StyleCop is a source code analysis tool for C# that checks for compliance with a set of best practice rules covering code style, layout, design, and documentation.

The goal is to define guidelines to enforce consistent style and formatting and help developers avoid common pitfalls and mistakes.

Over time, StyleCop evolved to include new rules that go beyond style checks. StyleCop has become a good complement to FxCop (Code Analysis). FxCop analyzes compiled .NET binaries, while StyleCop analyzes the original source code. This allows StyleCop to investigate issues in code that are thrown away by the compiler. StyleCop could also be used to investigate issues in non-compiled languages such as JavaScript or Xml.

StyleCop is an open source static code analysis tool from Microsoft that checks C# code for conformance to StyleCop's recommended coding styles and a subset of Microsoft's .NET Framework Design Guidelines. StyleCop analyzes the source code, allowing it to enforce a different set of rules from FxCop (which, instead of source code, checks .NET managed code assemblies). The rules are classified into the following categories:

Documentation

Layout

SA1505: OpeningCurlyBracketsMustNotBeFollowedByBlankLine

SA1507: CodeMustNotContainMultipleBlankLinesInARow

Maintainability

Naming

Ordering:

SA1200: UsingDirectivesMustBePlacedWithinNamespace

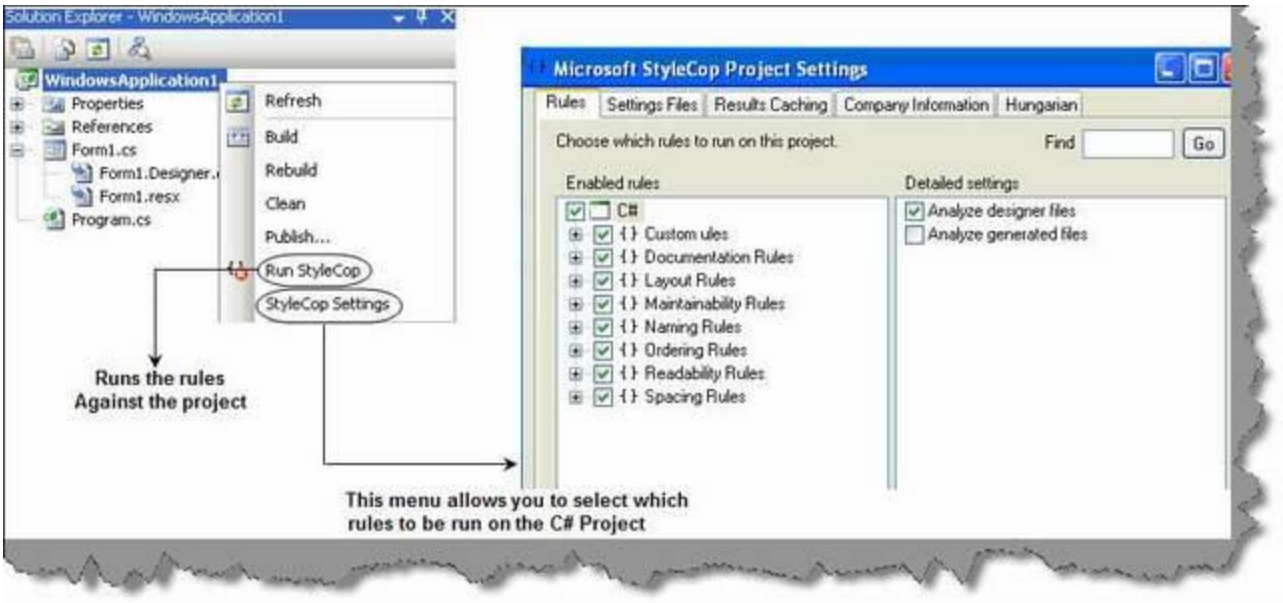
SA1203: ConstantsMustAppearBeforeFields

SA1208: SystemUsingDirectivesMustBePlacedBeforeOtherUsingDirectives

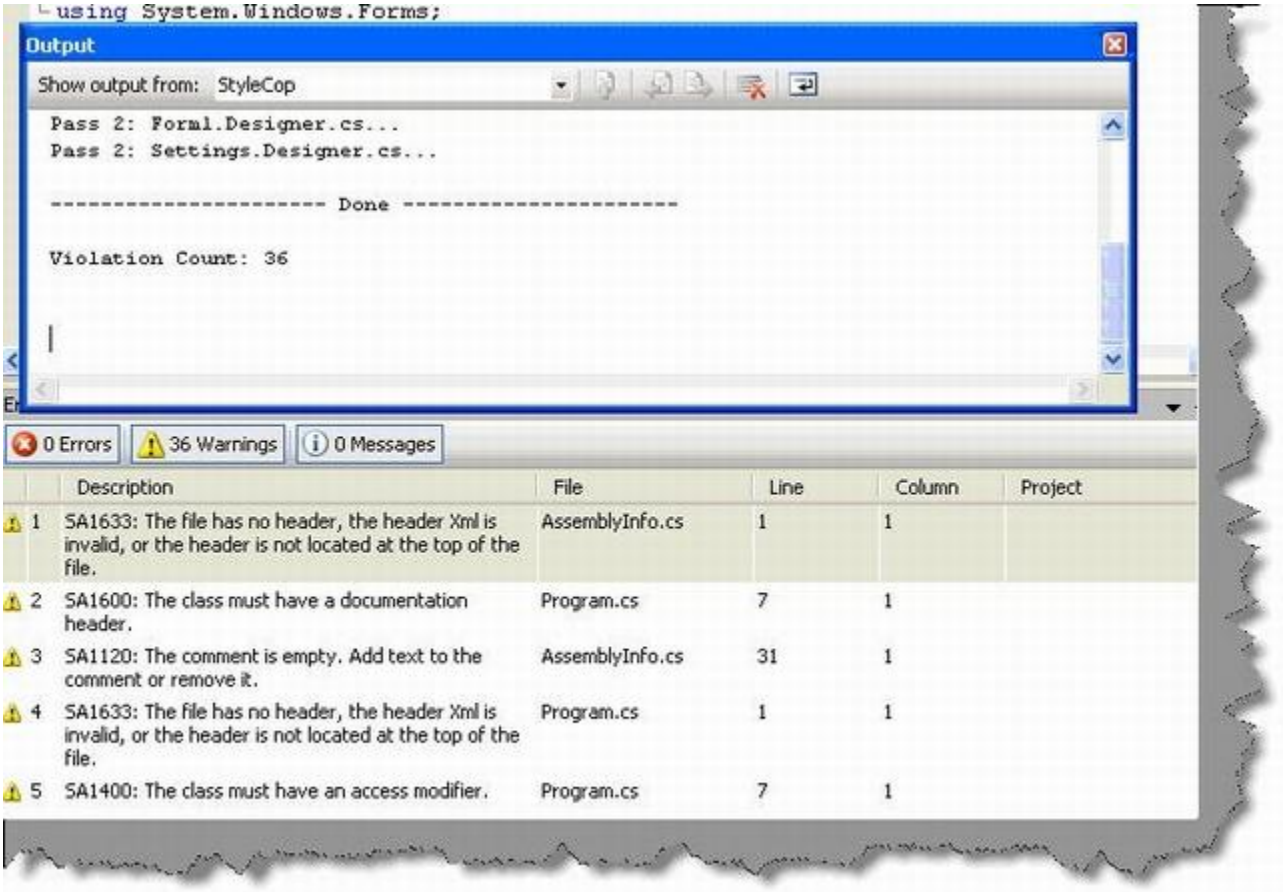
Readability

Spacing

Once you install StyleCop, nothing will be seen as in start menu. When you open any project of C# and right click on the project, you should see two menus in the project as shown in the figure below - one which helps us select the rules, i.e. the Stylecop settings menu and the other which runs these rules on the C# project.



