.NET Programming Standards
&
Reference Guide

Version 1.1

Office of Information & Technology
Department of Veterans Affairs

REVISION HISTORY

DATE	VER.	DESCRIPTION	AUTHOR	CONTRIBUTORS
8/2014	1.0	Original first draft using the latest OI&T publishing standard.	Raymond L. Steele (Sr. Technical Writer / IT Specialist)	Billy Collins Rick Jones Carlton Dodd Danila.Manapsal
12/8/14	1.1	Final draft to be submitted to TRM	Billy Collins Rick Jones Carlton Dodd Danila.Manapsal	Billy Collins Rick Jones Carlton Dodd Danila.Manapsal
6/18/15	0.5	First draft to be submitted to TRM	Workgroup	Workgroup
10/04/15	1.0	Revisions to wording from .NET Standards Committee inputs Initial submission to TRM	Carlton Dodd	Committee
5/24/16	1.1	Added Section 6, Fortify and 508 compliance	Billy Collins	Committee

ABSTRACT

The .NET Development Community has written and established this .Net Programming Standards and Reference Guide based on current industry and VA best practices. It was based upon the insight of experienced VA developers.

The .NET Standards Committee (NSC) is requiring the programmer's regular usage of StyleCop as an aid to make better VA code. StyleCop can quickly scan C# code to locate formatting standards violations, and will generally promote enhanced software maintainability.

This document is dynamic and will continue to evolve as it is meant to be a helpful tool to all .NET programmers.

Please read and follow the standards and conventions described in this document.

These standards apply to all new development and major releases. It is recommended for all published code.

The VA NSC is comprised of volunteers that champion these standards because there are proven VA benefits.

Feedback in the form of corrections or suggestions for improvement is encouraged. Comments should be sent to the VA OI&T .NET Standards Committee. <mailto:VAOITDotNetStandardsCommittee@va.gov>.

TABLE OF CONTENTS

.NET PROGRAMMING STANDARDS I

& I

REFERENCE GUIDE I

VERSION 1.0 I

OFFICE OF INFORMATION & TECHNOLOGY I

DEPARTMENT OF VETERANS AFFAIRS I

1) OVERVIEW 6

Introduction 6

Categories 7

Intended Audience 7

Acknowledgements 7

2) TECHNOLOGY 8

.NET Framework Language Selection (C# versus VB.NET) 8

Tools Used 8

3) NAMING CONVENTIONS 10

Capitalization Rules for Identifiers 10

General Naming Conventions 12

Names of Assemblies and DLLs 12

Names of Namespaces 13

Names of Classes, Structures, and Interfaces 13

Names of Methods 14

Names of Properties 14

Names of Events 14

Names of Variables or Parameters 14

4) COMMENTING AND DOCUMENTATION 16

General Comments 16

XML Documentation Comments 18

5) STYLE AND DESIGN 20

Style 20
Design 36
6) VA DEVELOPMENT POLICIES 46
Software Assurance (SwA) verification with Fortify 46
508 Compliance Standards 48
7) SENSITIVE INFORMATION PROHIBITED FROM PUBLIC DISTRIBUTION 49
8) APPENDICES 52
A) StyleCop Installation 52
B) StyleCop Rules to be Enforced in VA 52
C) StyleCop Rule Suppression 56
D) Web Resources 57

1) Overview

Introduction

This .NET Programming Standards and Reference Guide has been written to provide VA .NET programmers with a set of coding standards and conventions to follow when developing new .NET applications.

This document reflects solid .NET programming standards and best practices that promote a higher level of maintainability and readability within the software.

In addition, always use proper design and analysis techniques such as:

- Participate actively in individual and group code reviews.
- Build upon test-driven development efforts.
- Use continuous integration and coordinated implementation practices.
- Test locally before deployment globally.
- Use known VA best business practices and VA Lessons Learned.
- Stay up to date on current VA .NET standards.
- Look for .NET programming standards to evolve and move with it.

The VA benefits in the following manner:

- Improves the readability, and therefore, maintainability of code.
- Strongly improves development and technical discussions by offering a common reference point.
- Reduces the learning curve for new VA developers
- Reduces common coding issues/mistakes.
- Passes on quality .NET programming applications and functionality to the VA Enterprise Network

The importance and benefits of a consistent coding style are well known. This document draws from some of the industry standards StyleCop coding practices. Unlike guidelines and best practices, the concise set of standards outlined in this document, are meant to be enforced without exception. .

Consistency of coding style is more important than using a particular style. When a situation falls out of the scope of this document, experience and informed judgment should be used wherever doubt exists. Please propose additions to the standards if you believe you have a best practice that could benefit VA.

