

MicroCODE Software Engineering Services

Project Coding Standards

MCX-I01 (Internal C# Coding Standards).docx

Project: MicroCODE Control, Server, Remote

Development Environment: Microsoft Visual Studio 2017

OS: Microsoft Windows 10 Embedded (IoT, LTSC)

Platform: Microsoft .NET ??, Universal Windows Platform (UWP), XAML

Language: Microsoft C# ??

Tools: Microsoft StyleCop v6.0.6902.2, FxCop v??

Note: Overview of coding practices on this project

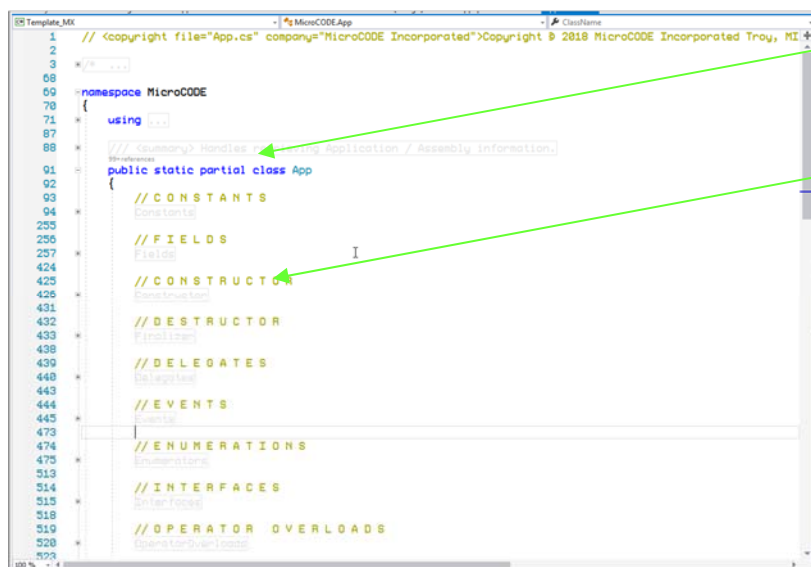
- 1) Start all new classes from “**ClassTemplate.cs**” and complete all documentation.
- 2) Following the MSDN Naming Guidelines in the “**ClassTemplate (MSDN Guidelines).cs**” copy.
- 3) Download Microsoft’s **StyleCop** and **FxCop** both are free.
- 4) As you are creating classes run StyleCop from within Visual Studio (right click in code and select ‘Run StyleCop’)
- 5) Correct any code formatting or naming issues that arise as you go.
- 6) Don’t wait until the module is done or you could end up with hundreds of errors to correct.

Note: MicroCODE **ClassTemplate.cs**

This class template was created after moving existing GEP C# code into SEP and using StyleCop and Regions to ensure all code will pass rigorous code reviews. The naming standards used are based on Microsoft’s current naming standards from the MSDN Developer’s site.

There are two versions of the template, one to actually copy and start classes from, and the other as references with all Microsoft standards included with any MicroCODE additions or changes.

Example: MicroCODE **ClassTemplate.cs** – viewed from within Visual Studio



The template includes all required XML documentation tags for the base code.

Regions are used to collapse all code and data sections in the class into the order specified by StyleCop.

These regions allow quick navigation to the major class components.

Do **not** remove empty Regions from the template when it is used to create a Class, leave them in place for future use and reference. e.g.: If I open a class that I have never seen before and I open the “Delegates” region and find it empty, then I know it has none.



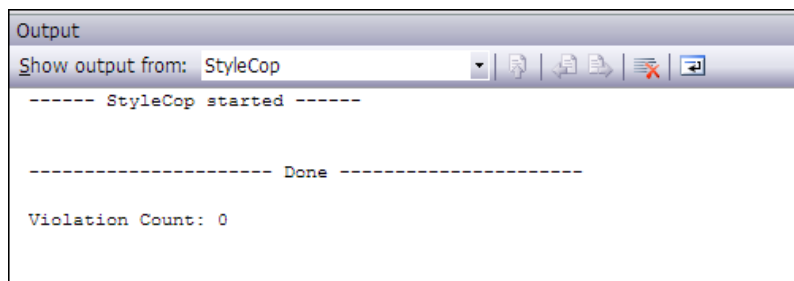
Note: MicroCODE **ClassTemplate.cs** – passes StyleCop with no warnings or errors when using the MicroCODE StyleCop Rules Set, described later in this document.



```
// <copyright file="ClassTemplate.cs" company="MicroCODE Incorporated">
namespace MicroCODE.Subsystem
{
    using ...

    /// <summary> ...
    public class ClassTemplate
    {
        // C O N S T A N T S
        Constants

        // F I E L D S
        Fields
    }
}
```

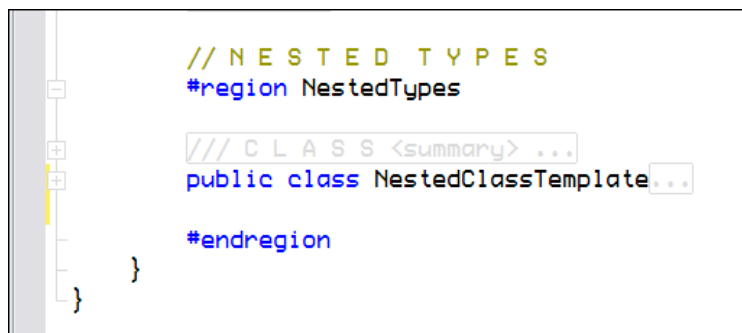


```
Output
Show output from: StyleCop
----- StyleCop started -----

----- Done -----

Violation Count: 0
```

Task: Copy template and rename to your Class name, one class per file, nested classes can be included if they are never used independently of the containing class, see “NestedTypes” region, and all nested class go here.

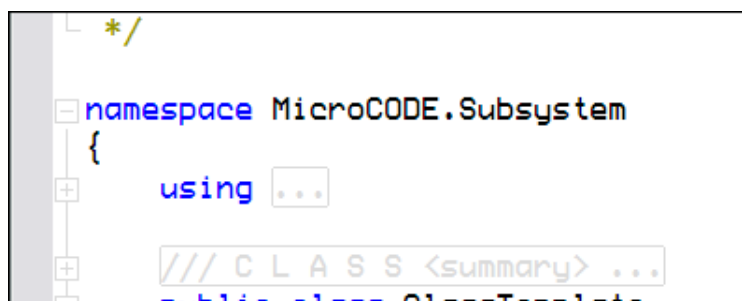


```
// N E S T E D   T Y P E S
#region NestedTypes

    /// <summary> ...
    public class NestedClassTemplate...

#endregion
}
```

Task: Substitute “ClassTemplate” to “YourClassName” whatever your class name is...



```
*/

namespace MicroCODE.Subsystem
{
    using ...

    /// <summary> ...
    public class ClassTemplate
```



Task: Fill in the documentation header with all required information, if you have questions refer to an existing MicroCODE Class for reference.

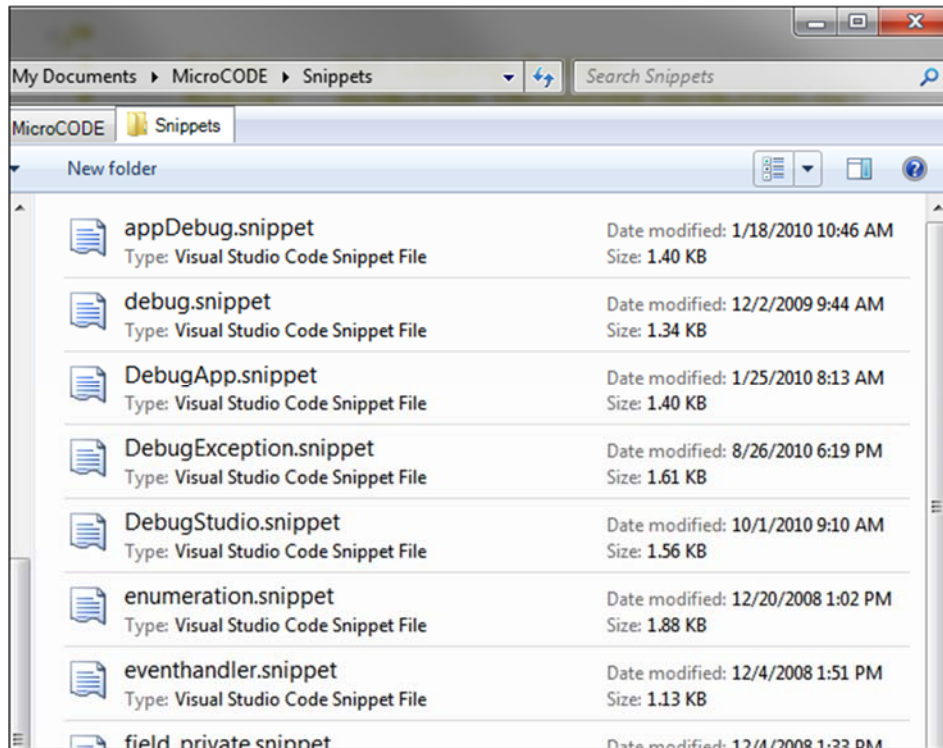
```
ClassTemplate.cs x
1 // <copyright file="ClassTemplate.cs" company="MicroCODE, Inc.">Copyright © 2008 Micro
2
3 /*
4  *      Title:      <Class Description>
5  *      Module:     ClassTemplate (<Namespace>:ClassTemplate.cs)
6  *      Project:    <Project Name>
7  *      Customer:   <Customer Name>
8  *      Facility:   MicroCODE Incorporated
9  *      Date:       <Class Creation Date>
10 *      Author:      <Class Author's Name>
11 *
12 *      Copyright © 2008 MicroCODE Incorporated Troy, MI
13 *
14 *      This software and related materials are the property of
15 *      MicroCODE Incorporated and contain confidential and proprietary
16 *      information. This software and related materials shall not be
17 *      duplicated, disclosed to others, or used in any way without written
18 *      permission from MicroCODE Incorporated.
19 *
20 *
21 *      DESCRIPTION:
```

Task: Document any sources used in the creation of the class

```
*
*
*      REFERENCES:
*      -----
*
*      1. "Microsoft MSDN Library CD-ROMs"
*          from MSDN for Visual Studio 2005.
*
*      2. "Build a Program Now! - Microsoft Visual C# 2005 - Express Edition"
*          by Patrice Pelland, 2005 Edition.
*
*      3. "Microsoft Visual C# 2005 - Step by Step"
*          by John Sharp, 2005 Edition.
*
*      4. "Windows Forms 2.0 - Programming"
```

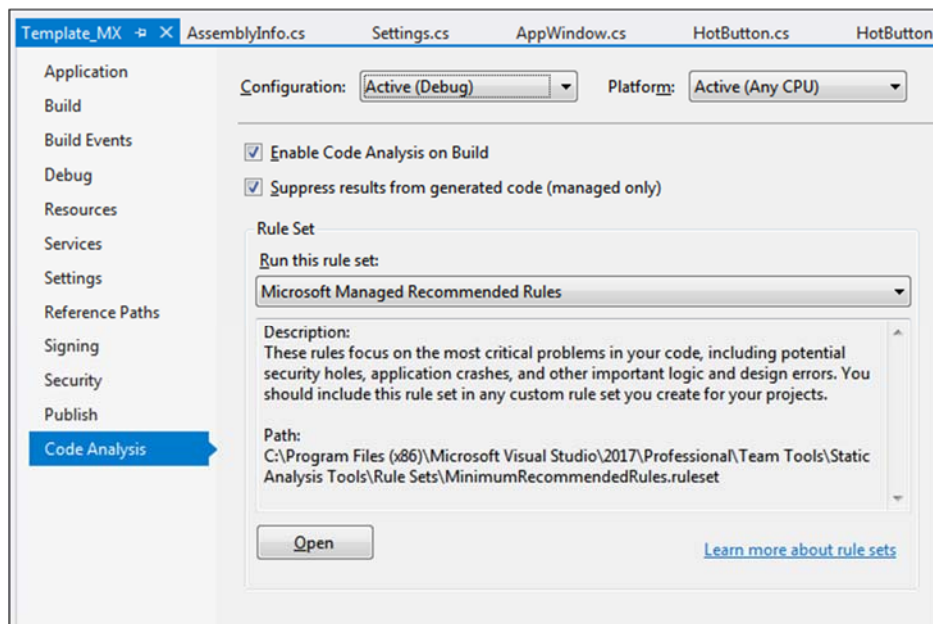


Task: Code new Class, use **MicroCODE Snippets** whenever possible, and follow all MicroCODE and MSDN Guidelines...



Task: Run **StyleCop** frequently to stay on top of coding violations as-you-go

- Change Visual Studio environment to always run StyleCop during a Build



Code Editor Settings recommended for MicroCODE Projects

Note: MicroCODE recommended **Editor Settings for Visual Studio**

With our standard settings, notice the appearance of the XML documentation tags in the VS Editor, see how they basically disappear so you can concentrate—and see clearly—the real code...

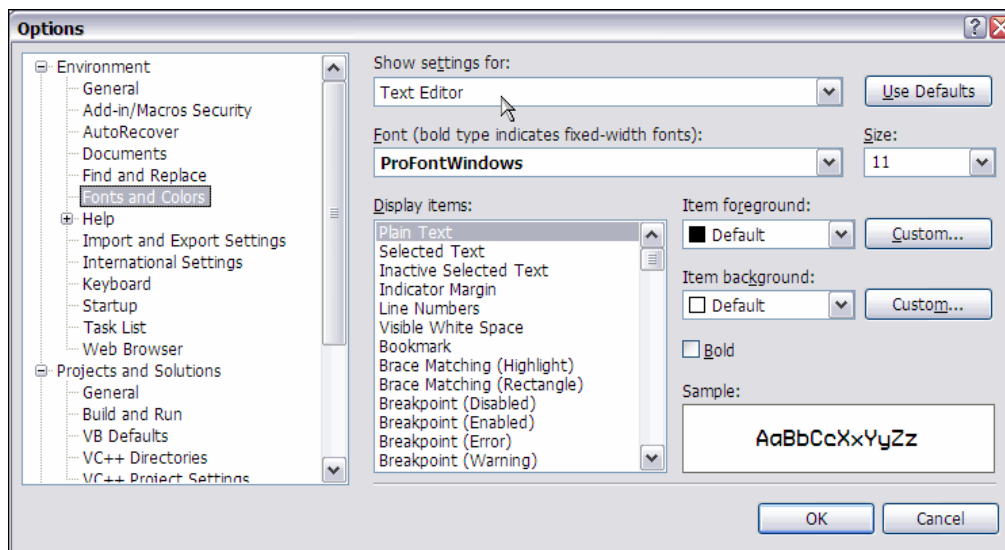
```
// C O N S T R U C T O R
#region Constructor

/// <summary>
/// Initializes a new instance of the BarcodeScannerDialogBox class.
/// </summary>
/// <param name="cfg">The caller's configuration parameters for the BarcodeScanner.</param>
/// <param name="ioxIndex">The screen/storage index for this object.</param>
public BarcodeScannerDialogBox(BarcodeScanner.Parameters cfg, int ioxIndex)
{
    // Hold local copy to edit and return to caller via 'Parameters' method.
    _scnParams = cfg;
}
```

...this is a by-product of VS Editor settings.