If you just run the stylecop, you should see broken rules in output window as shown in the figure below:



Comparing FXCOP with StyleCop

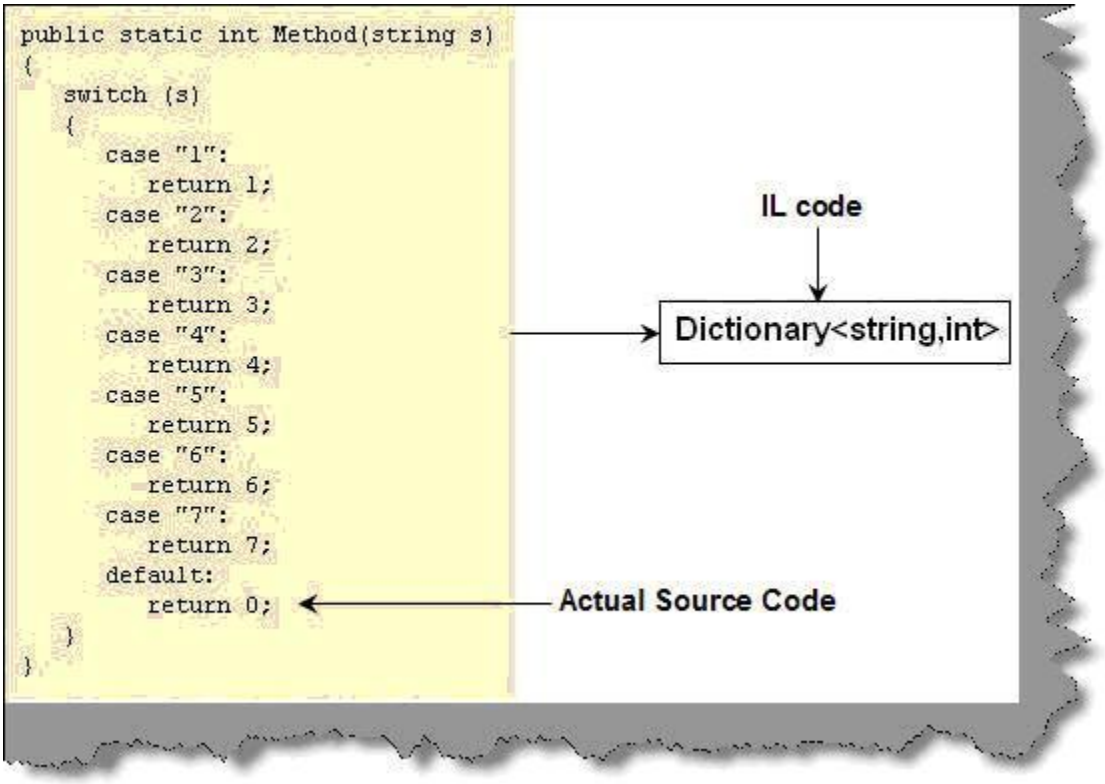
FXCOP	StyleCop
It runs on compiled DLLs.	It runs on actual source code.
As it runs on compiled IL code it can be used for C#, VB.NET, in short any language which compiles to IL code.	Currently it runs only on C#.

Currently, I can only think about two differences. If any one knows more, let me know so that I can update this. Many people are asking for a comparison table for these two codes review tools.

Issue of Code Review using FXCOP

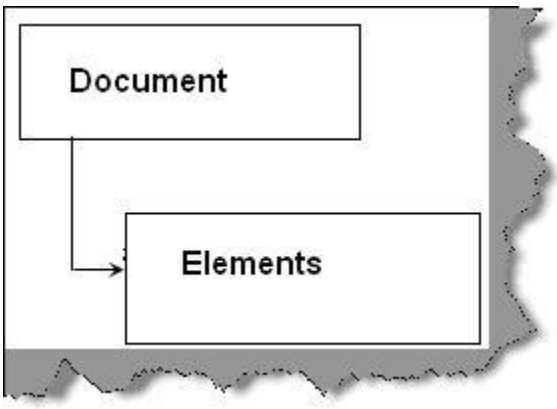
First, let me thank Thomas Wellar for giving me accurate information about FXCOP issues with code review.

