakpoints that you have set in your code will step into the debugger as they are encountered.

Alternatively, you can configure your rules class library project to start FxCop for debugging whenever you use Visual Studio to run your project (for example, by clicking the Run toolbar button or pressing the **F5** key.) Click the Project, <projectname> Properties menu item. Go to the Debug page. Choose Start external program and enter the path and filename for the FxCop program, such as **`C:\Program Files\Microsoft FxCop 1.36\FxCop.exe`**.

Tip

To make debugging more convenient, consider developing an FxCop project that uses your rules assembly. Then, set Command line arguments in the Debug page to the path and filename of this project. FxCop will automatically open this project when it is launched, saving you from having to perform basic setup tasks in FxCop every time you run it under the debugger.

Callers and Callees

FxCop has the ability to determine which methods call a given method (the callers) and which methods a given method calls (the callees). The ability to determine the callers is built-in to the API. The ability to determine the callees is easily accomplished with some custom code.

To determine the callees for a method, use the following static method of the `CallGraph` class. This method searches all assemblies that are in your FxCop project. If there are no callers, the returned `MethodCollection` contains zero `Method` objects.

`public static MethodCollection CallersFor(Method callee)`

To determine the callees of a method, visit every `MethodCall` expression in that method. The following shows a possible implementation of a utility, `Callees.CalleesFor`, to obtain the callees.

```
using System;
using System.Collections.Generic;
using Microsoft.FxCop.Sdk;

namespace TutorialRules
{
  public sealed class Callees
  {
    public static IList<Method> CalleesFor(Method caller)
    {
      if (caller == null)
        throw new ArgumentNullException("caller");

      CalleeVisitor cv = new CalleeVisitor();
      cv.VisitStatements(caller.Body.Statements);

      return cv.Callees;
    }

    private class CalleeVisitor : BinaryReadOnlyVisitor
    {
```

```
      private List<Method> m_Callees = new List<Method>();
      private Dictionary<int, int> m_CalleeIds = new Dictionary<int, int>();

      public IList<Method> Callees
      {
        get
        {
          return m_Callees;
        }
      }

      public override void VisitMethodCall(MethodCall call)
      {
        base.VisitMethodCall(call);

        MemberBinding mb = call.Callee as MemberBinding;
        if (mb == null)
          return;

        Method methd = mb.BoundMember as Method;
        if (methd == null)
          return;

        int key = methd.UniqueKey;

        if (!m_CalleeIds.ContainsKey(key))
        {
          m_Callees.Add(methd);
          m_CalleeIds.Add(key, key);
        }
      }
    }

    // Prevent instantiation of this class; all members are static.
    private Callees()
    {
    }
  }
}
```

Checking Documentation Comments

Note

This section is optional. It discusses a useful, advanced application of custom FxCop rules to check XML documentation comments. If this is your first time reading this document, I strongly recommend skipping to the examples.

With some work, FxCop custom rules can be developed to check the XML documentation files generated by .NET compilers. For example, by checking assemblies against XML documentation files, you can verify that:

- Every member has been documented.
- All method parameters have been documented.
- Method return values have been documented.
- Namespaces have been documented.

FxCop does not provide any built-in APIs to access the XML documentation file that is output by the compiler alongside the assembly. Fortunately, being XML, that file is fairly simple to parse using `XmlDocument` and other standard .NET XML APIs. The most significant hurdle lies in turning the names members (classes, methods, properties, and so on) into member "ID strings". Such strings are similar to the strings returned by the `Member.FullName` property. However, FxCop does not provide the right conversion function to the format used by documentation comments, and its implementation is relatively tricky. The syntax of ID strings is described by Microsoft under

"Processing the XML File (C# Programming Guide)" at http://msdn2.microsoft.com/en-us/library/fsbx0t7x.aspx. Unfortunately, this official documentation does not completely describe the syntax required to support generic types, and it has some ambiguities and inconsistencies.

In this chapter, we present code that can generate a member ID string from a (hopefully arbitrary) `Member` node. I say hopefully due to the sheer complexity of this task. It is possible that some esoteric cases are not handled correctly by the code given here. Please report any such cases that you find.