I recommend you use these, or something similar:

Go to **Tools>Options... Environment>Fonts and Colors**



Set your Font for code to **ProFontWindows** or another monospace font with slashed zeros and other features designed specifically for coding. It is very easy on the eyes and allows for text-based tables that line-up columns. This feature is important for List Processing data in our common code.

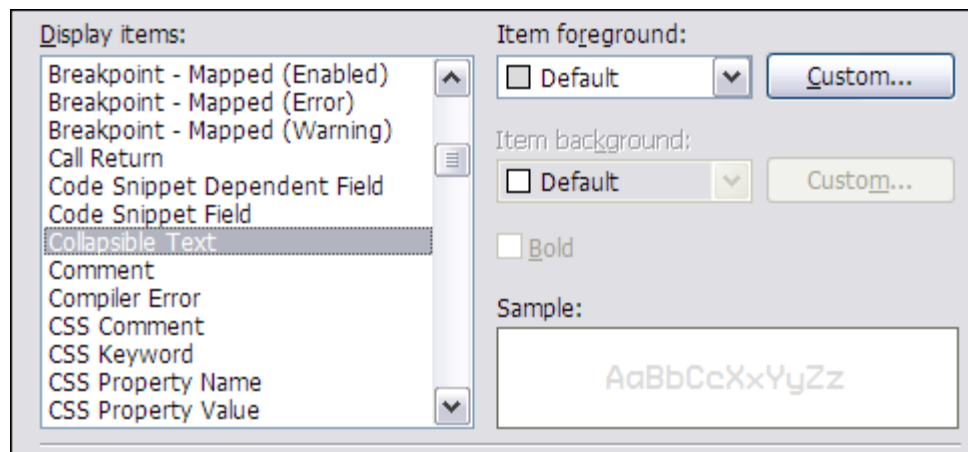
If you do not have Pro Font it's free and embedded here, I highly recommend this for code, and you won't go back.



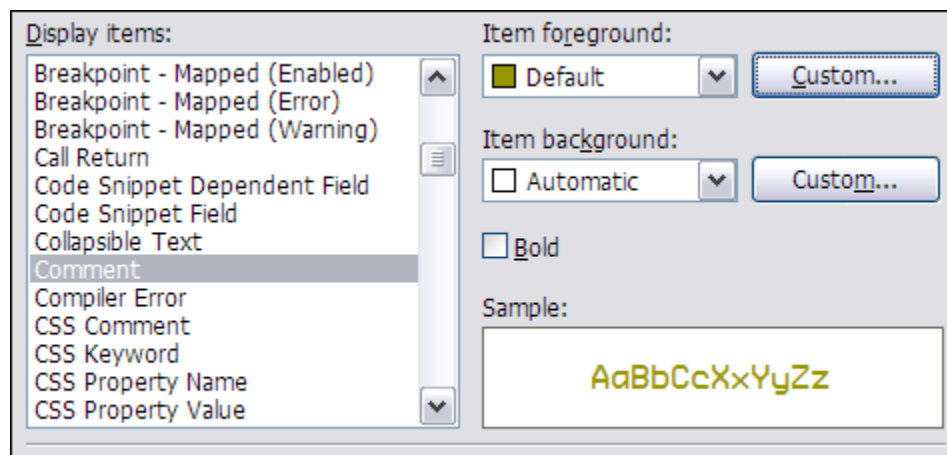
ProFontWindows.zip



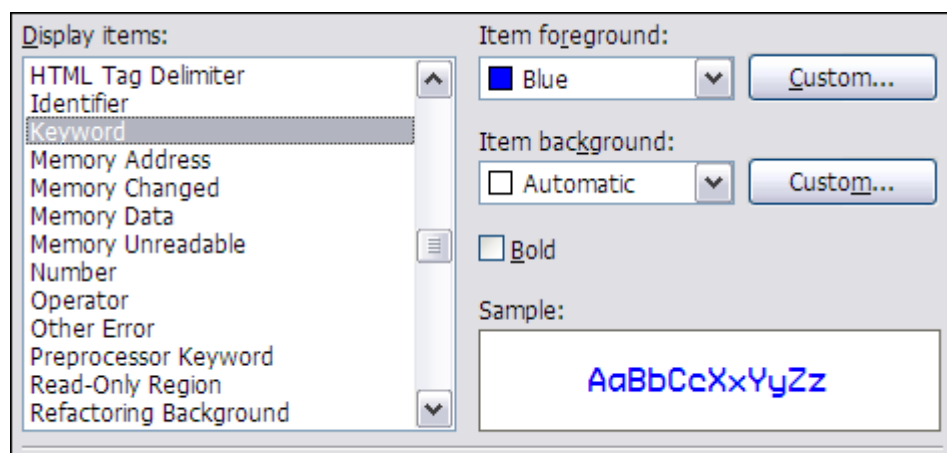
Set '**Collapsible Text**' to Light Gray... so the embedded XML 'disappears' in code view



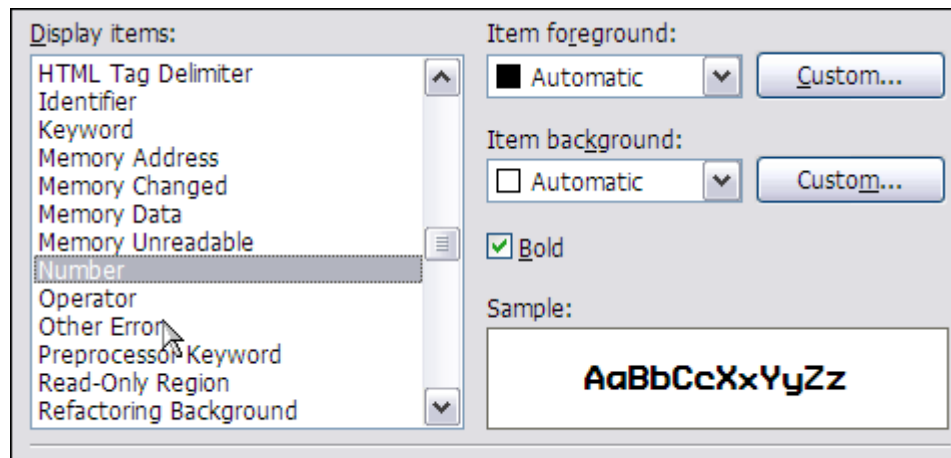
Set '**Comments**' to Dark Yellow... to differential from code



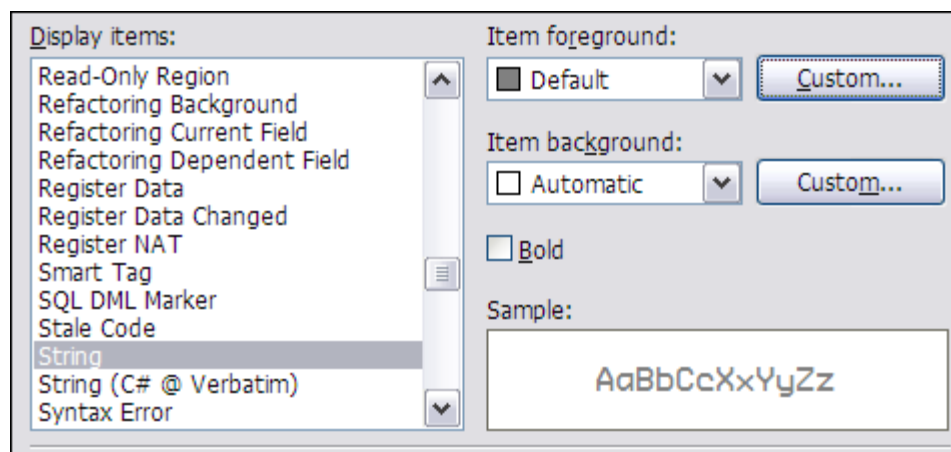
Set '**Keyword**' to Blue... to stand out and provide feedback while coding



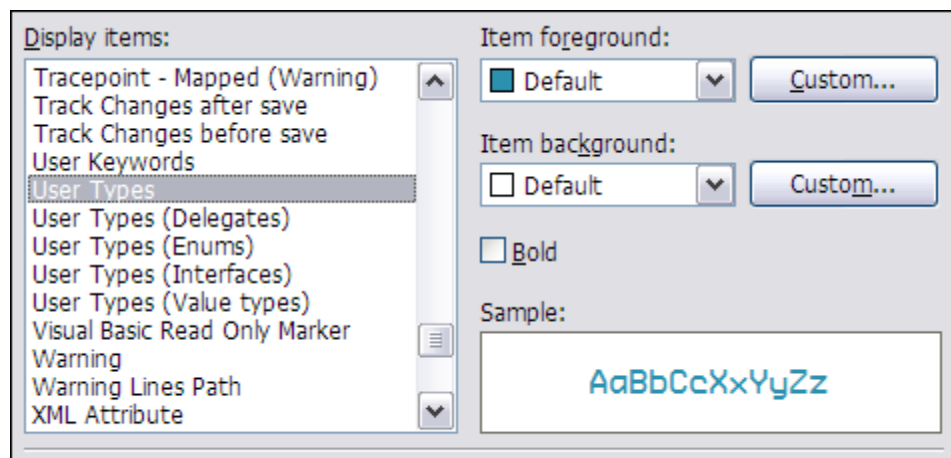
Set '**Number**' to Bold Black... so constants stand out



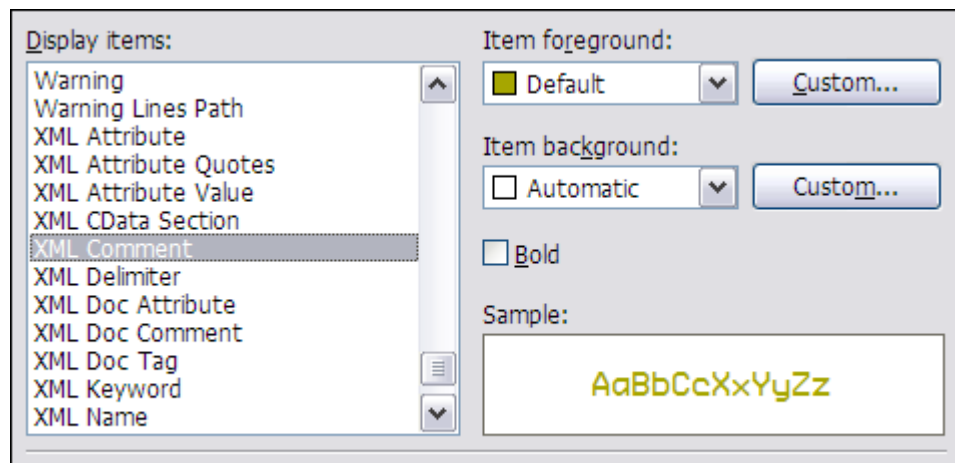
Set '**String**' to Dark Grey... to differential from code



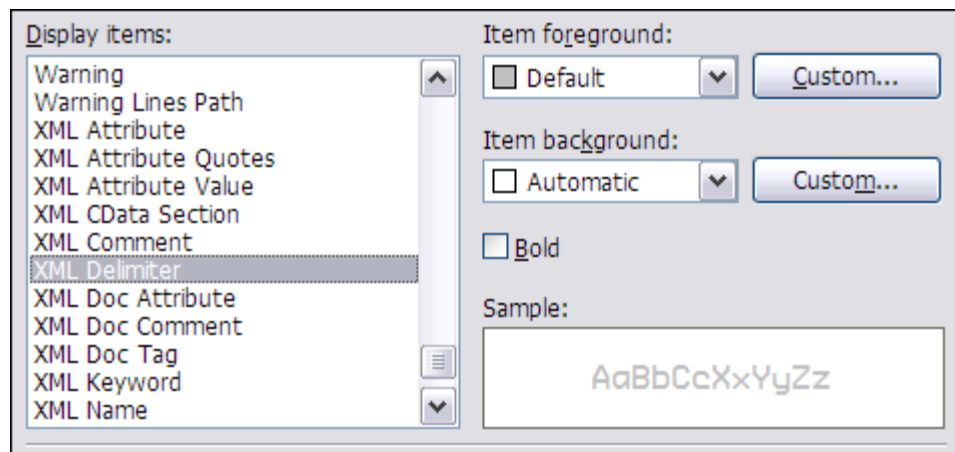
Set '**User Types**' to Teal... to provide feedback while coding, differentiating from built-in types and keywords