FXCOP works on the actual compiled DLL. For instance, a switch method shown below can be compiled to dictionary of `string` by the .NET compiler. Due to this, we cannot actually measure what kind of quality the code has. The conversion from actual source code to IL is very much non-linear. This is where StyleCop has an extra edge as it operates directly on the source code. The only negative point it has currently is that it only works for C#.



StyleCop Code Parsing Logic

StyleCop has two important concepts - one is the document and the other the elements. Document is the file and elements are everything inside the file.



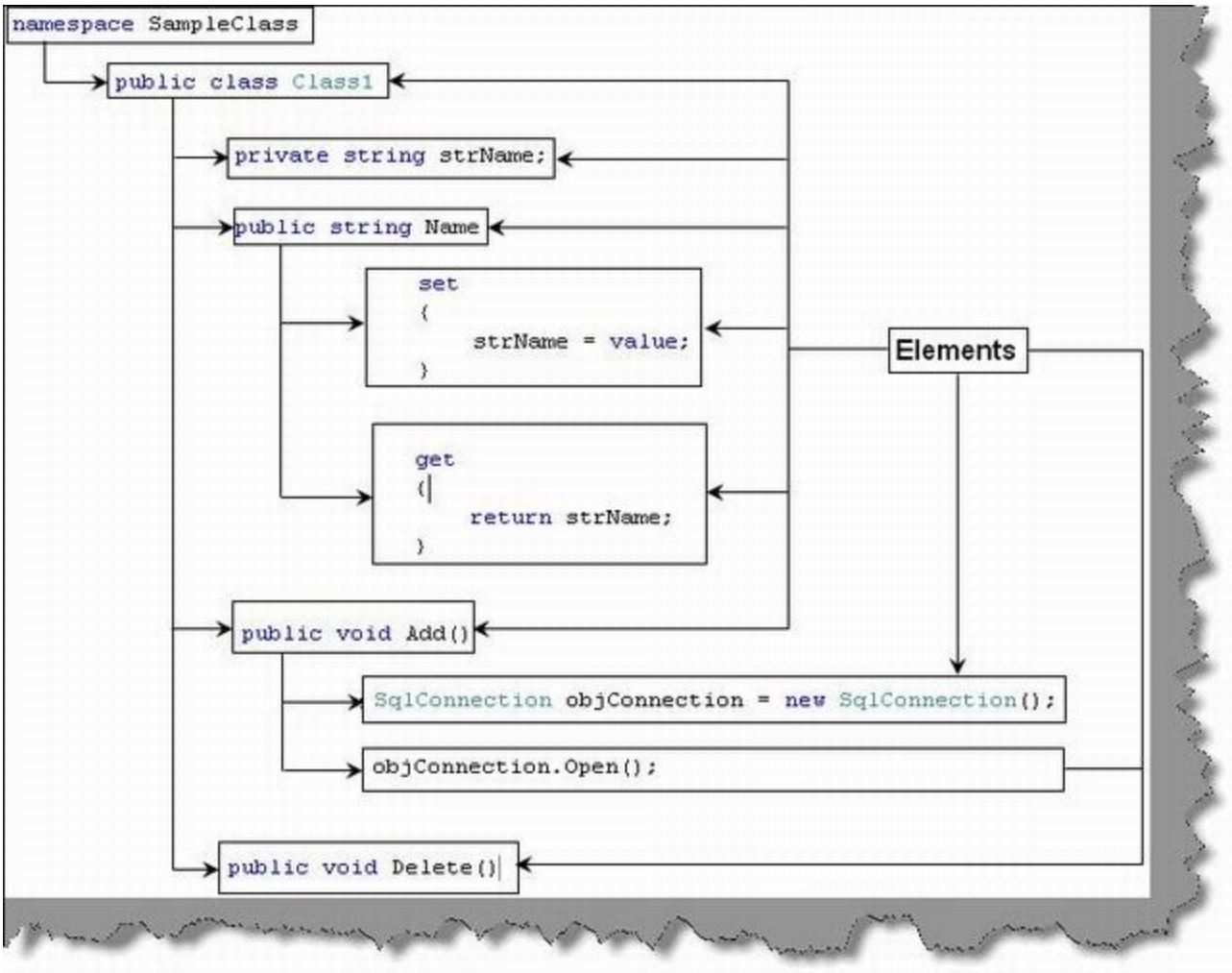
For instance, below is a simple source code which has two methods, properties and variables.

```
namespace SampleClass
{
    public class Class1
    {
        private string strName;

        public string Name...
        public void Add()
        {
            SqlConnection objConnection = new SqlConnection();
            objConnection.Open();
        }
        public void Delete()...

    }
}
```

