

FxCop: .NET Code Police

Product Overview by Craig Murphy

Coding styles and naming conventions can be an emotive subject, often sparking furious debate. If you are a solo developer, you may have the fortune to be able to choose your own coding style. If you belong to a large corporate, you may find yourself stuck with their in-house style – whilst you may not agree with it, you just have to muck in and live with it. However, with the arrival and maturation of .NET, things have changed.

Microsoft has put forward their own collection of .NET Framework Design Guidelines – a huge collection of rules and statements that help us write code that not only looks consistent, but code that is considered properly written in terms of adherence to sound object-oriented principles. To that effect, Microsoft has released FxCop, a product that analyses managed code assemblies and provides information about the assemblies, such as violations of the programming and design rules set forth in the Microsoft .NET Framework Design Guidelines.

.NET Framework Design Guidelines

Before I start looking at FxCop, it seems pertinent that Microsoft's .NET Framework Design Guidelines document (referred to as FxDG in this article, but not anywhere else!) is at least introduced. FxDG are Microsoft's guidelines for writing robust and easily maintainable code using the .NET Framework.

Amongst other things, FxDG provides guidelines covering these topics: Capitalisation Styles, Case Sensitivity, Abbreviations, Word Choice, Avoiding Type Name Confusion, Namespace Naming, Class Naming, Interface Naming, Attribute Naming, Enumeration Type Naming, Static Field Naming, Parameter Naming, Method Naming, Property Naming, and Event Naming. You can find a link to the FxDG in the Resources section of this article.

If we examine two of these topics, you will soon gain an understanding of the FxDG make-up.

Capitalisation Styles

This section describes the Pascal case, camel case, and uppercase capitalization styles to use to name identifiers in class libraries. Hopefully you are already familiar with Pascal and camel case, but if you are not, here is an extract from the FxFG:

Pascal case: The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters. For example: BackColor.

Camel case: The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example: backColor.

Uppercase: All letters in the identifier are capitalized. Use this convention only for identifiers that consist of two or fewer letters. For example: System.IO, System.Web.UI.

Interface Naming

The following rules outline the naming guidelines for interfaces:

- Name interfaces with nouns or noun phrases, or adjectives that describe behaviour. For example, the interface name IComponent uses a descriptive noun. The interface name ICustomAttributeProvider uses a noun phrase. The name IPersistable uses an adjective.
- Use Pascal case.
- Use abbreviations sparingly.
- Prefix interface names with the letter I, to indicate that the type is an interface.
- Use similar names when you define a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter I prefix on the interface name.
- Do not use the underscore character (_).

The following are examples of correctly named interfaces.

```
public interface IServiceProvider
public interface IFormatable
```

The following code example illustrates how to define the interface IComponent and its standard implementation, the class Component.

```
public interface IComponent
{
    // Implementation code goes here.
}
public class Component: IComponent
{
    // Implementation code goes here.
}
```

What is FxCop?

FxCop analyses programming elements in assemblies, called *targets*, and provides an informational report containing *messages* about the targets. FxCop represents the checks it performs during an analysis as rules. A rule is managed code that analyses a target and returns a message about its findings. Rule messages identify any relevant programming and design issues and, when possible, supply information on how to fix the target. Suggested improvements are often based on the .NET Framework Design Guidelines, which are, as we have already learnt, Microsoft's guidelines for writing robust and easily maintainable code using the .NET Framework.

For the remainder of this article I will assume that you have downloaded and installed FxCop.

Trying It Out

To find out how FxCop works, I thought I would try it out with a simple example. Using Visual Studio.NET (VS.NET), I created a regular Windows Forms application called FxCopTest. I also created a new class called "clsFxCopTest", presented in Listing 1.

```
using System;

namespace FxCopTest
{
    public class clsFxCopTest
    {
        public clsFxCopTest()
        {
        }
    }
}
```

Listing 1

For the sake of this article, I have deliberately not read the class naming guidelines, and have named this class in the same way that I would name a Delphi class (I told you coding style and naming conventions were an emotive subject!)

I then ran FxCop (via the Start, Programs, FxCop menu item). It was necessary to add my newly created .NET assembly (fxcoptest.exe) to a new FxCop project – this was a painless process made easy by the provision of an 'Add Target for Analysis' toolbar button. Asking FxCop essentially to audit the fxcoptest assembly was achieved by clicking on the Analyse button. Figure 1 presents the FxCop output.