This implementation differs from or expands on the Microsoft documentation mentioned above in at least the following ways:

1. Parameters of `ELEMENT_TYPE_PINNED` types are not supported. I cannot figure out any way to generate such parameters in compilers. I think pinning applies only to local variables, not parameters. Thus, it looks like it is not applicable to XML documentation.

2. Parameters of `ELEMENT_TYPE_GENERICARRAY` types are not supported. This term is absent from the ECMA CLI specification; I have no idea what the documentation is referring to.

3. When a function pointer does have any parameters, the parameters are represented as `(System.Void)`. Although the documentation states that the parentheses and parameters should be omitted in this case, this implementation is consistent with how the C++ compiler actually behaves.

4. Template arguments are rendered within curly braces. This facet is not explained in the documentation.

5. The names of methods with template parameters are suffixed with two backticks followed by the number of template parameters. The template parameters themselves are referenced with a double backtick notation. These facets are not explained in the documentation.

6. The lower bounds of `ELEMENT_TYPE_ARRAY` arrays are always specified (often as `0`). This seems consistent with the C# compiler.

The source code is given below. We will use this source code in a later example that demonstrates how to check that all externally visible members are documented.

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Text;

using Microsoft.FxCop.Sdk;

namespace TutorialRules
{
  sealed class DocComment
  {
    public static string GetMemberID(Member member)
    {
      char ch;
      TypeNode declaringType = member.DeclaringType;
      List<TypeNode> parentTypes = new List<TypeNode>();
      List<TypeNode> typeTemplateParameters = new List<TypeNode>();
      List<TypeNode> memberTemplateParameters = new List<TypeNode>();
      StringBuilder sb = new StringBuilder();

      if (member == null)
        throw new ArgumentNullException("member");

      // Determine prefix character.

      switch (member.NodeType)
      {
        case NodeType.Class:
        case NodeType.Interface:
        case NodeType.Struct:
        case NodeType.EnumNode:
```

```
    case NodeType.DelegateNode:
      ch = 'T';
      break;
    case NodeType.Field:
      ch = 'F';
      break;
    case NodeType.Property:
      ch = 'P';
      break;
    case NodeType.Method:
    case NodeType.InstanceInitializer:
    case NodeType.StaticInitializer:
      ch = 'M';
      break;
    case NodeType.Event:
      ch = 'E';
      break;
    default:
      throw new ArgumentException("Unsupported NodeType.", "member");
  }

  // Determine all parent types of this potentially nested type.

  for (TypeNode current = declaringType; current != null; current = current.DeclaringType)
  {
    parentTypes.Add(current);
  }
  parentTypes.Reverse();

  // Collect all template parameters for the types.

  foreach (TypeNode type in parentTypes)
  {
    if (type.TemplateParameters != null)
    {
      typeTemplateParameters.AddRange(type.TemplateParameters);
    }
  }

  // Collect all template parameters for the method.

  switch (member.NodeType)
  {
    case NodeType.Method:
    case NodeType.InstanceInitializer:
    case NodeType.StaticInitializer:
      Method method = (Method)member;

      if (method.TemplateParameters != null)
      {
        memberTemplateParameters.AddRange(method.TemplateParameters);
      }
      break;
  }

  // Output full method name.

  sb.Append(ch);
  sb.Append(':');
  if (declaringType == null)
  {
    TypeNode type = member as TypeNode;
    if (type != null)
    {
      if (type.Namespace.Name.Length != 0)
      {
        sb.Append(type.Namespace.Name);
        sb.Append('.');
      }
    }
  }
  else
```

```
    {
      sb.Append(declaringType.FullName.Replace('+', '.'));
      sb.Append('.');
    }
    sb.Append(member.Name.Name.Replace('.', '#'));

    // Output number of template parameters.

    if (memberTemplateParameters.Count != 0)
    {
      // Undocumented: based on output from MS compilers.
      sb.AppendFormat(CultureInfo.InvariantCulture, "``{0}", memberTemplateParameters.Count);
    }

    // Output parameters.