So stylecop treats everything inside the document as an element. Now let's understand how stylecop browses through the document by implementing a custom rule in stylecop.



Making the Custom Rules Class

The first thing is to import stylecop namespaces. Please ensure that you give references from *c:\Program files\Microsoft Style Cop* folder.

Hide Copy Code

```
using Microsoft.StyleCop.CSharp;
using Microsoft.StyleCop;
```

All stylecop custom rule classes should inherit from `SourceAnalyzer` class:

Hide Copy Code

```
[SourceAnalyzer(typeof(CsParser))]
public class MyRules : SourceAnalyzer
```

When the style cop starts running, it first hits the `analyzedocument` method. So override the method and let's put our logic in this:

Hide Copy Code

```
public override void AnalyzeDocument(CodeDocument document)
{}
```

In the analyze document, we type cast the document into `CsDocument` class. As we had previously said, stylecop walks through the document and then walks through each element. This it achieves by using delegates and visitor pattern. The `WalkDocument` method visits the `VisitElement` for every element in the source code. In case you are curious about visitor pattern, you can always read the same from this article.

Hide Copy Code

```
CsDocument document2 = (CsDocument)document;
if (document2.RootElement != null && !document2.RootElement.Generated)
{
document2.WalkDocument(new CodeWalkerElementVisitor<object>
                                    (this.VisitElement), null, null);
}
```

The `VisitElement` defined in the `WalkDocument` method is a `delegate`. So we need to also define that`delegate`. You can see in the below code how we have defined the `delegate` and used the `element` object to check whether it's a `method` or not. If the `element` is a `method`, then you can perform some logic. If there is any violation, we need to add the same to the violation collection using `AddViolation` method as shown in the below snippet:

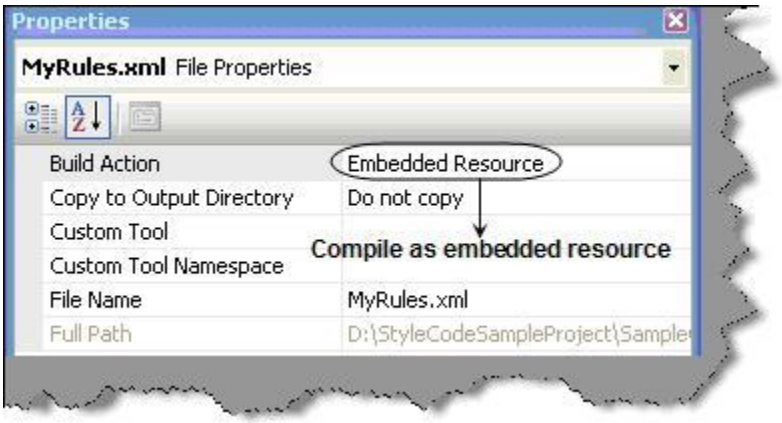
Hide Copy Code

```
private bool VisitElement(CsElement element, CsElement parentElement, object context)
{
if (element.ElementType == ElementType.Method)
{ this.AddViolation(parentElement, "MyCustomRule", "BlockStatementsShouldNotBeEmpty");
}
}
```