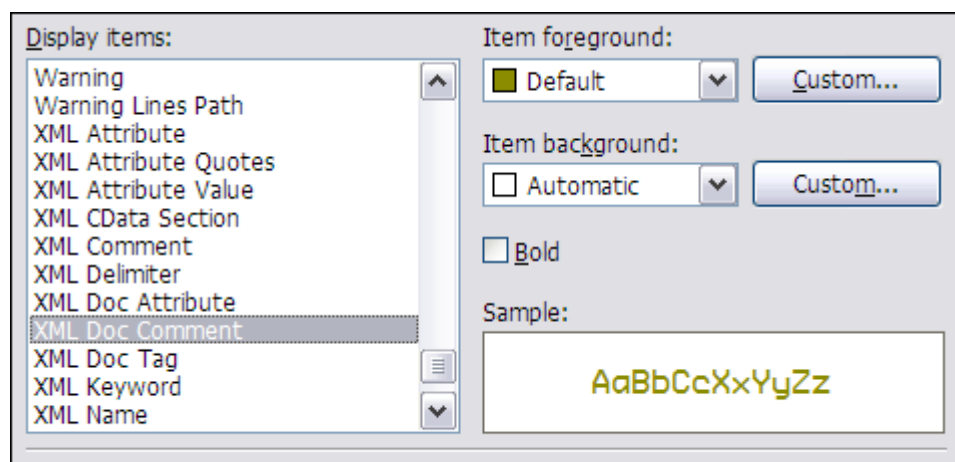
Set '**XML Comment**' to Dark Yellow... to match the other code documentation



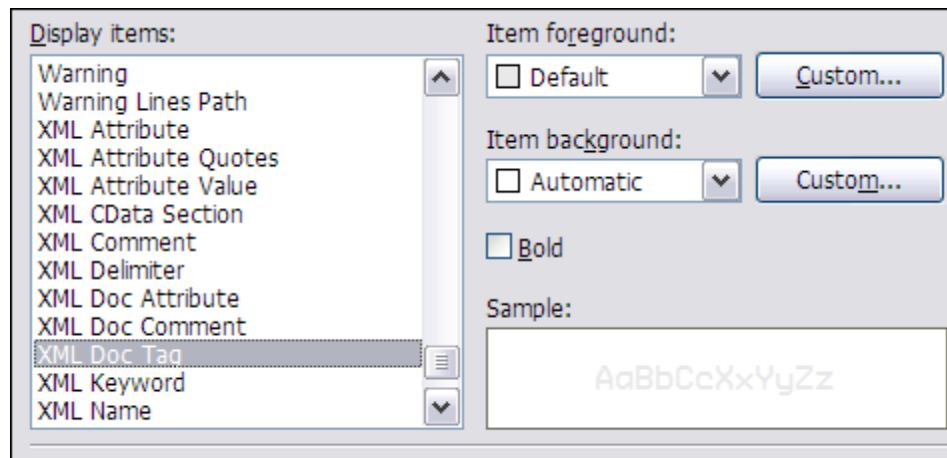
Set '**XML Delimiter**' to Light Grey... so they 'disappear' in code view



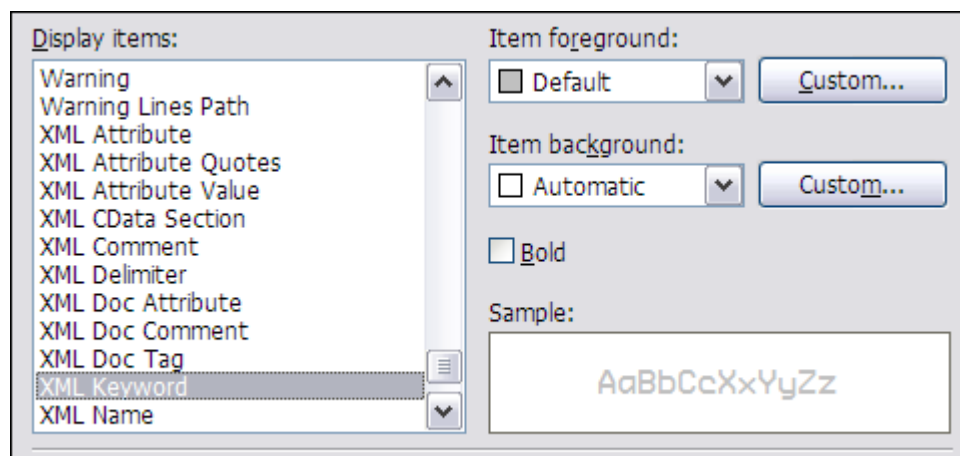
Set '**XML Doc Comment**' to Dark Yellow... to match the other code documentation



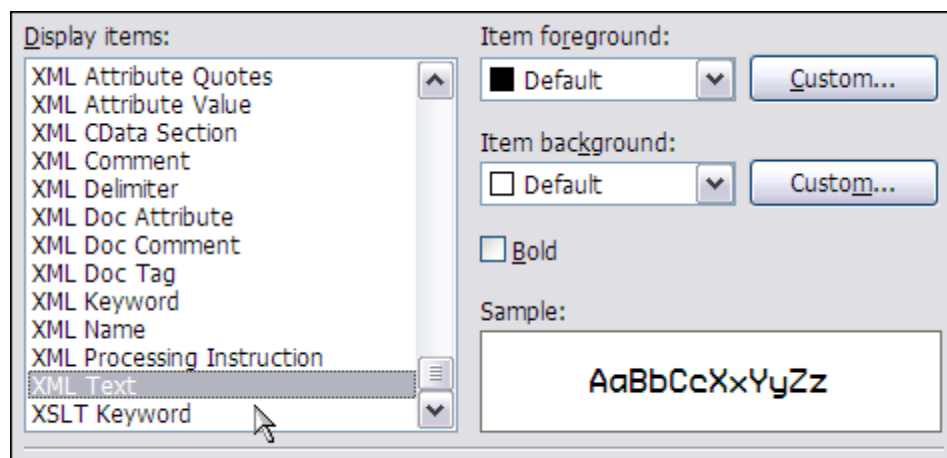
Set '**XML Doc Tag**' to Light Grey... so they 'disappear' in code view



Set '**XML Keyword**' to Light Grey... so they 'disappear' in code view



Set '**XML Text**' to Black... so they stand out in the XML documentation headers



Task: Pay very close attention to the naming standards in **ClassTemplate (MSDN Guidelines).cs**

```
* CAPITALIZATION CONVENTIONS:
```


```
* Pascal Casing - 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 Casing - 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. For example: IO
```

```
* Rules: 1) When an identifier consists of multiple words, do not use separators, such as underscores ("_") or hyphens ("-"), between words.  
* Instead, use casing to indicate the beginning of each word.  
* 2) Do use Pascal casing for all public member, type, and namespace names consisting of multiple words.  
* 3) Do use camel casing for parameter names.  
* 4) Do not use acronyms in identifiers unless they are widely known and well understood.  
* Do capitalize both characters of two-character acronyms, except the first word of a camel-cased identifier.  
* Do capitalize only the first character of acronyms with three or more characters, except the first word of a camel-cased identifier.  
* Do not capitalize any of the characters of any acronyms, whatever their length, at the beginning of a camel-cased identifier.  
* 5) Do not capitalize each word in so-called closed-form compound words.  
* These are compound words written as a single word, such as "endpoint" or "doorway".
```

```
*  
* Identifier Case Example Embedded Acronym Examples  
*-----
```

Class	Pascal	AppDomain	SepScanner
Enumeration type	Pascal	ErrorLevel	GepVehicleID
Enumeration values	Pascal	FatalError	Hhp
Event	Pascal	ValueChanged	CcrwConveyorStop
Exception class	Pascal	WebException	IOException, ParameterException
Read-only static field	Pascal	RedValue	GmStockName
Read-only const field	Upper	BIT00	CCRW_STOPCODE
Interface	Pascal	IDisposable	ICrwStoppable
Method	Pascal	Tostring	ToGepIONode
Namespace	Pascal	System.Drawing	Gep.Tracking
Parameter	Camel	typeName	ccrwConveyor
Property	Pascal	BackColor	CcrwTrimConveyor

```
*  
* Acronyms Pascal GepIO, GepIONode, GmVehicle, CcrwConveyor
```



The screenshot shows a Visual Studio code editor with the file 'ClassTemplate(MSDN Guidelines).cs' open. The code defines a namespace 'MicroCODE.Subsystem' and includes a region 'MSDN_ClassGuidelines' containing documentation for naming classes, interfaces, and derived classes. The documentation specifies that interfaces should start with 'I', derived classes should inherit from 'Exception', and interface names should be prefixed with 'I'.

```

*/
MSDN_CodingGuidelines
namespace MicroCODE.Subsystem
{
    MSDN_NamespaceGuidelines

    using ...

    #region MSDN_ClassGuidelines
    /*
    Do name classes, interfaces, and value types with nouns, noun phrases, or occasionally adjective phrases, using Pascal casing.
    Do not give class names a prefix (such as the letter C).
    - Interfaces, which should begin with the letter 'I', are the exception to this rule. (Like IDisposable).
    Consider ending the name of derived classes with the name of the base class.
    - For example, Framework types that inherit from Stream end in 'Stream', and types that inherit from Exception end in 'Exception'.
    Do prefix interface names with the letter 'I' to indicate that the type is an interface.
    Do ensure that when defining a class/interface pair where the class is a standard implementation of the interface, the names differ
    only by the letter 'I' prefix on the interface name.
    - For example, the Framework provides the 'IAsyncResult' interface and the 'AsyncResult' class.
    */
    }
}

```



StyleCop Settings for MicroCODE Projects

Task: Set up StyleCop rules for every MicroCODE project... overview of rules:

StyleCop 4.3

[Send comments on this topic.](#)

StyleCop 4.3 Rules

The StyleCop tool provides warnings that indicate style and consistency rule violations in C# code. The warnings are organized into rule areas such as documentation, layout, naming, ordering, readability, spacing, and so forth. Each warning signifies a violation of a style or consistency rule. This section provides an explanation of each of the default StyleCop rules.

In This Section

[Documentation Rules](#)

Rules which verify the content and formatting of code documentation.

[Layout Rules](#)

Rules which enforce code layout and line spacing.

[Maintainability Rules](#)

Rules which improve code maintainability.

[Naming Rules](#)

Rules which enforce naming requirements for members, types, and variables.

[Ordering Rules](#)

Rules which enforce a standard ordering scheme for code contents.

[Readability Rules](#)

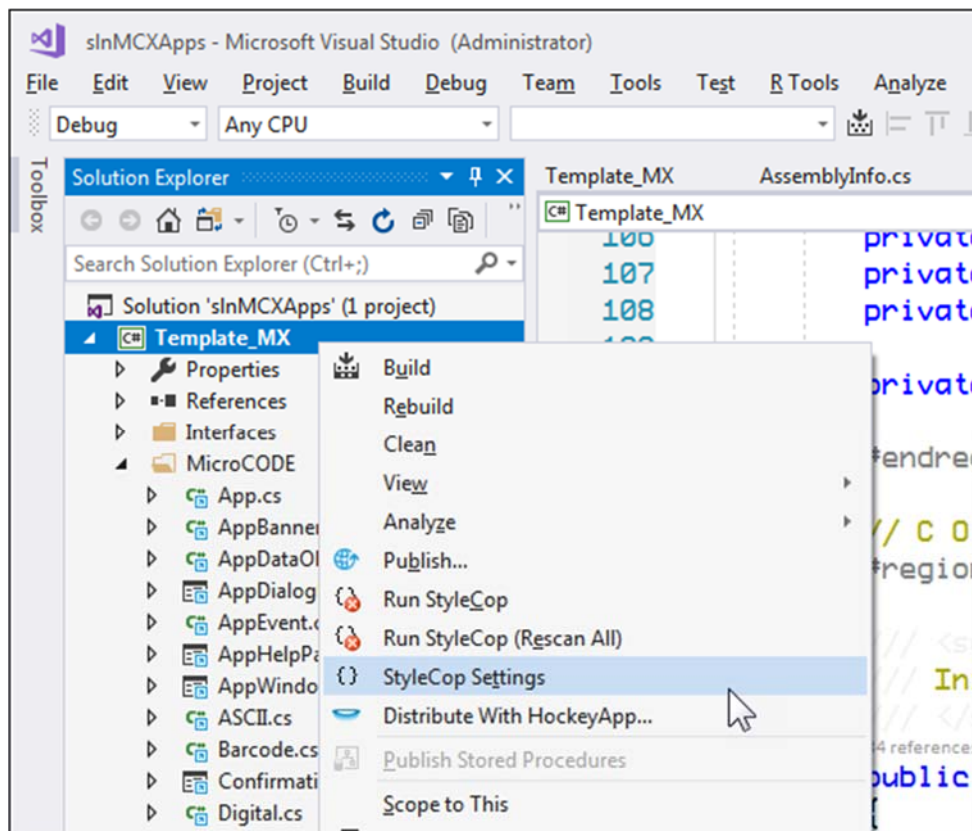
Rules which ensure that the code is well-formatted and readable.

[Spacing Rules](#)

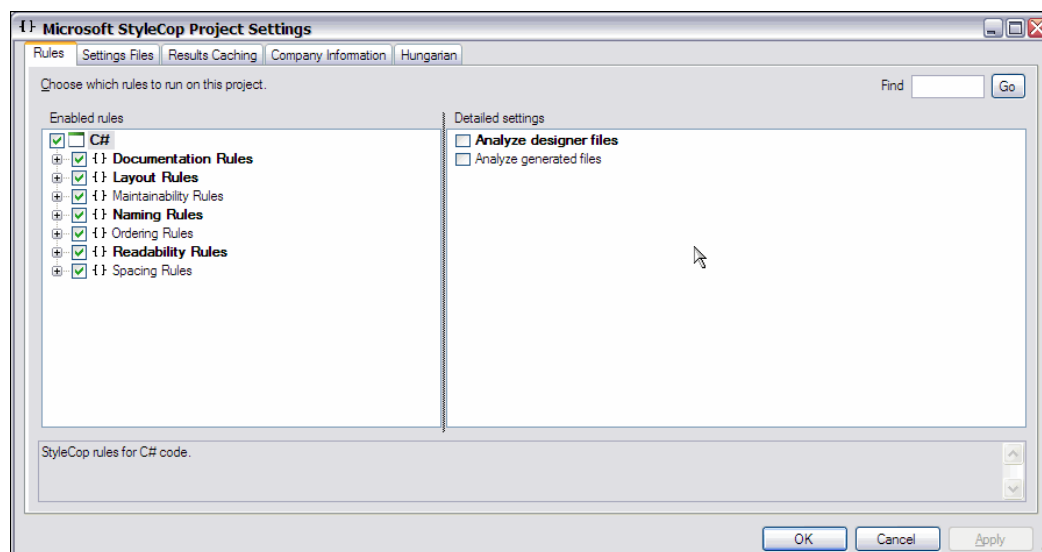
Rules which enforce spacing requirements around keywords and symbols in the code.