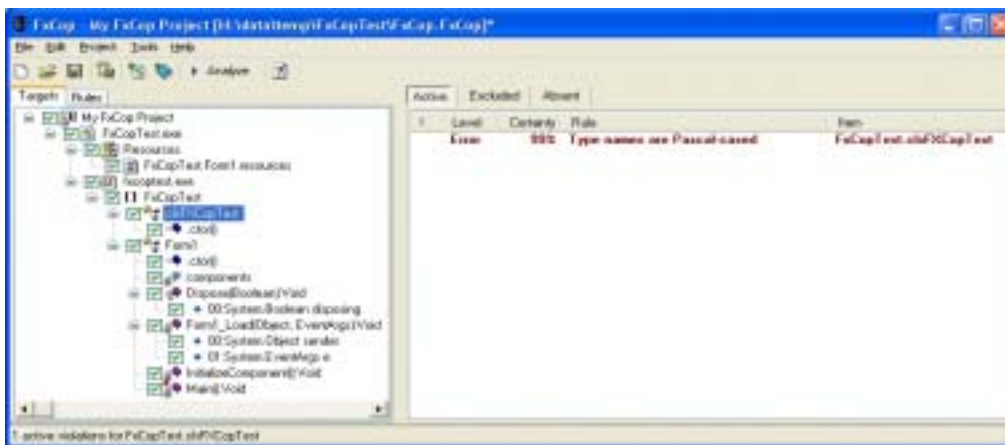


Figure 1 – FxCop output

I have deliberately selected `clsFXCopTest` in the FxCop screenshot shown in Figure 1. FxCop is 99% certain that my class name `clsFXCopTest` breaks the class naming rule. We know that `clsFXCopTest` clearly is not Pascal-cased, so why does FxCop report a certainty of 99%? Given that nothing is ever certain, rules must have a certainty of between 1 and 99 – this is a rule [sic] built-in to FxCop. Incidentally, FxCop refers to rule breaches as *violations*.

Whilst this is a simple example, there will come a time when we might not know how to avoid an FxCop violation. Thankfully FxCop allows us to double-click on a violation so that we can learn more about it. Figure 2 presents the extra information we can glean from FxCop relating to this violation.

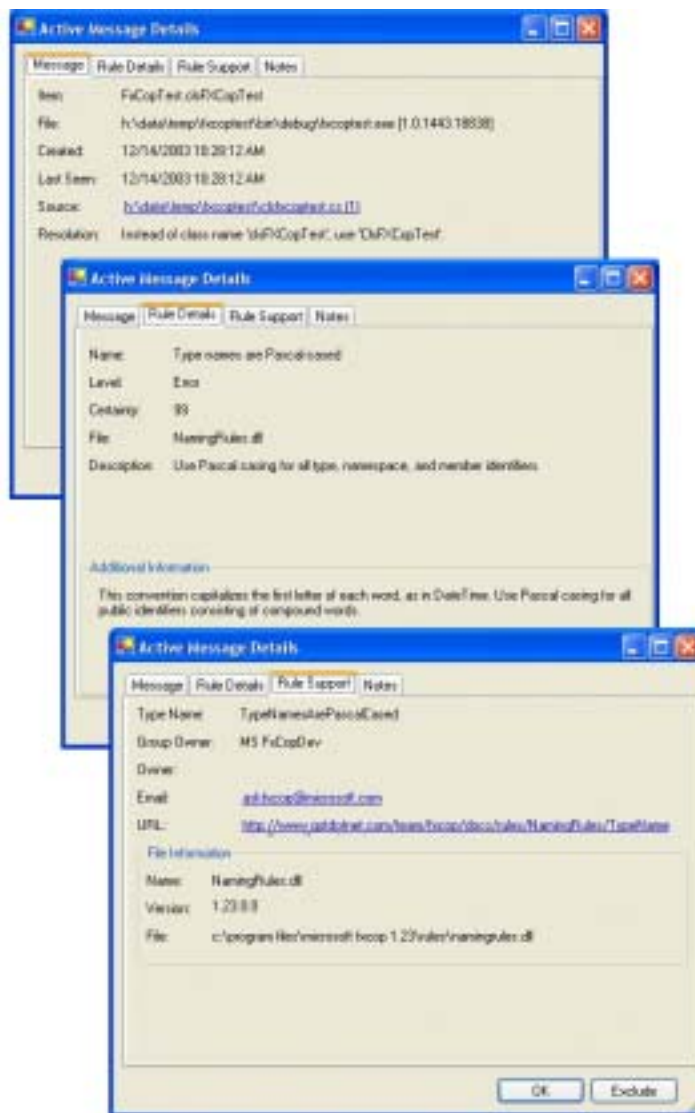


Figure 2 – FxCop analysis of our naming violation

Fixing Violations

Using the information provided by FxCop, we can correct Listing 1 to pass FxCop's inspection. Listing 2 presents the corrected code: remember that C# is case-sensitive, so we have to change the constructor's case too.

```
using System;

namespace FxCopTest
{
    public class ClsFXCopTest
    {
        public ClsFXCopTest()
        {
        }
    }
}
```