    ParameterCollection parameters;

    switch (member.NodeType)
    {
      case NodeType.Property:
        parameters = ((PropertyNode)member).Parameters;
        break;
      case NodeType.Method:
      case NodeType.InstanceInitializer:
      case NodeType.StaticInitializer:
        parameters = ((Method)member).Parameters;
        break;
      default:
        parameters = null;
        break;
    }

    if (parameters != null && parameters.Count != 0)
    {
      bool comma = false;
      sb.Append('(');
      foreach (Parameter parameter in parameters)
      {
        if (comma)
        {
          sb.Append(',');
        }
        sb.Append(GetStringForTypeNode(parameter.Type,
          typeTemplateParameters, memberTemplateParameters));
        comma = true;
      }
      sb.Append(')');
    }

    // Output return type (for conversion operators).

    if (member.NodeType == NodeType.Method && member.IsSpecialName &&
      (member.Name.Name == "op_Explicit" || member.Name.Name == "op_Implicit"))
    {
      Method convOperator = (Method)member;

      sb.Append('~');
      sb.Append(GetStringForTypeNode(convOperator.ReturnType,
        typeTemplateParameters, memberTemplateParameters));
    }

    return sb.ToString();
  }

  private static string GetStringForTypeNode(TypeNode type,
    List<TypeNode> typeTemplateParameters, List<TypeNode> memberTemplateParameters)
  {
    StringBuilder sb = new StringBuilder();

    switch (type.NodeType)
    {
```

```
/* Ordinary types */

case NodeType.Class:
case NodeType.Interface:
case NodeType.Struct:
case NodeType.EnumNode:
case NodeType.DelegateNode:
  if (type.DeclaringType == null)
  {
    if (type.Namespace.Name.Length != 0)
    {
      sb.Append(type.Namespace.Name);
      sb.Append('.');
    }
  }
  else
  {
    sb.Append(GetStringForTypeNode(type.DeclaringType,
      typeTemplateParameters, memberTemplateParameters));
    sb.Append('.');
  }

  if (type.IsGeneric)
  {
    String templateName = type.Template.Name.Name.Replace('+', '.');
    int pos = templateName.LastIndexOf('`');
    if (pos != -1)
    {
      sb.Append(templateName.Substring(0, pos));
    }
    else
    {
      sb.Append(templateName);
    }
  }
  else
  {
    sb.Append(type.Name.Name.Replace('+', '.'));
  }
  break;

/* Simple pointer / reference types */

case NodeType.Reference:
  sb.Append(GetStringForTypeNode(((Reference)type).ElementType,
    typeTemplateParameters, memberTemplateParameters));
  sb.Append('@');
  break;
case NodeType.Pointer:
  sb.Append(GetStringForTypeNode(((Pointer)type).ElementType,
    typeTemplateParameters, memberTemplateParameters));
  sb.Append('*');
  break;

/* Generic parameters */

case NodeType.ClassParameter:
case NodeType.TypeParameter:
  int index;
  if ((index = typeTemplateParameters.IndexOf(type)) != -1)
  {
    sb.AppendFormat(CultureInfo.InvariantCulture, "`{0}", index);
  }
  else if ((index = memberTemplateParameters.IndexOf(type)) != -1)
  {
    // Undocumented: based on output from MS compilers.
    sb.AppendFormat(CultureInfo.InvariantCulture, "``{0}", index);
  }
  else
  {
    throw new InvalidOperationException("Unable to resolve TypeParameter to a type argument.");
  }
```

```
      break;

    /* Arrays */

    case NodeType.ArrayType:
      ArrayType array = ((ArrayType)type);
      sb.Append(GetStringForTypeNode(array.ElementType,
        typeTemplateParameters, memberTemplateParameters));
      if (array.IsSzArray())
      {
        sb.Append("[]");
      }
      else
      {
        // This case handles true multidimensional arrays.
        // For example, in C#: string[,] myArray
        sb.Append('[');
        for (int i = 0; i < array.Rank; i++)
        {
          if (i != 0)
          {
            sb.Append(',');
          }