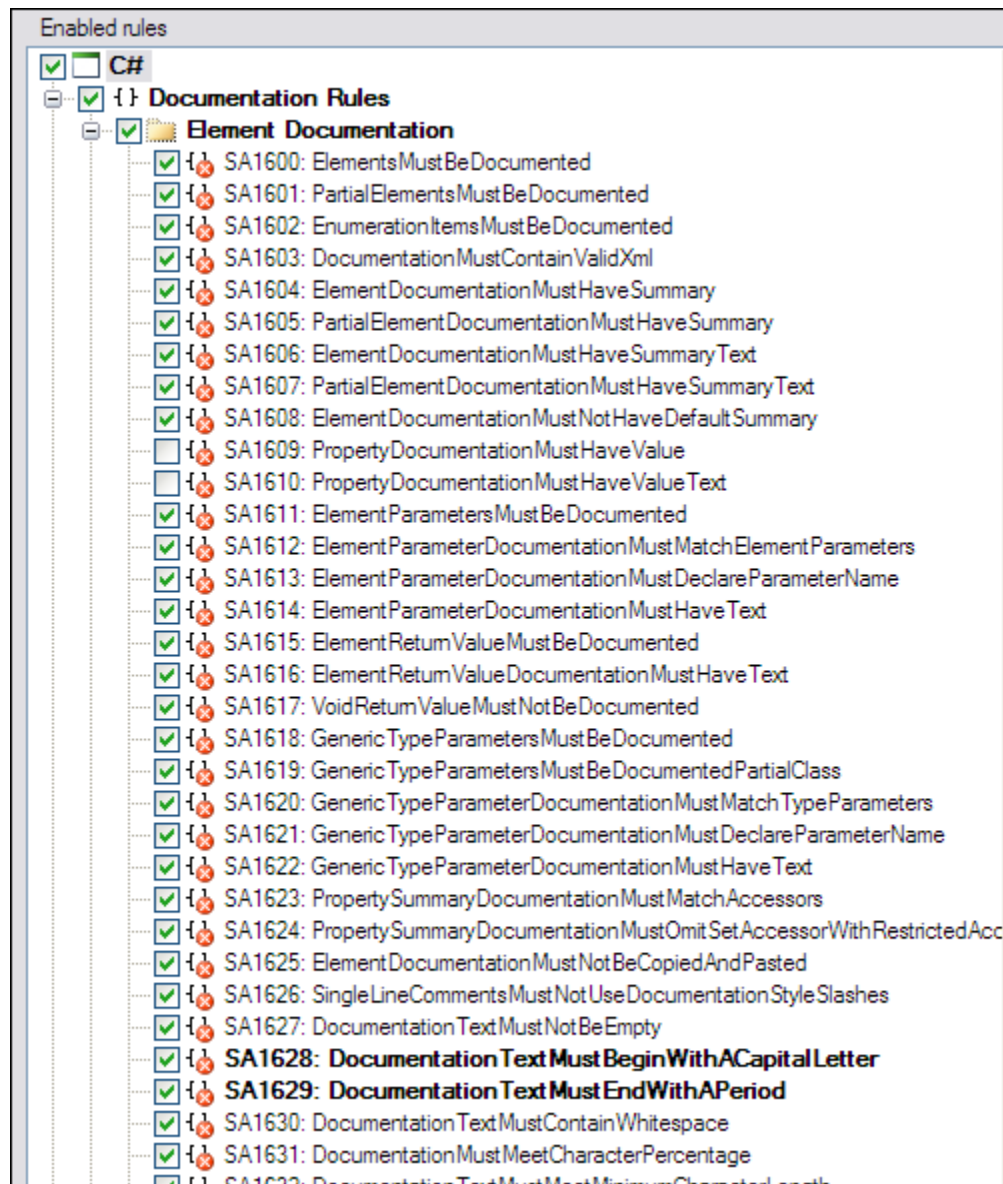
Note: Access rule settings by right-clicking the project name and selecting 'StyleCop Settings')



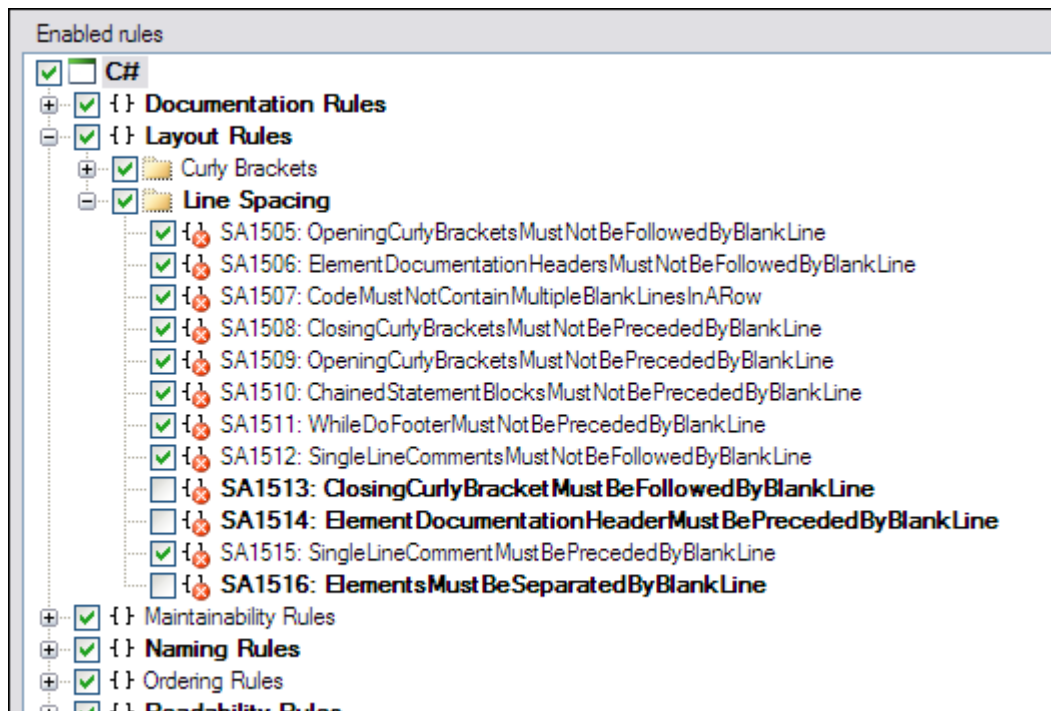
Note: Items in **bold** are modified from the StyleCop defaults... do ***not*** analyze VS generated or designer code, they both fail Microsoft's own tests.



Note: Turn on **SA1628** and **SA1629** to ensure all documentation produces readable sentences for extraction by Enterprise Architect



Note: Turn off SA1513, SA1514 and SA11516 to allow for compact private methods as shown in the example below... public methods should follow these rules



```
// Stores dialog control values for 'operator' into 'Configuration'
private void WriteOprParams()...

// Stores dialog control values for 'JobDataGrid' into 'Configuration'
private void WriteGridParams()...

// Verify 'Auto-Advance' is in the correct state
private void AutoAdvanceCheck()...

// Verify 'Keep-At-Top' is in the correct state
private void KeepAtTopCheck()...

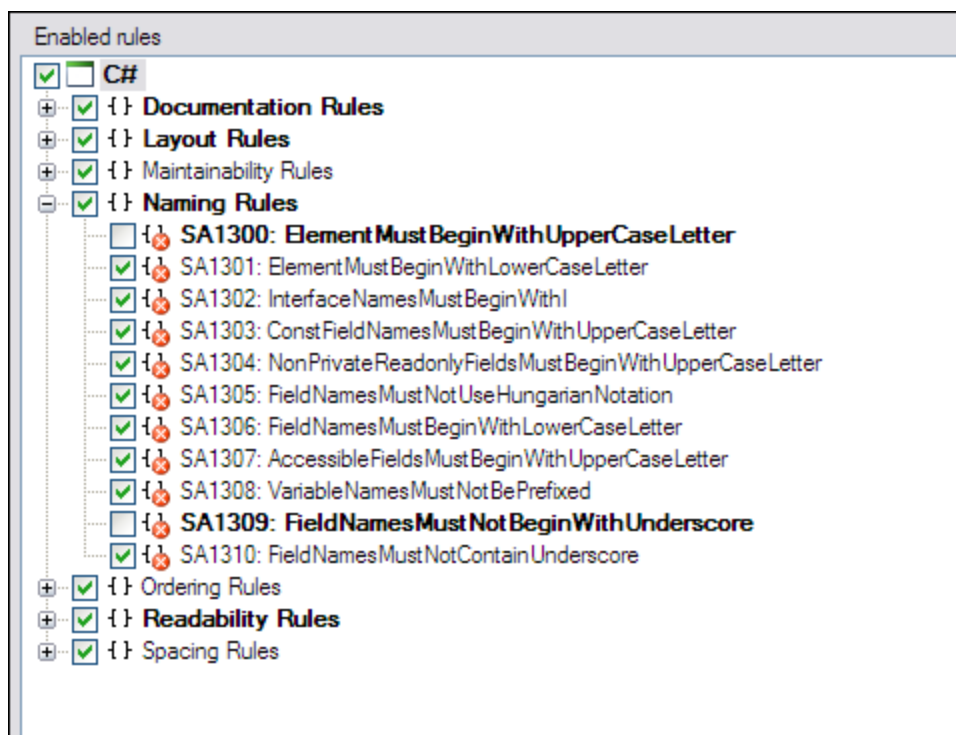
// Check for Unique Column Names
private bool PerformColumnChecks()...
private bool CheckForUniqueColumn(int count, string[] names, out string[] names)
private bool CheckForNotBlank(int count, string[] names, out string[] names)

// Validation of I/O Configuration and results display
private void CheckIOBitUsage()...
private int CountIOBitUsage(int bit)...

// Enable/disable EPA Controls based on configuration
```



Note: Turn off SA1300 and SA1309...



Turn off **SA1300** so we can follow the **MSDN Guideline** to name method parameters and method level variables with a lower-case letter.

Example: Method level variables named with a starting lowercase letter.

```

/// <summary> ...
public bool Read(SEP.ProgressBar progressBar)
{
    int rowIndex = 0;
    int bdxGetNextIndex, bdxQuantityUsed;

    _f = _cell.GmpC1x.IsConnected;

    try
    {
        // First check validity of BDX Tag Reference
        _cell.UpdateBdxPosition();
    }
}

```

```

*
*   CAPITALIZATION CONVENTIONS:
*   -----
*
*   Pascal Casing - the first letter in the identifier and the first letter of
*                   each word in the identifier are capitalized. You can use Pascal case for identifiers of three or more
*                   words. For example: BackColor
*
*   Camel Casing - the first letter of an identifier is lowercase and the first
*                   letters of the remaining words are capitalized. For example: backgroundColor
*
*   Uppercase - All letters in the identifier are capitalized. For example: BACKCOLOR
*
*   Rules: 1) When an identifier consists of multiple words, do not use separate
*           words. Instead, use casing to indicate the beginning of each word.
*           2) Do not use Pascal casing for all public member types and namespaces.

```



Turn off **SA1309** so we can follow the new practice of using an underscore to begin class wide variables. These were named **m_variableName** (Module Wide) or **g_variableName** (Global Variable) in past standards. Global variables are ***not*** to be used, Class Properties are to be used for this purpose. Module wide variables are now to be named **_variablename** and placed in the “**Fields**” Region.

Example: Module/Class wide variables in the Field Region.

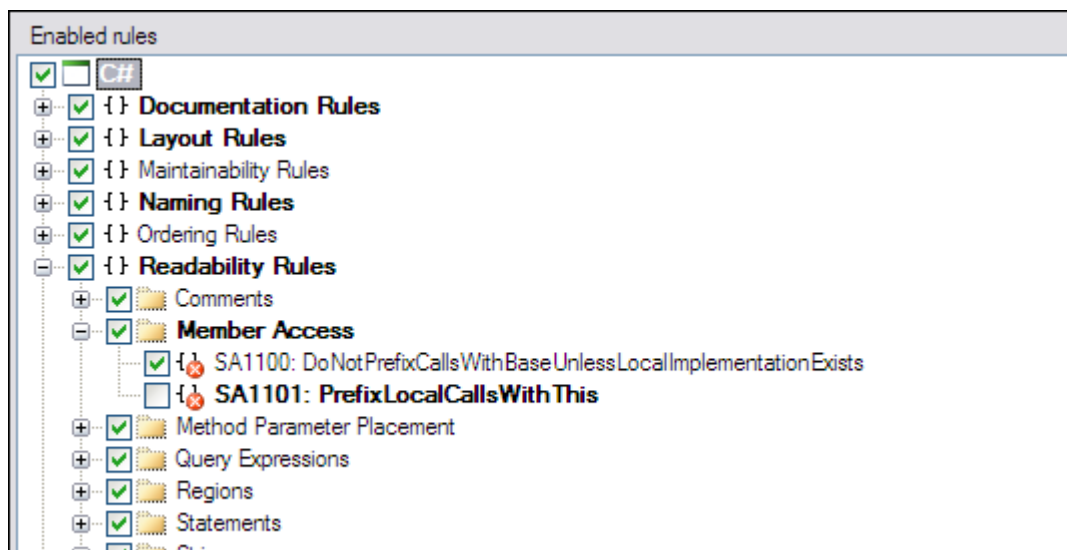
```
// F I E L D S
#region Fields

private DataGridView _grid;
private VScrollBar _vsb;
private HScrollBar _hsb;

private object _renderer; // Renderer for the grid

private Color[] _rowBackColor; // Color to fill Row
private Color[] _rowTextColor; // Color to use for
private int _rowsConfigured; // Number of rows cu
```

Note: Turn off **SA1101**; the use of **this._variable** is redundant if the other conventions are followed. Any variable outside of a method's parameters or private data will begin with an underscore “_” if it belongs to the object (this), all other data must come from method or property calls to other objects.