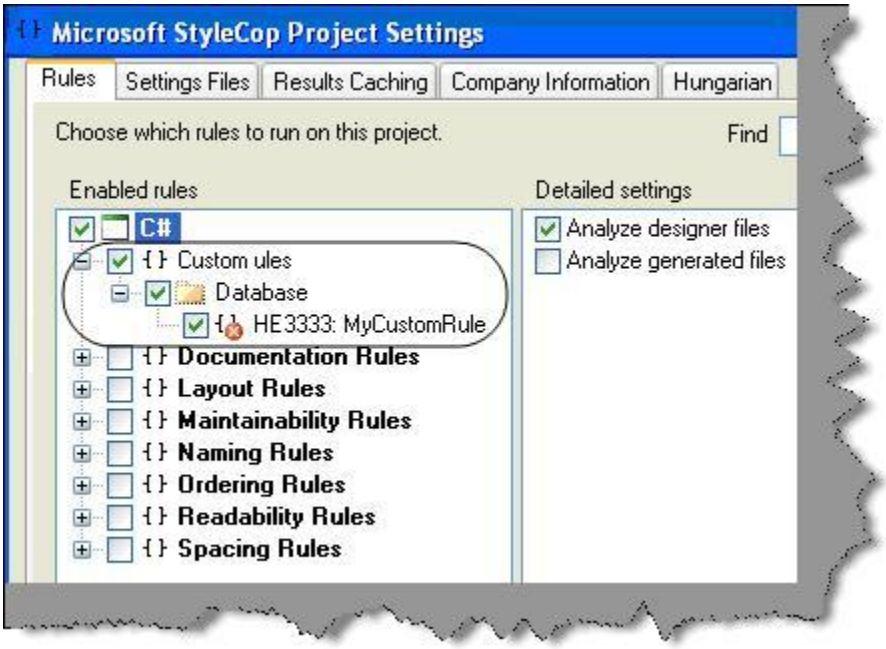
One of the values we pass to the rule is `MyCustomRule` value. This is nothing but the XML file. Below is a sample of a GOTO sample rule. When you compile the DLL, you need to ensure that this XML file is compiled as an embedded resource.

Hide Copy Code

```
<SourceAnalyzer Name="Custom Rules">
<Description> Custom rules added to analyzer. </Description>
<Rules>
<RuleGroup Name="Database">
<Rule Name="MyCustomRule" CheckId="HE3333">
<Context>Always close the connection object</Context>
<Description>Always close the connection object</Description>
</Rule>
</RuleGroup>
</Rules>
</SourceAnalyzer>
```



Once you have compiled the DLL, you need to put the DLL in the "*C:\Program Files\Microsoft StyleCop 4.3*" folder. If everything is fine, you should see the rules in the rules dialog box as shown in the figure below:



Let Us Do Something Practically

So let's take up something practically. Let's say we want to check if any connection object is opened, it should be closed. So let's write some good logic in the `VisitElement delegate`.

The first thing we will do is define two variables - `boolFoundConnectionOpen` which states that the connection object is opened and `boolFoundConnectionClosed` which indicates that the connection object is closed.

Hide Copy Code

```
bool boolFoundConnectionOpen=false;
```

```
bool booFoundConnectionClosed=false;
```

In the `VisitElement delegate`, we type cast the element document into `CsDocument` type and read the source into a `TextReader`.

Hide Copy Code

```
CsDocument document3 = (CsDocument)element.Document;
System.IO.TextReader objReader = document3.SourceCode.Read();
```

Ok, now we look through the `reader` object and start reading the code. If we find a "`.Open`", we set the`boolFoundConnectionOpen` to `true`. If we find a corresponding `close`, we set`boolFoundConnectionClosed` to `true`.