          // The following appears to be consistent with MS C# compiler output.
          sb.AppendFormat(CultureInfo.InvariantCulture, "{0}:", array.GetLowerBound(i));
          if (array.GetSize(i) != 0)
          {
            sb.AppendFormat(CultureInfo.InvariantCulture, "{0}", array.GetSize(i));
          }
        }
        sb.Append(']');
      }
      break;

    /* Strange types (typically from C++/CLI) */

    case NodeType.FunctionPointer:
      FunctionPointer funcPointer = (FunctionPointer)type;
      sb.Append("=FUNC:");
      sb.Append(GetStringForTypeNode(funcPointer.ReturnType,
        typeTemplateParameters, memberTemplateParameters));
      if (funcPointer.ParameterTypes.Count != 0)
      {
        bool comma = false;
        sb.Append('(');
        foreach (TypeNode parameterType in funcPointer.ParameterTypes)
        {
          if (comma)
          {
            sb.Append(',');
          }
          sb.Append(GetStringForTypeNode(parameterType,
            typeTemplateParameters, memberTemplateParameters));
          comma = true;
        }
        sb.Append(')');
      }
      else
      {
        // Inconsistent with documentation: based on MS C++ compiler output.
        sb.Append("(System.Void)");
      }
      break;
    case NodeType.RequiredModifier:
      RequiredModifier reqModifier = (RequiredModifier)type;
      sb.Append(GetStringForTypeNode(reqModifier.ModifiedType,
        typeTemplateParameters, memberTemplateParameters));
      sb.Append("|");
      sb.Append(GetStringForTypeNode(reqModifier.Modifier,
        typeTemplateParameters, memberTemplateParameters));
      break;
```

```
        case NodeType.OptionalModifier:
          OptionalModifier optModifier = (OptionalModifier)type;
          sb.Append(GetStringForTypeNode(optModifier.ModifiedType,
            typeTemplateParameters, memberTemplateParameters));
          sb.Append("!");
          sb.Append(GetStringForTypeNode(optModifier.Modifier,
            typeTemplateParameters, memberTemplateParameters));
          break;

        default:
          throw new ArgumentException("Unsupported NodeType.", "type");
      }

      if (type.IsGeneric && type.TemplateArguments.Count != 0)
      {
        // Undocumented: based on output from MS compilers.
        sb.Append('{');
        bool comma = false;
        foreach (TypeNode templateArgumentType in type.TemplateArguments)
        {
          if (comma)
          {
            sb.Append(',');
          }
          sb.Append(GetStringForTypeNode(templateArgumentType,
            typeTemplateParameters, memberTemplateParameters));
          comma = true;
        }
        sb.Append('}');
      }

      return sb.ToString();
    }

    // Prevent instantiation of this class; all members are static.
    private DocComment()
    {
    }
  }
}
```

About the Examples

The remainder of this document gives several example implementations of custom FxCop rules. These examples are not intended to be bulletproof rules for you to incorporate into your custom rule sets as is. Instead, they are presented in increasing order of complexity to illustrate key concepts that may be useful as you develop your own rules.

Example: Class Field Name Prefixes

It is a common practice to use the prefix `"m_"` (member) for instance fields and `"g_"` for static fields in classes.

The implementation is straightforward; a few special cases are recognized for compatibility with the Windows Forms designer and Visual Basic.

Rule XML:

```
<Rule TypeName="ClassFieldNamePrefixes" Category="Tutorial" CheckId="TT1010">
  <Name>Class field names should use the 'm_' and 'g_' suffixes.</Name>
  <Description>
    Instance fields in classes should begin with the 'm_' prefix.
    Static fields in classes should begin with the 'g_' prefix.
  </Description>
  <Url></Url>
  <Resolution Name="Static">Prefix the static field '{0}' with 'm_'.</Resolution>
  <Resolution Name="Instance">Prefix the instance field '{0}' with 'g_'.</Resolution>
  <MessageLevel Certainty="95">Warning</MessageLevel>
```