Listing 2 – Correcting the class name violation

Defining the Rules of Arrest

FxCop is supplied with a number of default rules covering these topics:

- COM – rules that detect COM Interop issues.
- Design – rules that detect potential design flaws. These coding errors typically do not impact the execution of your code.
- Globalization – rules that detect missing or incorrect usage of information related to globalization and localization.
- Naming – rules that detect incorrect casing, key-word collisions, and other issues related to the names of types, members, parameters, namespaces, and assemblies.
- Performance – rules that detect elements in your assemblies that will degrade performance.
- Security – rules that detect programming elements that leave your assemblies vulnerable to malicious users or code.
- Usage – rules that detect potential flaws in your assemblies that can impact code execution.

The rules for each of these topics are defined in a number of DLLs that are created using the FxCop SDK. FxCop rules are implemented as public types (classes). A rule targets a specific kind of programming element such as a property, method, or constructor. Every rule implements a `Check` method; the value returned by this method determines whether FxCop reports a violation. Rules are self-descriptive; that is, a rule has properties that describe its friendly name, purpose, ownership, message level, and certainty. The message level indicates to rule clients how important it is to fix violations detected by the rule.

Extending or Customising FxCop

It is possible to define our own FxCop rules using the FxCop SDK. For example, you may have a corporate policy that promotes improving the design of your existing code (we can but wish!), also known as refactoring. We could create FxCop rules that help us automatically spot areas that need to be re-designed/refactored.

Our own custom rules must implement one of these interfaces: `IAssemblyRule`, `IConstructorRule`, `IEventRule`, `IFieldRule`, `IMemberRule`, `IMethodBaseRule`, `IModuleRule`, `INamespaceRule`, `IParameterRule`, `IPropertyRule`, `IResourceStreamRule` or `ITypeRule`. These interfaces define a `Check()` method that we must implement in our custom rule. The FxCop SDK documentation provides an explanation of *when* each of these interfaces should be used – I shall refrain from repeating that information in this article. We will, however, take a look at how the `IParameterRule` interface can be implemented.

The FxCop SDK documentation is supplied with a handful of sample rules – one of which we could use to highlight our refactoring need. Here is the scenario:

Imagine you produce commercial software libraries and you have discovered that your clients do not like having delegates as method parameters. The purpose of this custom rule is to report violations when FxCop detects delegate parameters in public or protected methods in public or protected non-nested and nested types. This rule will be a critical error rule, because you do not want to ship a library that has this defect. This is best implemented as a parameter rule, so you will implement the `IParameterRule` interface.

The `IParameterRule` interface's `Check()` method has the following signature:

```
object Check(Module parentModule, ParameterInfo parameter)
```

The `IParameterRule` interface is implemented by rules that check for defects in parameters. For example, a rule that checks whether parameter names start with a lowercase letter would implement this interface. The 'parentModule' and 'parameter' parameters reflect the module and parameter to be analysed by the rule. Inside the `Check()` method we can use reflection to examine the make up of 'parameter'. Listing 3 presents one of the FxCop C# sample rules that defines a rule that raises a violation if a parameter is a delegate.

```
using System;
using System.Globalization;
using System.Reflection;
using System.Xml;
using Microsoft.Tools.FxCop.Sdk;

namespace Microsoft.Tools.FxCop.Sdk.Docs.Samples
{
    public class ParametersAreNotDelegates : IParameterRule
    {
```

```

public object Check(Module parentModule,
                    ParameterInfo parameter)
{
    Type parameterType = parameter.ParameterType;

    // Check whether the parameter is a delegate type.
    if(parameterType.IsSubclassOf(typeof(Delegate)))
    {
        // Return the name of the parameter.
        return parameterType.Name;
    }
    else
    {
        // No violation occurred.
        return null;
    }
}

// Set the IReflectionRule property values.

public ProtectionLevels TypeProtectionLevel
{
    get{return ProtectionLevels.Public |
            ProtectionLevels.Family;}
}

public ProtectionLevels NestedTypeProtectionLevel
{
    get{return ProtectionLevels.Public |
            ProtectionLevels.Family |
            ProtectionLevels.Assembly;}
}

public ProtectionLevels MemberProtectionLevel
{
    get{return ProtectionLevels.Public |
            ProtectionLevels.Family;}
}

// Set the IRule property values.

public string Name
{
    get{return "Do not use delegate parameters (SDK sample 1)";}
}

public string Description
{
    get{return "Company policy states that parameters should " +
            "not be delegates.";}
}

public string LongDescription
{
    get{return "Methods that require delegates should " +
            "be redesigned. Consider using the " +
            ".NET event model.";}
}

public MessageLevel MessageLevel
{
    get{return MessageLevel.CriticalError;}
}

public int    CertaintyHigh {get{return 99;}}
public int    CertaintyLow  {get{return 99;}}
public string GroupOwner   {get{return "FxCop Custom Rules Team";}}
public string DevOwner     {get{return "Jane Developer";}}
public string Url          {get{return "http://www.example.com";}}
public string Email        {get{return "JaneD@example.com";}}

// Implement the IRule Methods.

public string DisplayResolution(object value)
{