Hide Copy Code

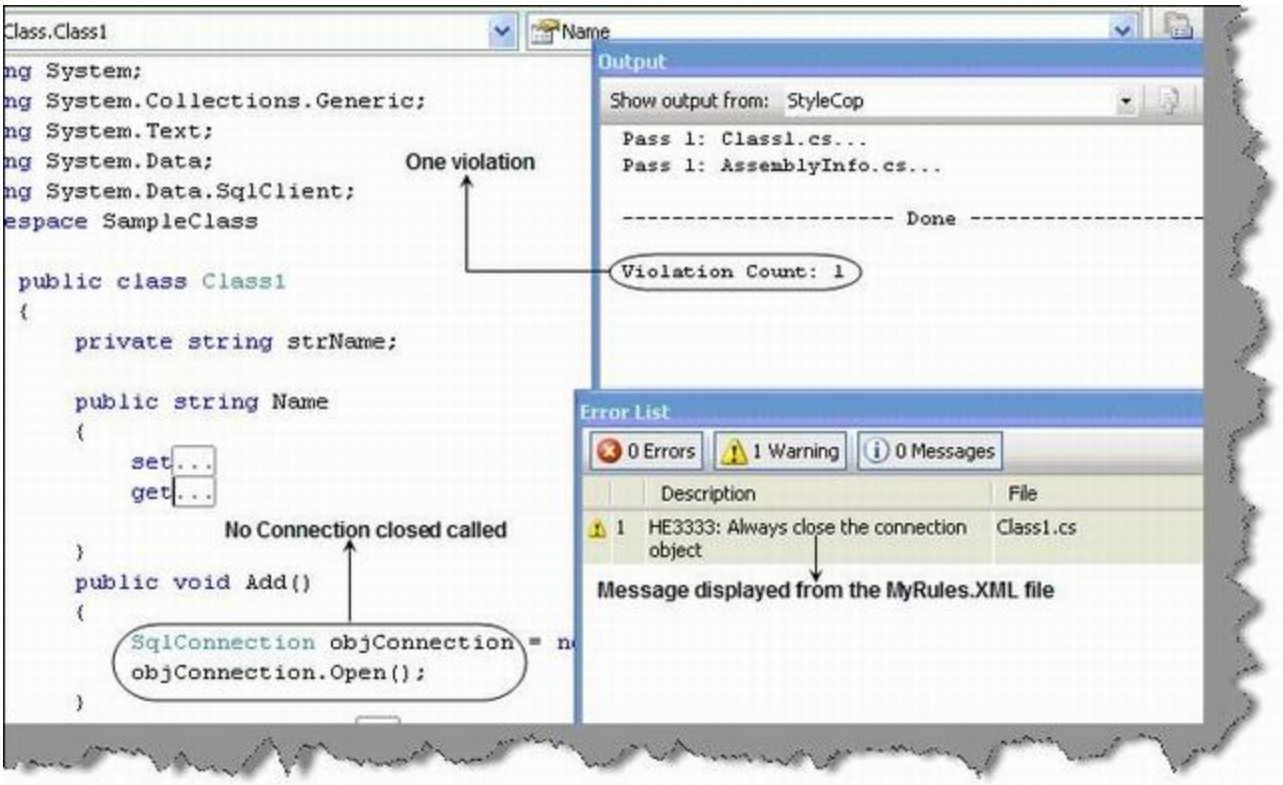
```
while ((strCode = objReader.ReadLine()) != null)
{
if (strCode.Contains(".Open();"))
{
boolFoundConnectionOpen = true;
}
if(boolFoundConnectionOpen)
{
if (strCode.Contains(".Close();"))
{
booFoundConnectionClosed = true;
}
}
}
```

Now the final thing. If we find that the connection is open and not closed, we just add to the violation collection the rule.

Hide Copy Code

```
if ((boolFoundConnectionOpen) && (booFoundConnectionClosed==false))
{
this.AddViolation(parentElement, "MyCustomRule", "BlockStatementsShouldNotBeEmpty");
}
```

Run and enjoy... You can see in the below figure that we do not have a connection closed. You can see how the error is displayed in the output window.


Class.Class1
Name
ng System;
ng System.Collections.Generic;
ng System.Text;
ng System.Data;
ng System.Data.SqlClient;
espace SampleClass
public class Class1
{
private string strName;
public string Name
{
set...
get...
}
public void Add()
{
SqlConnection objConnection = n
objConnection.Open();
}
One violation
No Connection closed called
Output
Show output from: StyleCop
Pass 1: Class1.cs...
Pass 1: AssemblyInfo.cs...
---------- Done ----------
Violation Count: 1
Error List
0 Errors
1 Warning
0 Messages
Description
File
1 HE3333: Always close the connection object
Class1.cs
Message displayed from the MyRules.XML file