```
  <Email></Email>
  <FixCategories>Breaking</FixCategories>
  <Owner></Owner>
</Rule>
```

Rule Implementation:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.FxCop.Sdk;

namespace TutorialRules
{
  class ClassFieldNamePrefixes : BaseIntrospectionRule
  {
    public ClassFieldNamePrefixes() :
      base("ClassFieldNamePrefixes", "TutorialRules.TutorialRules",
      typeof(ArgumentExceptionParameterNaming).Assembly)
    {
    }

    public override ProblemCollection Check(Member member)
    {
      if (!(member.DeclaringType is ClassNode))
        return this.Problems;

      Field fld = member as Field;
      if (fld == null)
        return this.Problems;

      if (fld.IsStatic && !fld.Name.Name.StartsWith("g_", StringComparison.Ordinal) &&
        fld.Name.Name != "__ENCList")
      {
        this.Problems.Add(new Problem(this.GetNamedResolution("Static", fld.Name.Name)));
      }
      else if (!fld.Name.Name.StartsWith("m_", StringComparison.Ordinal) &&
        fld.Name.Name != "__ENCList" && fld.Name.Name != "components")
      {
        this.Problems.Add(new Problem(this.GetNamedResolution("Instance", fld.Name.Name)));
      }

      return this.Problems;
    }
  }
}
```

Example: Spell Checking String Resources

String resources are commonly used to centralize strings used by a program so that the text can be localized to multiple languages. This example uses the FxCop `WordParser` class and the FxCop spell checking facilities to verify that the words in these strings are spelled correctly. Note that this is a very simple implementation and more sophisticated checks, such as whether strings begin with a capital letter, are not included.

Rule XML:

```
<Rule TypeName="SpellCheckStringResources" Category="Tutorial" CheckId="TT1020">
  <Name>String resources should be spelled correctly</Name>
  <Description>
    String resources should be spelled correctly.
  </Description>
  <Url></Url>
  <Resolution>Check the spelling of the word '{0}'.</Resolution>
  <MessageLevel Certainty="75">Warning</MessageLevel>
  <Email></Email>
  <FixCategories>NonBreaking</FixCategories>
  <Owner></Owner>
</Rule>
```

Rule Implementation:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Resources;
using System.Text;

using Microsoft.FxCop.Sdk;

namespace TutorialRules
{
  class SpellCheckStringResources : BaseIntrospectionRule
  {
    public SpellCheckStringResources() :
      base("SpellCheckStringResources", "TutorialRules.TutorialRules",
      typeof(SpellCheckStringResources).Assembly)
    {
    }

    public override ProblemCollection Check(Resource resource)
    {
      using (MemoryStream mstream = new MemoryStream(resource.Data, false))
      {
        using (ResourceReader reader = new ResourceReader(mstream))
        {
          foreach (DictionaryEntry entry in reader)
          {
            string s = entry.Value as string;
            if (s != null)
            {
              foreach (String word in WordParser.Parse(s, WordParserOptions.None))
              {
                if (NamingService.DefaultNamingService.CheckSpelling(word) !=
                  WordSpelling.SpelledCorrectly)
                {
                  this.Problems.Add(new Problem(this.GetResolution(word), word));
                }
              }
            }
          }
        }
      }

      return this.Problems;
    }
  }
}
```

Example: Generic List<T> Instead of ArrayList

Generic collections provide many programming advantages and reduce bugs. In most cases, they should be used instead of their non-generic equivalents. This example checks for fields and local variables of type `ArrayList` and outputs warnings that they should be replaced with `List<T>`.

Rule XML:

```
<Rule TypeName="UseGenericList" Category="Tutorial" CheckId="TT1030">
  <Name>Use List&lt;T&gt; instead of ArrayList</Name>
  <Description>
    For greater type safety, use List&lt;T&gt; instead of ArrayList.
  </Description>
  <Url></Url>
  <Resolution>Replace ArrayList with List&lt;T&gt;.</Resolution>
  <MessageLevel Certainty="60">Warning</MessageLevel>
  <Email></Email>
  <FixCategories>DependsOnFix</FixCategories>
```