```

```

        return String.Format(
            CultureInfo.CurrentCulture,
            "'{0}' cannot be used as a parameter in an " +
            "externally visible method.",
            value.ToString());
    }

    public object LoadResolution(XmlNode value)
    {
        return value.InnerText;
    }

    public void SaveResolution(XmlWriter writer, object value)
    {
        writer.WriteString(value.ToString());
    }

    public void BeforeAnalysis() {}
    public void AfterAnalysis() {}
}
}

```

Listing 3 – Implementing your own rules

Loosening the Handcuffs...

You may have noticed the checkbox treeview control on the left-hand side of the FxCop window: this allows us to instruct FxCop to ignore particular parts of an assembly. FxCop saves its own project settings in a separate file, any changes you make to this treeview are still in place the next time you load the FxCop project.

The rules tab, as shown in Figure 3, highlights the Usage rules that FxCop implements out of the box. Again, notice, the checkbox treeview control – we can uncheck any one of these rules that we deem inappropriate.

Being able to enable or disable certain rules may prove very useful in your organisation – after all, you may not want to implement all of Microsoft’s recommendations. Equally, you may not want to have the code police (FxCop) examine incomplete assemblies. These checkboxes give us the ability to loosen the handcuffs or weaken FxCop’s grip thus providing us with a little breathing space.



Figure 3 – Rules can be enabled and disabled

Earlier in this article I demonstrated that FxCop provides explanations for all of its rules. I used a fairly trivial example, a case violation. Before I wrap this article up, I thought it would be useful to look at a more complicated violation.

If we look at Figure 3 a little more closely, we can see that one of FxCop’s Usage rules is: “Constructors should not call virtual methods defined by the class”. On the surface, this might not look like a problem, and if you do not have any derived classes there is no problem. However, as FxCop explains, if you do have derived classes, then you do have a problem. Figure 4 presents FxCop’s explanation of this particular rule. I have repeated the explanation here:

“Virtual methods defined on the class should not be called from constructors. If a derived class has overridden the method, the derived class version will be called (before the derived class constructor is called).”



Figure 4 – FxCop provides explanations for all rules

Summary

This article has provided a fly-by look at FxCop version 1.23. I have deliberately not gone into very much detail about the precise operation of FxCop – if you are interested in writing quality code that adheres to Microsoft’s guidelines, it is worth downloading FxCop and finding out for yourself.

Whilst I did not mention it in this article, the default assembly created by VS.NET for a simple empty Windows Forms application contained four violations. As a test, I built the same empty application using C#Builder. Not surprisingly, the C# assembly’s violations were identical to the VS.NET equivalent. On the subject of .NET compilers/languages, FxCop works with Visual Basic.NET too.

There is an MSDN TV MPEG video available from msdn.microsoft.com – it is about 22mb in size and runs for 12 minutes. The video does not actually demonstrate FxCop, but it does provide an overview of what FxCop can do. Interestingly, in this age of modern technology, the video includes a written example (using a whiteboard) of a few violations that FxCop can spot. You can find a link to the video in the Resources section of this article.

Finally, I discovered FxCop via careful use of RssReader (www.rssreader.org) and the MSDN RSS newsfeed. Instead of me having to visit the MSDN web-site, the MSDN web-site comes to me – a great time-saver. Be careful however, RSS can be a time-killer too. If you are interested in learning more about RSS, please feel free to download and read my article “Implementing RSS using PHP” (published at the same time as this article).

Resources

- FxCop download: <http://gotdotnet.com/team/fxcop/>
- Michael Murray and Jeffrey Van Gogh from the CLR Team describe FxCop, the freely-available code analysis tool from Microsoft, MSDN TV: <http://msdn.microsoft.com/msdntv/episode.aspx?xml=episodes/en/20031204FxCopMM/manifest.xml>
- Microsoft’s .NET Framework Design Guidelines (all one line): <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp>

Craig is an author, developer, speaker and Dilbert Evangelist – he specialises in all things XML, particularly SOAP and XSLT. Craig is evangelical about .NET, C#, Test-Driven Development and Extreme Programming. He can be reached via e-mail at: Craig@isleofjura.demon.co.uk, or via his web site: <http://www.craigmurphy.com> (where you can also find the source code and PowerPoint files for all of Craig’s Developers Group articles, reviews and presentations).