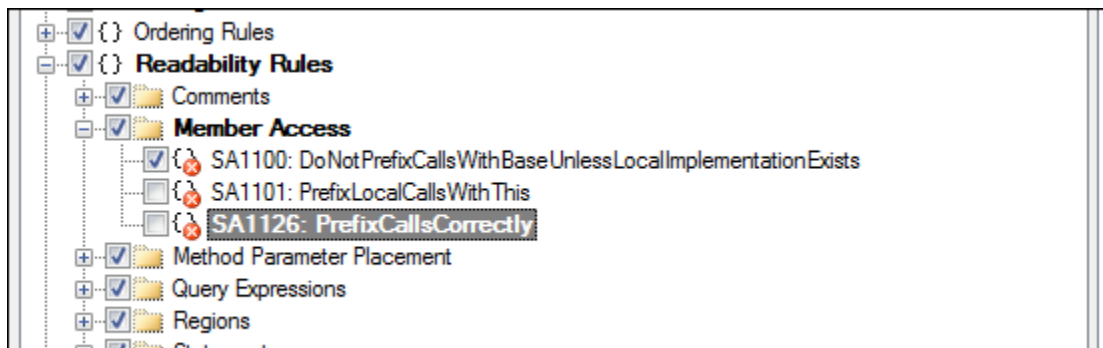


Note: [8:45:37 AM 03-Oct-2012] Changes for VS2008 – StyleCop 4.7

Note: Turn off **SA1126**; the use of **this._variable** is redundant if the other Microsoft conventions are followed. Any variable outside of a Method's Parameters or an Object's Properties will begin with an underscore "_" if it belongs to the object (this), all other data must come from property calls to other objects.

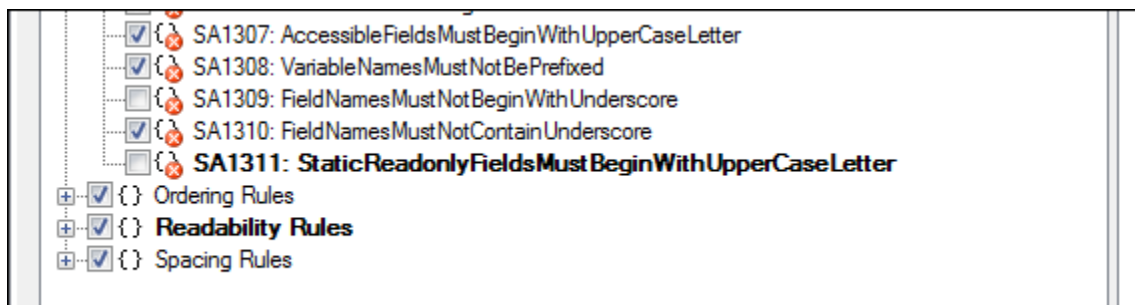
Summary of Member Access Style in use by MicroCODE...

<privateField> - name begins with '' underscore, 'this.' implied
I<instanceProperty> - name begins with uppercase letter, 'this.' implied
m<methodParameter> - name begins with lowercase letter, method scoped

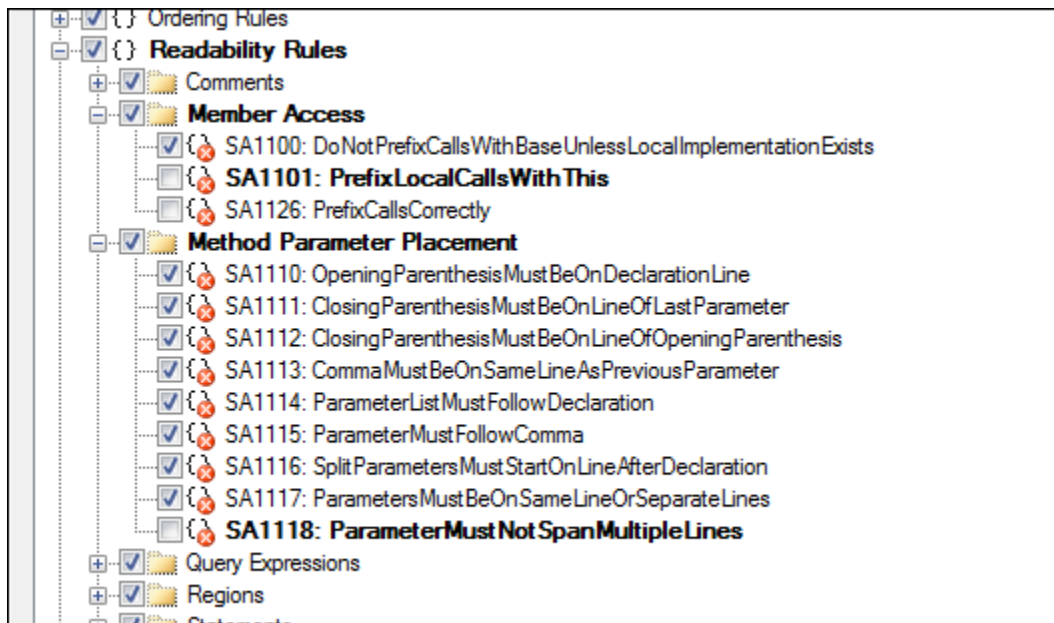


Note: Turn off **SA1311**; class 'lists' (for List Processing) are **all** private static arrays and so are named following the rule...

<privateField> - name begins with '' underscore, 'this.' implied



Note: Turn off SA1118; the restriction on multi-line parameters prevents simple concatenation for readability...



```

if (!allTriggered)
{
    Point dialogPoint = new Point(((AppScreen.ScreenWidth / 2) - (ConfirmationDialogBox.Width / 2)), (AppScreen.ScreenHeight - Confirmati

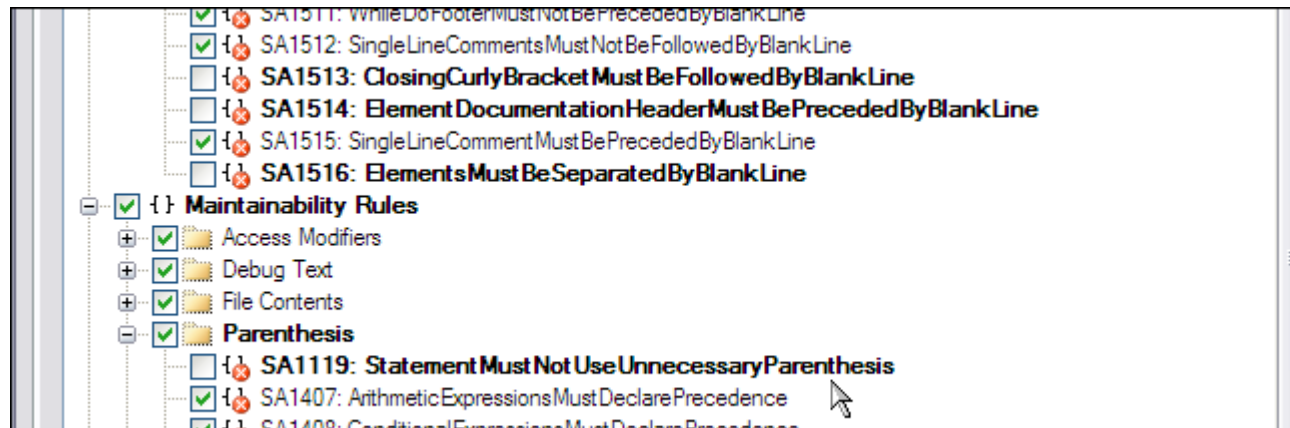
    // warn user, this ain't gonna work...
    ConfirmationDialogBox untriggeredMessage = new ConfirmationDialogBox(
        dialogPoint,
        "Untriggered Action Behavior",
        "One or more Behaviors are configured without a means of activation. " +
        "Use the 'Always' prefix or configure Trigger(s) to active each Behavior.",
        "Correct configuration or uncheck 'Enable'...");

    untriggeredMessage.GetOk();
}

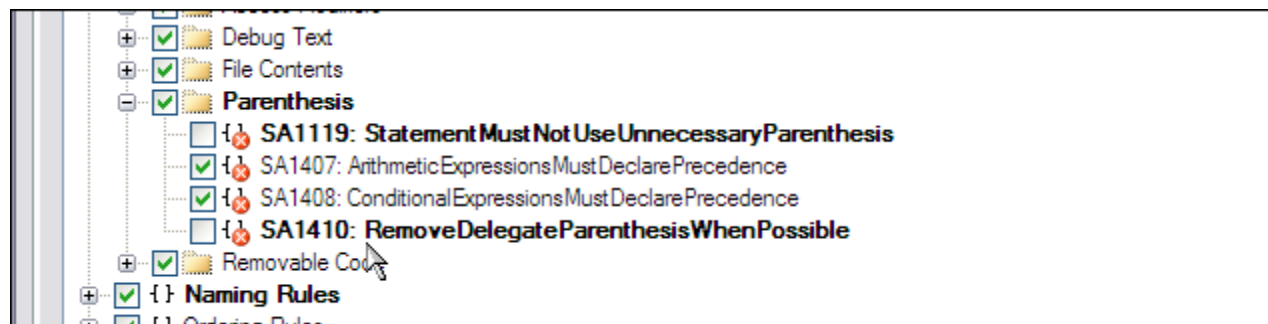
```



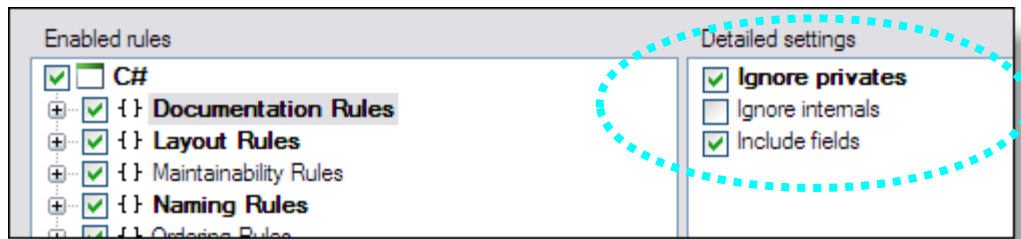
Note: Turn off **SA1119**; the restriction on extra parenthesis in logically statements. Extra parenthesis makes the original coder's intentions clear to the new reader.



Note: Turn off **SA1410**; the restriction on extra parenthesis in delegate calls. Extra parenthesis makes the original coder's intentions clear to the new reader.

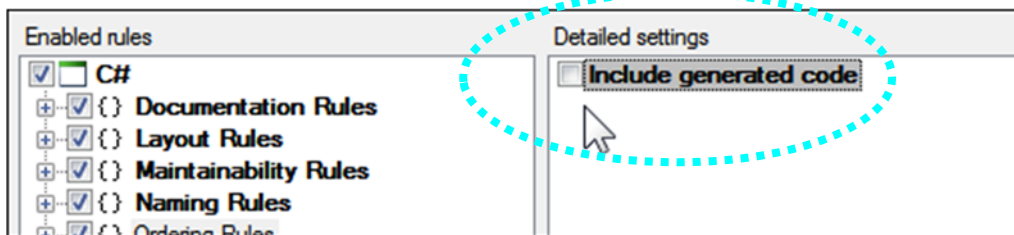


Note: Under “Documentation Rules” check “**Ignore privates**” and “**Include fields**”

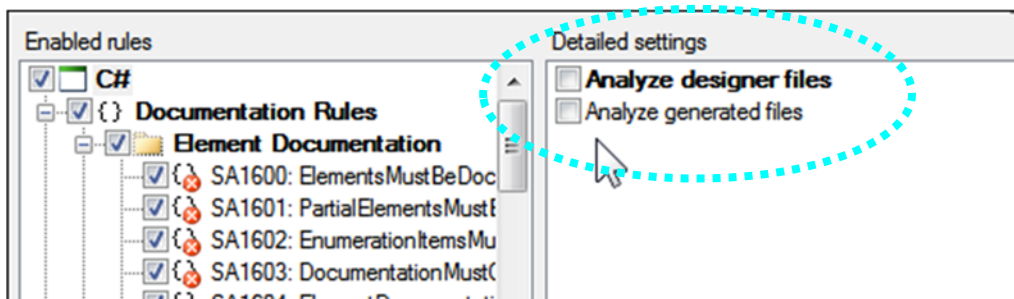


This alleviates the need to create XML documentation for private methods, fields, etc; but forces us to document any public fields.

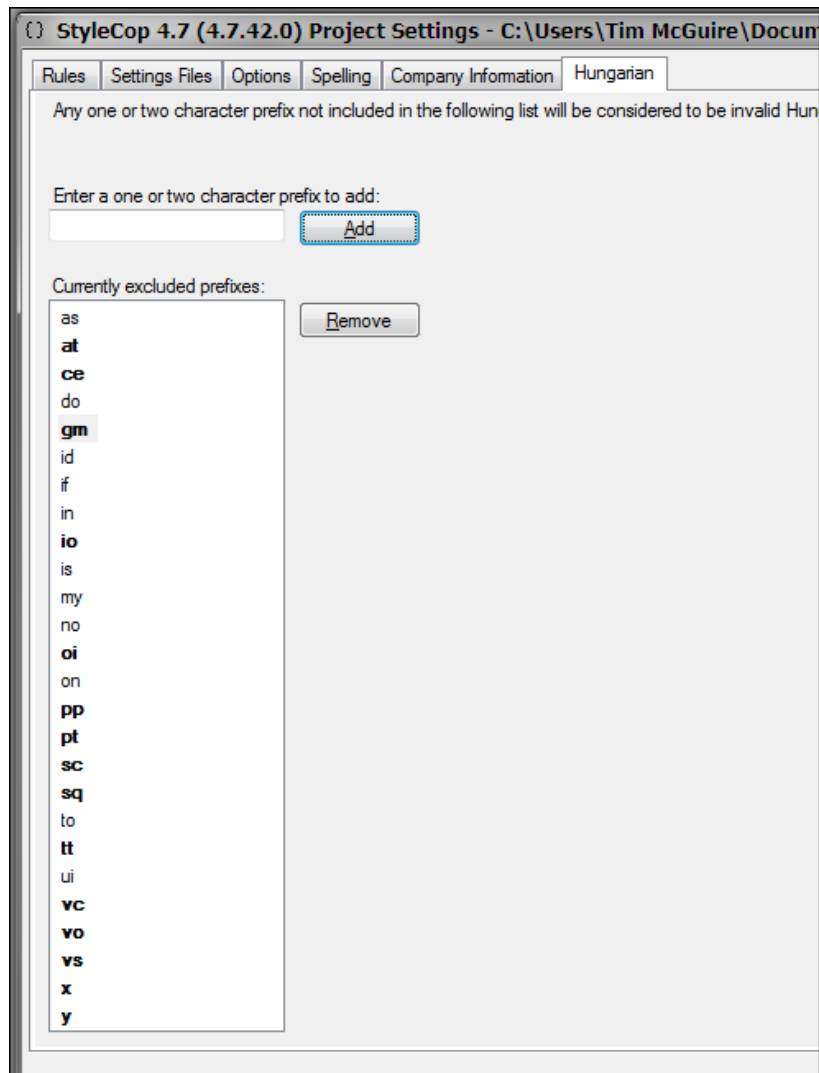
Note: Under “Ordering Rules” uncheck “**Include generated code**”, generated code is not to be manually modified to pass these rules. (Regeneration would undo any work done to pass these rules).