```
  <Owner></Owner>
</Rule>
```

Rule Implementation:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.FxCop.Sdk;

namespace TutorialRules
{
  class UseGenericList : BaseIntrospectionRule
  {
    private TypeNode m_ArrayList;

    public UseGenericList() :
      base("UseGenericList", "TutorialRules.TutorialRules",
      typeof(UseGenericList).Assembly)
    {
    }

    public override void BeforeAnalysis()
    {
      m_ArrayList = FrameworkAssemblies.Mscorlib.GetType(
        Identifier.For("System.Collections"), Identifier.For("ArrayList"));
    }

    public override ProblemCollection Check(Member member)
    {
      // Check field type, if this is a field.

      Field fld = member as Field;
      if (fld != null)
      {
        if (fld.Type == m_ArrayList)
        {
          this.Problems.Add(new Problem(this.GetResolution()));
        }
      }

      // Check local variable types, if this is a method.

      Method methd = member as Method;
      if (methd != null && methd.Locals != null)
      {
        foreach (Local lcl in methd.Locals)
        {
          if (lcl.Type == m_ArrayList)
          {
            this.Problems.Add(new Problem(this.GetResolution(), lcl.Name.Name));
          }
        }
      }

      return this.Problems;
    }
  }
}
```

Example: Specify Justification for SuppressMessage Attributes

When `SuppressMessageAttribute` is used in code to override an FxCop rule, the programmer may optionally specify the `Justification` property to explain the reason for using the attribute. This is a good practice which the following rule will help enforce.

The implementation requires numerous `Check` methods to be overridden because the attribute `SuppressMessageAttribute` can be placed on various kinds of targets in the source code.

Rule XML:

```
<Rule TypeName="SpecifySuppressMessageJustification" Category="Tutorial" CheckId="TT1035">
  <Name>Specify SuppressMessage justification</Name>
  <Description>
    The SuppressMessage.Justification property should be set to an explanation
    of why you are suppressing an FxCop rule.
  </Description>
  <Url></Url>
  <Resolution>Specify the Justification property.</Resolution>
  <MessageLevel Certainty="95">Warning</MessageLevel>
  <Email></Email>
  <FixCategories>NonBreaking</FixCategories>
  <Owner></Owner>
</Rule>
```

Rule Implementation:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.FxCop.Sdk;

namespace TutorialRules
{
  class SpecifySuppressMessageJustification : BaseIntrospectionRule
  {
    public SpecifySuppressMessageJustification()
      :
      base("SpecifySuppressMessageJustification", "TutorialRules.TutorialRules",
      typeof(SpecifySuppressMessageJustification).Assembly)
    {
    }

    public override ProblemCollection Check(ModuleNode module)
    {
      AssemblyNode assmbly = module as AssemblyNode;
      if (assmbly != null)
      {
        CheckSuppressMessage(assmbly.ModuleAttributes);
      }

      CheckSuppressMessage(module.Attributes);
      return this.Problems;
    }

    public override ProblemCollection Check(TypeNode type)
    {
      CheckSuppressMessage(type.Attributes);
      return this.Problems;
    }

    public override ProblemCollection Check(Member member)
    {
      Method methd = member as Method;
      if (methd != null && methd.ReturnAttributes != null)
      {
        CheckSuppressMessage(methd.ReturnAttributes);
      }

      CheckSuppressMessage(member.Attributes);
      return this.Problems;
    }

    public override ProblemCollection Check(Parameter parameter)
    {
      CheckSuppressMessage(parameter.Attributes);
```

```
      return this.Problems;
    }

    public void CheckSuppressMessage(AttributeNodeCollection attributes)
    {
      foreach (AttributeNode attr in attributes)
      {
        if (attr.Type == FrameworkTypes.SuppressMessageAttribute)
        {
          Literal lit = attr.GetNamedArgument(Identifier.For("Justification")) as Literal;
          if (lit == null || string.IsNullOrEmpty((string)lit.Value))
          {
            this.Problems.Add(new Problem(this.GetResolution()));
          }
        }
      }
    }
  }
}
```

Example: ArgumentException Parameter Names

Various constructors to `ArgumentException` have a *`paramName`* parameter. Typically, a `String` naming one of the calling method's parameters should be passed as this argument. As a special case, when the `ArgumentException` is constructed in a property, the *`paramName`* should be the property's name.

The implementation makes a call to `VisitStatements`, which recursively examines all of the expressions contained in the member, to find invocations of the constructor of `ArgumentException` or the constructor of a class that inherits from `ArgumentException`.

Rule XML:

```
<Rule TypeName="ArgumentExceptionParameterNaming" Category="Tutorial" CheckId="TT1040">
  <Name>ArgumentException constructor's paramName parameter should be a parameter name</Name>
  <Description>
    When an ArgumentException is constructed by a constructor that takes paramName,
    that paramName should be the name of a calling method's parameter, or the name of the
    property if called in a property setter.
  </Description>
  <Url></Url>
  <Resolution Name="Method">Pass the correct parameter name instead of '{0}'.</Resolution>
  <Resolution Name="Property">Pass '{0}' as the paramName argument.</Resolution>
  <MessageLevel Certainty="95">Warning</MessageLevel>
  <Email></Email>
  <FixCategories>NonBreaking</FixCategories>
  <Owner></Owner>
</Rule>
```

Rule Implementation:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.FxCop.Sdk;

namespace TutorialRules
{
  public class ArgumentExceptionParameterNaming : BaseIntrospectionRule
  {
    private Member m_CurrentMember;
    private TypeNode m_ArgumentException;

    public ArgumentExceptionParameterNaming() :
      base("ArgumentExceptionParameterNaming", "TutorialRules.TutorialRules",
```

```
  typeof(ArgumentExceptionParameterNaming).Assembly)
{
}

public override void BeforeAnalysis()
{
  m_ArgumentException = FrameworkAssemblies.Mscorlib.GetType(
    Identifier.For("System"), Identifier.For("ArgumentException"));
}

public override ProblemCollection Check(Member member)
{
  if (member is Method)
  {
    m_CurrentMember = member;
    VisitStatements(((Method)member).Body.Statements);
  }
  else if (member is PropertyNode)
  {
    PropertyNode propNode = (PropertyNode)member;
    if (propNode.Setter != null)
    {
      m_CurrentMember = member;
      VisitStatements(propNode.Setter.Body.Statements);
    }
  }

  return this.Problems;
}

public override void VisitExpression(Expression expression)
{
  Construct cnstruct;
  InstanceInitializer instInit;
  int i = 0;

  cnstruct = expression as Construct;
  if (cnstruct == null)
    return;

  if (!cnstruct.Type.IsAssignableTo(m_ArgumentException))
    return;

  instInit = (InstanceInitializer)((MemberBinding)cnstruct.Constructor).BoundMember;

  foreach (Expression operand in cnstruct.Operands)
  {
    if (instInit.Parameters[i].Name.Name == "paramName")
    {
      Literal lit;
      String litString;

      lit = operand as Literal;
      if (lit == null)
        continue;

      litString = lit.Value as String;
      if (litString == null)
        continue;

      if (m_CurrentMember is Method)
      {
        bool found = false;

        foreach (Parameter param in ((Method)m_CurrentMember).Parameters)
        {
          if (param.Name.Name == litString)
          {
            found = true;
            break;
          }
        }
```

```
            if (!found)
              this.Problems.Add(new Problem(this.GetNamedResolution("Method", litString)));
          }
          else // m_CurrentMember is PropertyNode
          {
            if (m_CurrentMember.Name.Name != litString)
              this.Problems.Add(new Problem(this.GetNamedResolution("Property", litString)));
          }
        }

        i += 1;
      }
    }
  }
}
```

Example: Check that Members are Documented

The C#, VB.NET, and C++ compilers support the generation of structured XML files that contain comments known as "documentation comments" (or "doc comments" for short) extracted from your source code. This example checks that all externally visible members (in assemblies that have a corresponding XML file) are decorated with a doc comment.

This example requires the source code presented in the section called “Checking Documentation Comments” for determining the member ID strings used by the XML documentation system.