Note: Under “C#” uncheck “**Analyze designer files**” and “**Analyze generated files**”, generated code is not to be manually modified to pass these rules. (Regeneration would undo any work done to pass these rules).

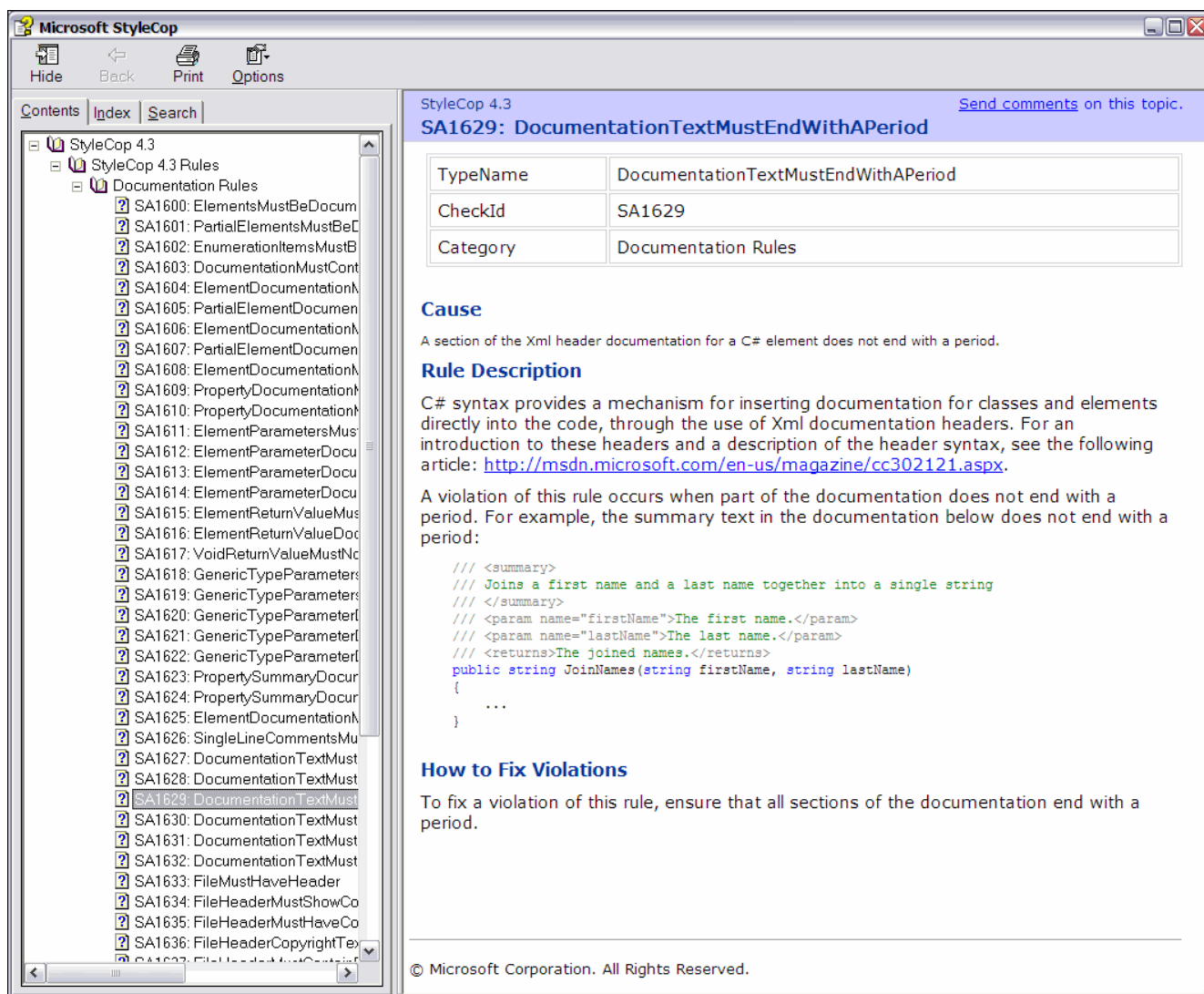
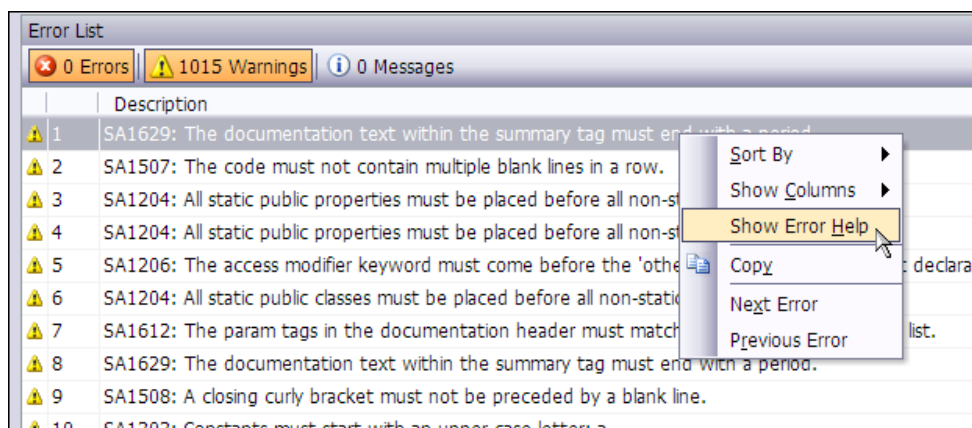


Note: Under “Hungarian” add our common two letter acronyms for EPA types so StyleCop does take them for Hungarian Notation.



Note: For help on any StyleCop violation, right-click on the message and select “**Show Error Help**”

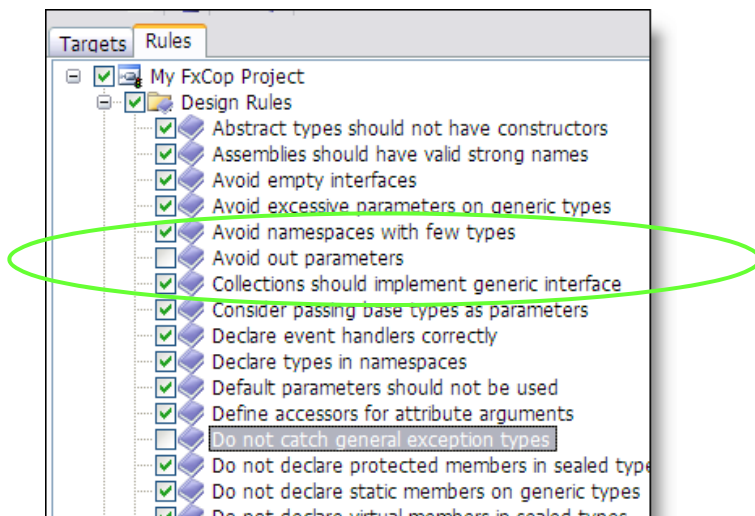




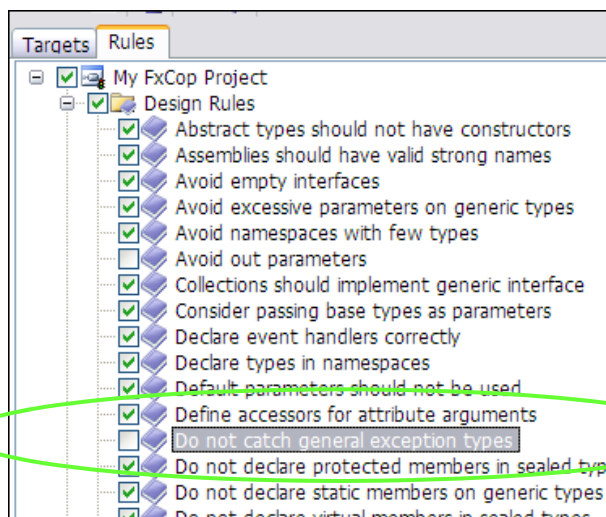
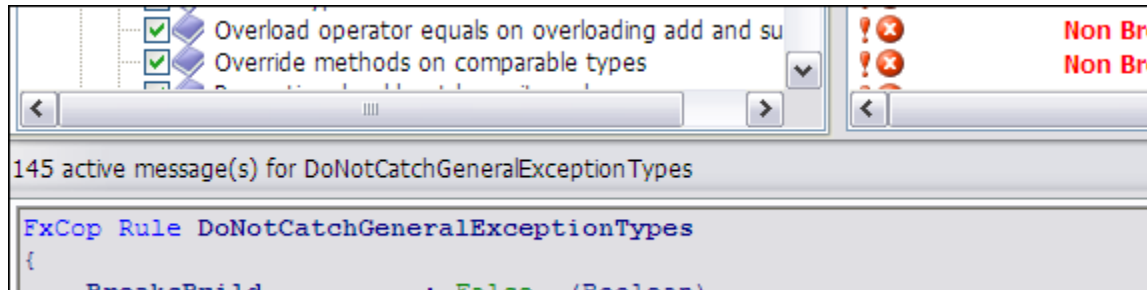
FxCop Settings for MicroCODE Projects

Task: FxCop usage – after any changes to a MicroCODE App (or Test Program) are complete and you build an executable (.EXE), run FxCop and have it analysis the program code.

Task: Remove the rule “Avoid out Parameters”, this pattern is used in heavily overloaded methods to distinguish signatures.



Task: Remove the rule “DoNotCatchGeneralExceptionTypes”, all ‘catch’ statements end with handling general exceptions by logging and continuing operation of the App as possible.



```
// Fire the event to all subscribers, remove any that are a problem
foreach (EventHandler<IONode.IONodeEventArgs> eventDelegate in eventDelegates)
{
    try
    {
        // Fire event asynchronously on caller's thread
        eventDelegate.Invoke(this, e);
    }
    catch (Exception exception)
    {
        App.Exception(FullClassName, exception);

        // Dead, Abandoned, or Invalid Delegate, remove from subscribers
        IONodeEvent -= eventDelegate;
    }
}
```



Note: Ideal exception handling should be done as shown in this example.

Exceptions that are expected—and can be dealt with in the method—should be caught, handled and execution should continue. These exceptions are not normally logged.

Exceptions that are possible, but not expected—and can be dealt with—are caught, handled and logged to provide feedback to the developers.

As a final step—to keep unexpected exceptions from crashing the entire application—any other unexpected exceptions are caught and logged, and—because they are not deemed possible—there is normally no method code to deal with the exception.

```

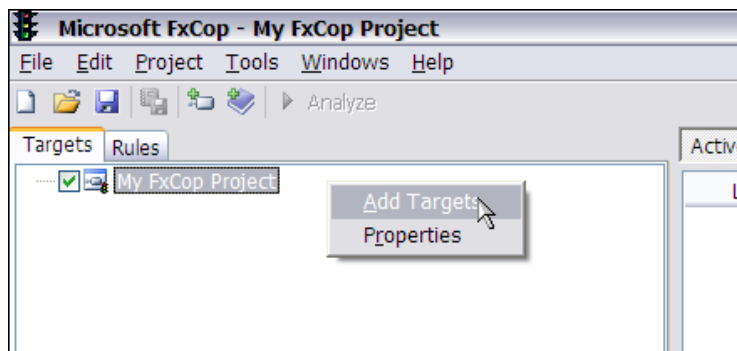
    }
    catch (ExpectedException1)
    {
        // Handle 1st expected exception, no need to log...
        Code...
    }
    catch (ExpectedException2)
    {
        // Handle 2nd expected exception, no need to log...
        Code...
    }
    catch (ExpectedExceptionN)
    {
        // Handle Nth expected exception, no need to log...
        Code...
    }
    catch (UnexpectedButPossibleException1)
    {
        // Handle 1st unexpected but possible exception and log skip display...
        Code...
        App.Exception(FullClassName, exception, false);
    }
    catch (UnexpectedButPossibleException2)
    {
        // Handle 2nd unexpected but possible exception and log skip display...
        Code...
        App.Exception(FullClassName, exception, false);
    }
    catch (UnexpectedButPossibleExceptionN)
    {
        // Handle Nth unexpected but possible exception and log, skip display...
        Code...
        App.Exception(FullClassName, exception, false);
    }
    catch (Exception exception)
    {
        // log unexpected, seemingly impossible exception, and display...
        App.Exception(FullClassName, exception, true);
    }
    finally
    {
        // do clean or finalization...
        Code...
    }

```

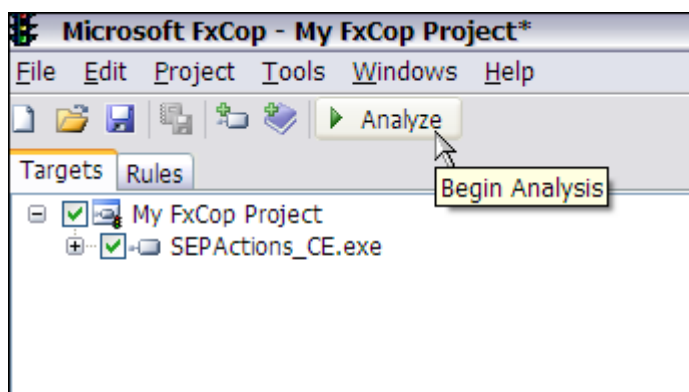
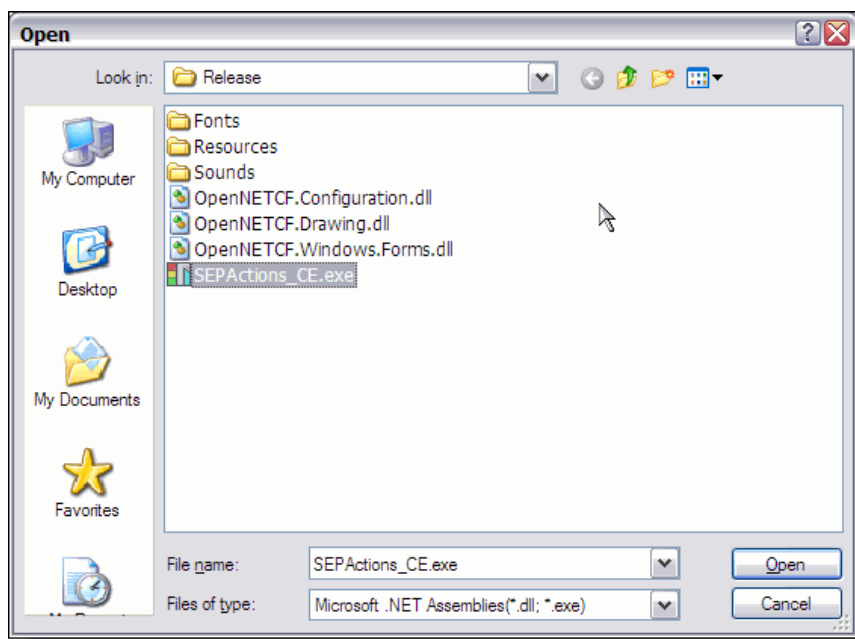
This construct guards the App from crashing when a single thread is hit with an unexpected exception. Elevating an unexpected exception—by not catching the General Exception—leaves the exception to some high level to deal with it, at some point it would be caught, either as a specific type (difficult above the source) or as the General exception. Logging the exception allows the App support team to diagnose these situations.