Rule XML:

```
<Rule TypeName="DocumentExternallyVisibleMembers" Category="Tutorial" CheckId="TT1050">
  <Name>Externally visible members should have doc comments</Name>
  <Description>
    Since externally visible members comprise your published API, they should be
    documented using doc comments in the source code.
  </Description>
  <Url></Url>
  <Resolution>Document the member.</Resolution>
  <MessageLevel Certainty="95">Warning</MessageLevel>
  <Email></Email>
  <FixCategories>NonBreaking</FixCategories>
  <Owner></Owner>
</Rule>
```

Rule Implementation:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Xml;

using Microsoft.FxCop.Sdk;

namespace TutorialRules
{
  class DocumentExternallyVisibleMembers : BaseIntrospectionRule
  {
    private Dictionary<String, bool> m_ProcessedAssemblies =
      new Dictionary<String, bool>();

    private Dictionary<String, XmlElement> m_DocumentedMemberIds =
      new Dictionary<String, XmlElement>();

    public DocumentExternallyVisibleMembers() :
      base("DocumentExternallyVisibleMembers", "TutorialRules.TutorialRules",
      typeof(DocumentExternallyVisibleMembers).Assembly)
    {
```

```
}

public override TargetVisibilities TargetVisibility
{
  get
  {
    return TargetVisibilities.ExternallyVisible;
  }
}

public override ProblemCollection Check(Member member)
{
  AssemblyNode assembly = member.DeclaringType.DeclaringModule.ContainingAssembly;
  bool assemblyHasDocComments;

  // Does not apply to special members (such as get_ and set_ members),
  // with the exception of constructors.
  if (member.IsSpecialName && member.NodeType != NodeType.InstanceInitializer)
  {
    return this.Problems;
  }

  // Have we loaded the XML for this assembly yet?

  if (!m_ProcessedAssemblies.TryGetValue(assembly.Name, out assemblyHasDocComments))
  {
    // XML not loaded yet for this assembly. Determine
    // the path to the possible XML documentation file.

    string path = Path.Combine(Path.GetDirectoryName(assembly.Location),
      Path.GetFileNameWithoutExtension(assembly.Location)) + ".xml";

    // If the file exists, load the XML.

    if (File.Exists(path))
    {
      XmlDocument doc = new XmlDocument();
      doc.Load(path);

      // Put the member ID strings of the documented members
      // into the dictionary.

      foreach (XmlElement element in doc.SelectNodes("/doc/members/member"))
      {
        m_DocumentedMemberIds.Add(element.Attributes["name"].Value, element);
      }

      assemblyHasDocComments = true;
    }
    else
    {
      assemblyHasDocComments = false;
    }

    m_ProcessedAssemblies.Add(assembly.Name, assemblyHasDocComments);
  }

  // If assembly has an XML documentation file present, then its members are
  // eligible for checking to see if they have documentation.

  if (assemblyHasDocComments)
  {
    string memberId = DocComment.GetMemberId(member);

    if (!m_DocumentedMemberIds.ContainsKey(memberId))
    {
      this.Problems.Add(new Problem(this.GetResolution()));
    }
  }

  return this.Problems;
}
```

```
    }
}
```

Resources

- **Manual** http://msdn2.microsoft.com/en-us/library/bb429476.aspx
- **Blog** http://blogs.msdn.com/fxcop
- **Forum** http://forums.microsoft.com/MSDN/ShowForum.aspx?ForumID=98&SiteID=1
- **Book** *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* by Krzysztof Cwalina and Brad Abrams (ISBN 0321246756)
- **Article** “Create Custom FxCop Rules” by Dave Scarmozzino http://www.thescarms.com/dotnet/fxcop1.aspx
- **Article** “Bad Code? FxCop to the Rescue” by John Robbins http://msdn.microsoft.com/msdnmag/issues/04/06/Bugslayer/default.aspx
- **Article** “Three Vital FxCop Rules” by John Robbins http://msdn.microsoft.com/msdnmag/issues/04/09/Bugslayer/default.aspx