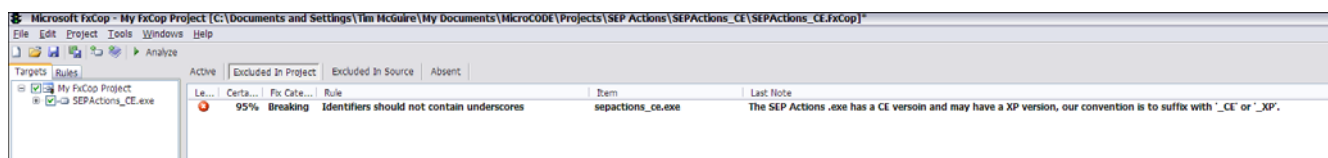
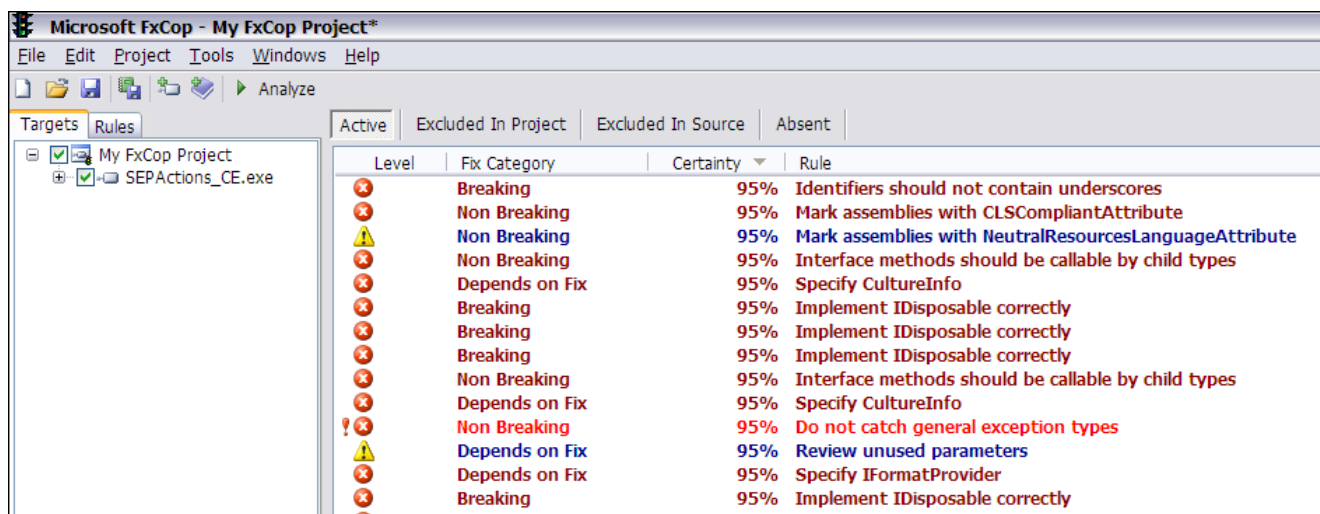
Note: Run FxCop, right click on the 'My FxCop Project' and select 'Add Targets'



Drill to the .exe, select and click 'Open'...

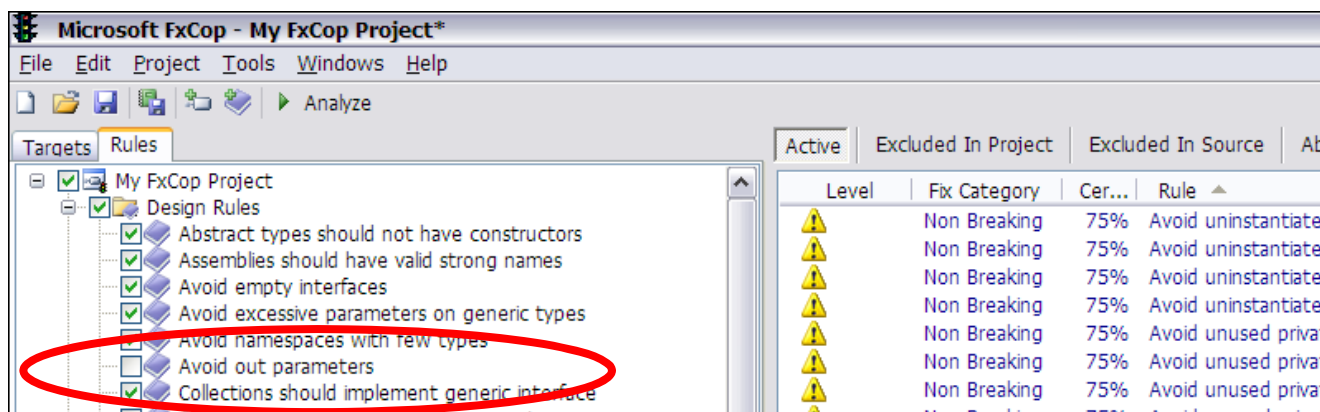


Note: Correct FxCop all issues. If something cannot or should not be fixed or changed, right click on the issue and document why it is being excluded.

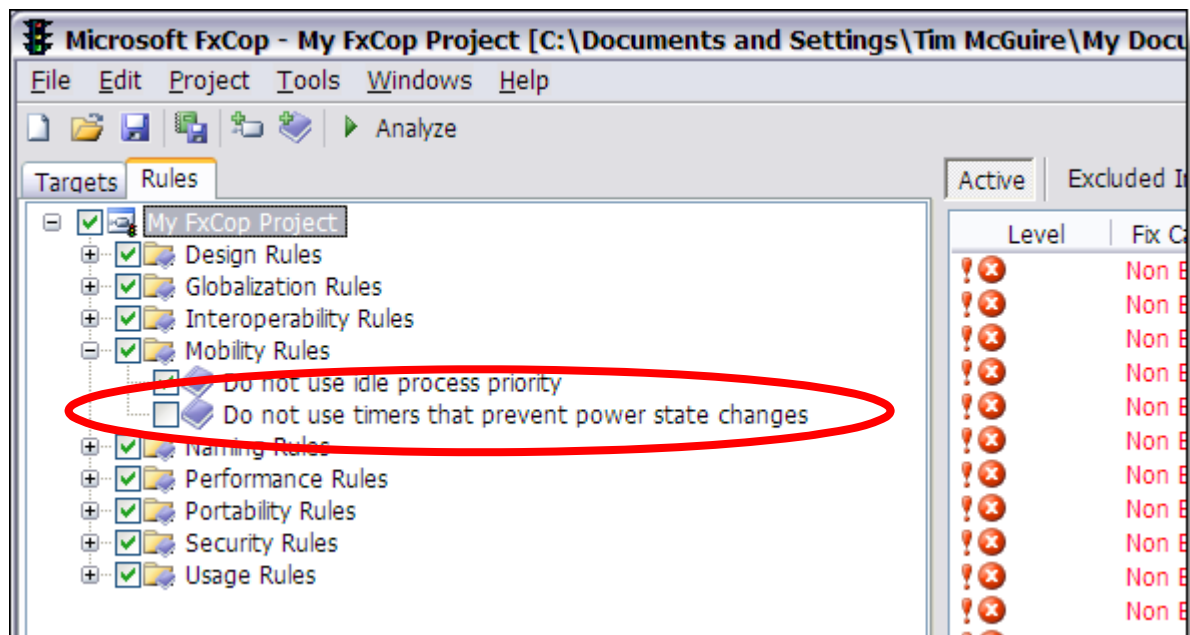


Task: Suppress FxCop by setting by an FxCop Project and using the following Rule changes...

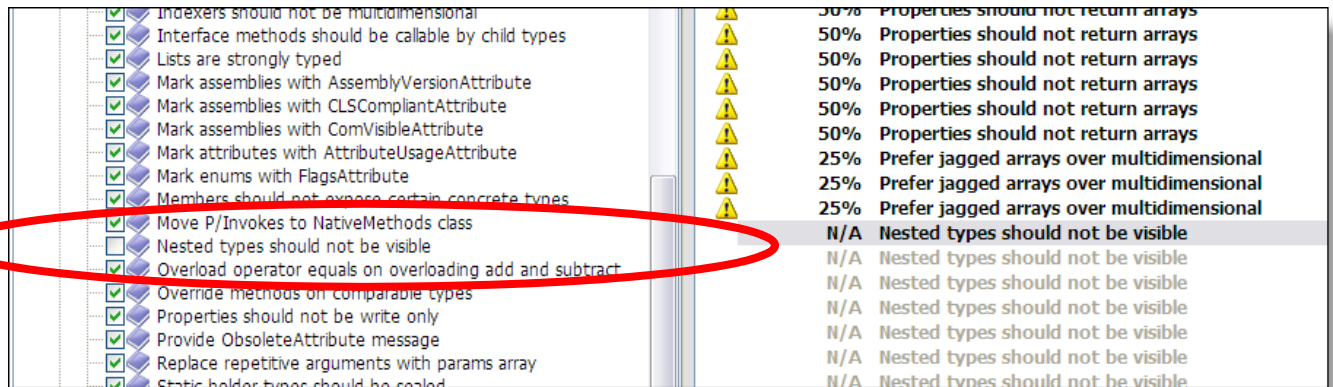
Note: Turn off **CA1021**; methods can only be overridden by differently parameter lists, in the case of many MicroCODE utility classes this is accomplished using 'out' instead of returning a value.



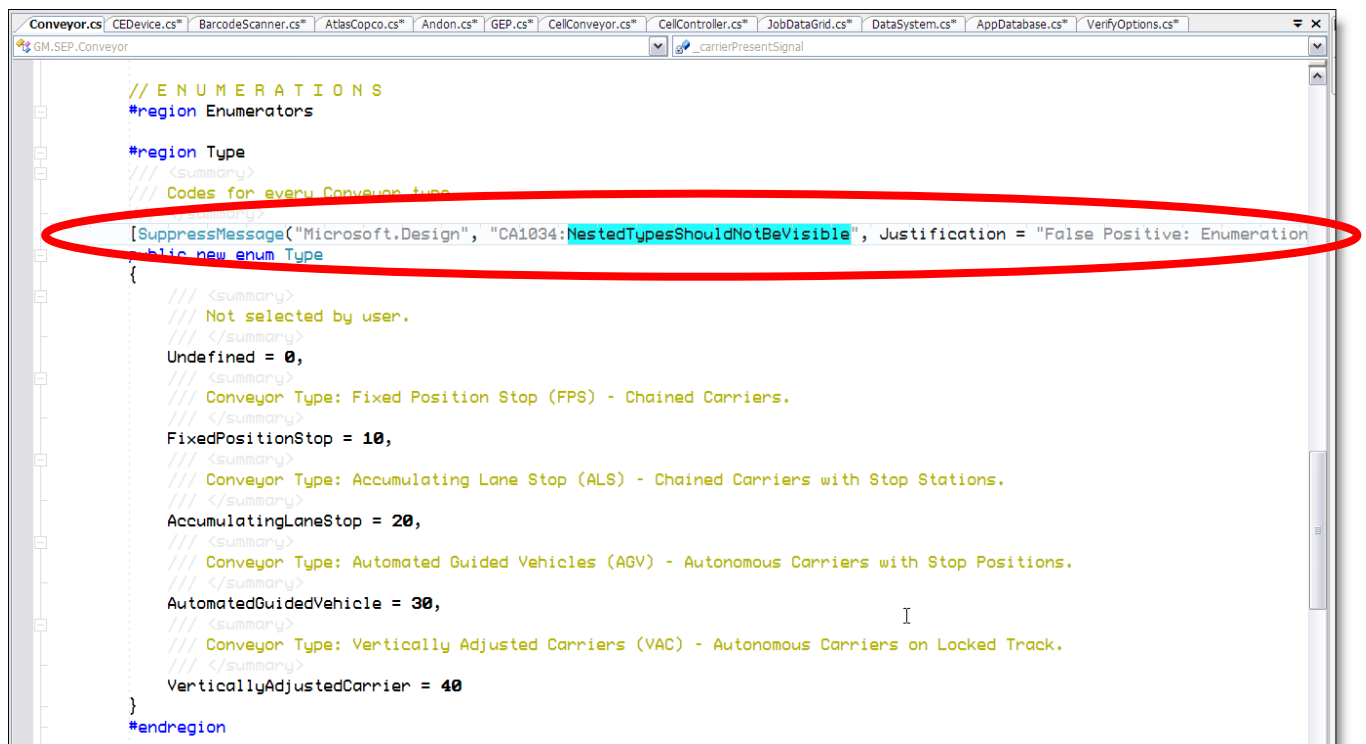
Note: Turn off **CA1601**; this warning concerning Timers with intervals less than 1 Seconds does not apply to our application, this warning is for battery operated devices.



Note: Turn off **CA1034**; this warning concerning Nested Types interferes with ENUMs and any sub-classes that act as parameters to a parent class, much like an enumeration.



With this rule enabled every enum in the solution requires a suppression statement like this one:




```
Unmatched in-source exclusion for NestedTypesShouldNotBeVisible
{
  Target      : GM.SEP.UsbHidIONode+Parameters (IntrospectionTargetType)
  Help        : http://msdn2.microsoft.com/library/ms182162\(VS.90\).aspx (String)
  Category    : Microsoft.Design (String)
  CheckId     : CA1034 (String)
  RuleFile    : Design Rules (String)
  Info        : "Do not use public, protected or protected internal (
                Protected Friend) nested types as a way of grouping
                types. Use namespaces for this purpose. There are very
                limited scenarios where nested types are the best design.
                Also, nested type member accessibility is not clearly
                understood by all audiences. Enumerators are exempt
                from this rule."
  Note        : Microsoft.FixSep.Common Note (Note)
  Created     : 6/12/2010 10:29:51 PM (DateTime)
  LastSeen    : 1/1/0001 12:00:00 AM (DateTime)
  Status      : ExcludedInSource (MessageStatus)
  Fix Category : Breaking (FixCategories)
```

This also prevents the creation of sub-classes—owned by a single parent class—that are best kept in the same file as the parent class for simplifying code organization and storage. Using Namespace, as suggested by Microsoft's Rule, does not provide the same benefit and is actually harder to understand than nested types, IMHO.



Note: [9:49:46 AM] Use the embedded XML Dictionary for every MicroCODE project...

Place this in your project directory (where the FxCop Project (.FxCop) and Solution (.sln) Files are located)

This is placed in My Documents > MicroCODE >

The parent directory of all projects.



CustomDictionary.xml

|



General Practices

Note: Value comparisons, test variable against a limit, not a limit against a variable...

Use...

```
if (ObjectBeingTested == TestValue)
{
    ...
}
```

***Not*...**

```
if (TestValue == ObjectBeingTested)
{
    ...
}
```

Example:

Use...

```
if (Object == null)
{
    ...
}
```

***Not*...**

```
if (null == Object)
{
    ...
}
```

Reason:

The habit of placing the Object being tested first matches the way we speak, "If the object is null" and it makes ordered comparisons logically correct like this...

Use...

```
if (Object >= LowLimit)
{
    ...
}
```

***Not*...**

```
if (LowLimit <= Object)
{
    ...
}
```



Note: Naming for Events, Posting, Firing, etc.

PostExampleEvent - does any internal state maintenance and *then* calls 'OnExampleEvent'

OnExampleEvent - checks for null and *then* fires event

ExampleEvent() - fires the event (from within 'OnExampleEvent', using .NET delegates

```

/// <summary> ...
public void PostActionEvent(Action.ActionEventArgs e)
{
    if ((_state != Action.EpaState.Disabled) && ((e.State == Action.EpaState.Enabled) || (e.State == Action.EpaState.Faulted)))
    {
        _state = Action.EpaState.Undefined;
    }

    // Only elevate state, reset elsewhere
    if (e.State > _state)
    {
        _state = e.State; // update internal state to 'new job' or higher state only, see definition of 'EpaState'.
        _lightStackStep = 0; // reset light stack cycle to show new state immediately

        OnActionEvent(e); // now fire event to any subscribers
    }
}

private void OnActionEvent(Action.ActionEventArgs e)
{
    // Fire the event to all subscribers
    if (ActionEvent != null)
    {
        ActionEvent(this, e);
    }
}

```

If errors within Delegates need to be controlled explicitly call each for a **try-catch** construct...

Alternative code for 'OnExampleEvent ()' to catch errors and log, or kill dead delegates...

```

// Fire the event to all subscribers, remove any that are a problem
if (EnteringEvent != null)
{
    // get the list of event 'subscribers'
    Delegate[] eventDelegates = EnteringEvent.GetInvocationList();

    foreach (EventHandler<EventArgs> eventDelegate in eventDelegates)
    {
        try
        {
            // Fire event asynchronously on listener's thread, passing
            eventDelegate.Invoke(this, new EventArgs());
        }
        catch (Exception exception)
        {
            // Problem in Delegate, remove from subscribers
            ///* EnteringEvent -= eventDelegate; * don't remove, any
            // Log exception, don't show user...
            App.Exception(FullClassName, exception);
            App.DebugEvent(FullClassName, eventDelegate.ToString() + "
        }
    }
}

```



Note: Naming of Windows Forms Controls

The Windows Forms Designer code is *not* included in StyleCop by default configuration and we do not turn it on. We do however; have several standards concerning the naming of the form controls:

- 1) **All Controls must be named explicitly by function, no generated named are to be left after Design.**
- 2) **After the design is complete all Tab Indices should be set (or disabled) in a logically fashion.**
- 3) **All Controls are to be named using the following hybrid format:**

pfxSYS_CamelCaseName

Where:

pfx = a three letter acronym – Control Type ‘Hungarian Notation’

SYS_ = a three letter acronym—followed by one underscore—for the Base Class of the Object being viewed or configured

CamelCaseName = Parameter name in standard Camel Case, no underscores

- 4) **All Controls related Event Handler use the default name generated by Visual Studio:**

pfxSYS_CamelCaseName_Event

Where:

_Event = the Event name—preceded by one underscore—being handler

Standard prefixes for Controls – (3) character acronyms:

btn	=	Button
cbo	=	ComboBox
chk	=	CheckBox
ctx	=	ContextMenu
dtp	=	DateTimePicker
gbx	=	GroupBox
iml	=	ImageList
lbl	=	Label
lst	=	ListView
mnu	=	MenuItem
num	=	NumericUpDown
pic	=	PictureBox
pnl	=	Panel
rbtn	=	Radio Button
txt	=	TextBox
ssc	=	StatusBar
tvw	=	TreeView



Task: [11:36:23 AM] Do **not** use Event Delegates for custom EventArgs, this design pattern has been deprecated in Microsoft standards, use 'typed' EventHandlers which became available with 'Generics':

Don't use this:

```
// D E L E G A T E S
#region Delegates

/// <summary>
/// Delegate for App Database event handlers.
/// </summary>
/// <param name="sender">The object throwing the event.</param>
/// <param name="e">The App Database event being throw.</param>
public delegate void AppDatabaseEventHandler(object sender, AppDatabaseEventArgs e);

#endregion // Delegates
```

Use this:

```
// E V E N T S
#region Events

/// <summary>
/// Event indicating something in the Database has changed.
/// </summary>
public event EventHandler<AppDatabaseEventArgs> AppDatabaseEvent;

#endregion // Events
```

```
// Connect/Disconnect Dialog Box to/from a running DataSystem -- COMMON to all DataSystem Types
private void ConnectDataSystem()
{
    // Connect checkboxes to running I/O...
    ConnectIO();

    // Set form methods as delegates to perform actual data display in form
    _dataSystem.AppDatabaseEvent += new EventHandler<AppDatabaseEventArgs>(AppDatabase_AppDatabaseEvent);

    // Start up the dialog 'Test Mode' animation, shows the dialog is 'subscribed to events'
    App.Clock.T0500Milliseconds += new EventHandler<EventArgs>(Running_500msEvent);
}

private void DisconnectDataSystem()...

// Open/Close class specific device level testing -- COMMON to all DataSystem Types
private void OpenDeviceConnection()...
private void CloseDeviceConnection()...

#endregion
```



Note: Code Metrics

Code Analysis and Code Metrics

The Visual Studio Code Analysis Team Blog

This Blog


[Email](#)

Syndication

[RSS 2.0](#)

[Atom 1.0](#)

Search



I want to...

[File a bug/suggestion](#)

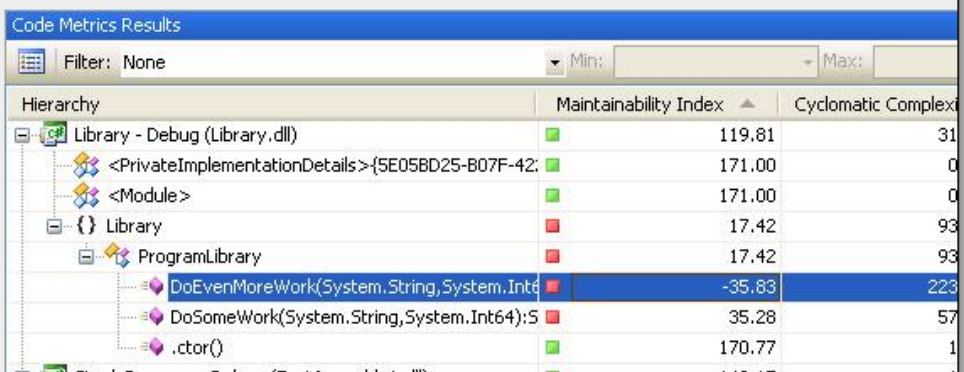
[Ask us a question](#)

[Get our latest bits](#)

[Download tools, samples and power toys](#)

Announcing Visual Studio Code Metrics!

Announcing the new Code Metrics feature for Visual Studio 'Orcas'! Available in Visual Studio 2019, the Code Metrics Results tool window shows results in the Code Metrics Results tool window:

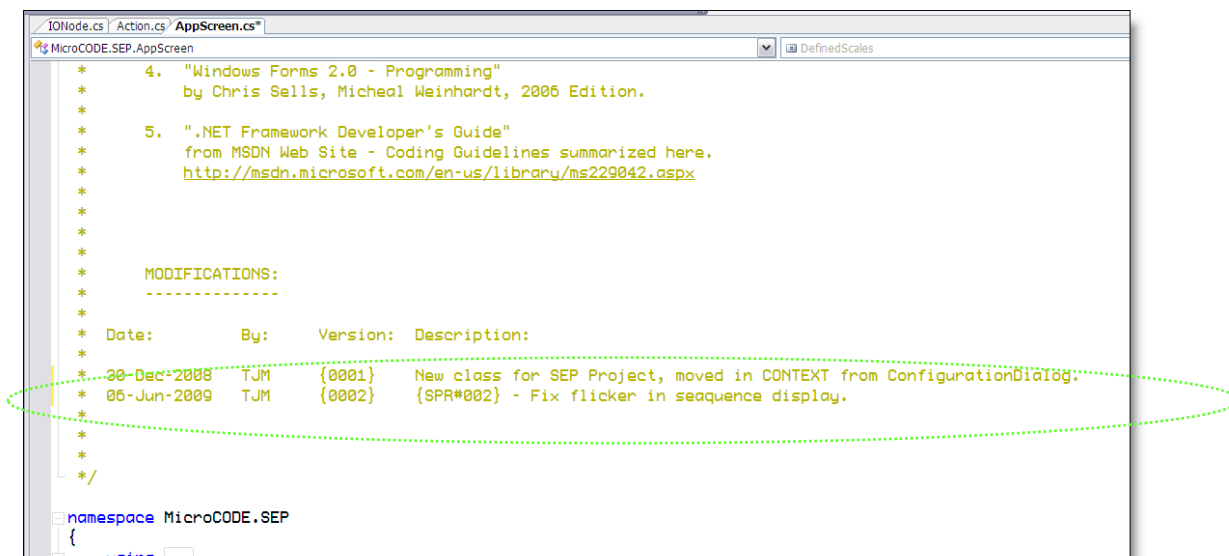


Hierarchy	Maintainability Index	Cyclomatic Complexity
Library - Debug (Library.dll)	119.81	31
<PrivateImplementationDetails>{5E058D25-B07F-42...	171.00	0
<Module>	171.00	0
Library	17.42	93
ProgramLibrary	17.42	93
DoEvenMoreWork(System.String, System.Int64):S	-35.83	223
DoSomeWork(System.String, System.Int64):S	35.28	57
.ctor()	170.77	1

Note: Code Maintenance – procedure for edited released code

1 – Never modify released code without an SPR or SER, i.e.: Software Problem Report (SPR) or Software Enhancement Request (SER)

2 – When modifying a class start with the edit history in the header... add Date, Initials, SPR/SER id and description of change



3 – Mark all edits as follows:

(The four slashes '////' mark a line in StyleCop as an independent comment free of placement rules)

////* = permanently removed, for reference only

```

////~ = not implemented, template code or partially developed

```

```

/// <summary>
/// Initializes a new instance of the AppScreen class.
/// </summary>
public AppScreen()
{
    try
    {
        ///* {SPR#002} removed the splash screen
        ///* // Display the Splash Screen
        ///* _splashScreen.Show();

        ///* {SPR#002} added to clean by display on entry
        /// Get the 'splash' shown...
        Application.DoEvents();

        // Required for Windows Form Designer support
        InitializeComponent();

        ///~ {SPR#002} not implemented yet, add after feature x is implemented
        ///~ // Alarms object
        ///~ _alarms = Context.Alarm;


        // Application Configuration
        appParameters = Context.Cfg.AppParameters;
    }
}

```



4 – Examine all Class usage and perform regression tests on affected classes and/or functions.

Document all tests using a **Software Acceptance Tests (SAT)** forms for the affected applications features.

	General Motors Corporation – Global Error Proofing (GEP) System SAT Form for a Torque Tool (TT) Action
GM Global Error Proofing (GEP) System System Acceptance Test (SAT) – Torque Tool	
GMGEP-C22 (SAT - Error Proof Torque Tool - TT).doc	
<hr/>	
Torque Tool (TT) – SAT Record	
This is the final acceptance test by the GEP Deployment Team before turning this Torque Tool (TT) over to the plant for production.	
GEP Panel: Cell:	Pn - Panel Name, Cn
Track Zone:	Track Zone Name, Conveyor Name