Suggestions for changes to this document should be sent to the VA OI&T .NET Standards Committee .<mailto:VAOITDotNetStandardsCommittee@va.gov>

Categories

This document is broken down into four broad categories of standards:

CATEGORY	DESCRIPTION
TECHNOLOGY	Overview of language, tools, and environment settings for use in .NET development in VA
NAMING CONVENTIONS	Overview of casing styles, naming rules, and name choice for .NET identifiers
COMMENTING AND DOCUMENTATION	Overview of comment types in .NET source code and XML comments used to generate documentation
STYLE AND DESIGN	Overview of StyleCop rules that increase the readability and maintainability of .NET source code
VA DEVELOPMENT POLICIES	Overview of select VA Development Policies and how to comply during .NET development

Intended Audience

This document is intended for .NET developers, development team managers, systems architects, Software Quality Assurance (SQA), and technical writers.

Acknowledgements

This document is based on the coding style that is prevalent in Microsoft Developer Network (MSDN) example code, and should already be familiar to most developers. The guidelines presented here were not created in a vacuum. In the process of creating this document, the authors have scanned many existing .NET code conventions and guideline documents including MSDN Best Practice Guidelines.

2) Technology

.NET Framework Language Selection (C# versus VB.NET)

ASP.NET was developed to provide language interoperability (each language can use code written in other languages) across several programming languages (C#, VB.NET, F# etc…). Two of these languages have become very popular within the ASP.NET community in regards to console and web application development – C# (pronounced: C-Sharp) and VB.NET.

With the majority of the functionality differences between these two languages being insignificant, the major variation between them is how syntactically different these languages are. This can lead to developers using project time to determine which language to use internally. Additionally, there is time risk if a developer is not familiar with each language at a professional level.

C# has a large documentation advantage. At the time of this writing, the number of C# articles on the *MSDN Blogs* is approximately 27,000 compared to VB.NET at 8,000. Also, the extremely popular developer help-forum, *Stackoverflow.com*, has over 600,000 articles on C#, versus just over a tenth of that for VB.NET. This shows that C# has clearly become the industry standard choice of .Net languages. Therefore, all new development will be written in C# (other release rules below).

- All new .NET projects in VA will be written in C#.
- All Major releases of VA projects (e.g.: v1.x to v2.x) will use the existing language unless the team determines converting to C# will reduce the overall development time (this will likely be uncommon).
- All Minor releases of VA projects (e.g.: x.1 to x.2) will use the existing language.

If a project is using another .NET language, such as Visual Basic, the majority of the standards in this document will still apply and be useful. However, the team will not be able to use StyleCop (see "Tools Used") for automated checking of its included rules. Additional best practice guidelines may be found in the Microsoft Developer Network (MSDN) Development Guide.

Tools Used

Microsoft Visual Studio

Developers should use the latest version of Microsoft Visual Studio approved in the VA Technical Reference Model (TRM) http://trm.oit.va.gov/ToolPage.asp?tid=5670#, with appropriate approved plugins, including StyleCop (see below). Due to Project (`.csproj`) and Solution (`.sln`) file format incompatibilities between some versions, some teams may not be able to use the latest version of Visual Studio, but should make an effort to update as soon as feasible.

StyleCop

There are a number of technologies that can be used within the .NET Development Environment to aid developers in compliance with the .NET Programming Standards. The .NET Standards Committee has decided to use StyleCop, a simple, yet powerful tool for defining and enforcing programming standards. Instructions for installation of StyleCop may be found in Appendix A.

StyleCop analyzes C# source code to enforce a set of style and consistency rules. It can be run from inside of Visual Studio or integrated into an MSBuild project. StyleCop has also been integrated into many third-party development tools.

The StyleCop tool provides warnings that indicate style and consistency rule violations in C# code. These 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. The Standards Committee has reviewed all rules enforced by StyleCop, and decided on a subset that will be enforced in VA. Those rules are noted in the following sections, preceded by the StyleCop "SAxxxx" rule number and title. A customized `Settings.StyleCop` rule file for inclusion in your project/solution directory is available through TRM.

The complete list of rules that can be checked by Style Cop can be found at: http://www.stylecop.com/docs/StyleCop%20Rules.html.

Fortify Static Code Analyzer

Fortify Static Code Analyzer (SCA) scans source code, identifies root causes of software security vulnerabilities and correlates and prioritizes results. The VA Secure Code Review SOP requires Fortify SCA scans for compliance.

3) Naming Conventions

Naming conventions make programs more understandable by making them easier to read and ensuring consistency.

Choosing identifiers that conform to these guidelines improves the reusability of your code.

Casing Style Definitions

The following terms describe different ways to case identifiers.

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:

```
APPLICATIONPATH
```

Capitalization Rules for Identifiers

SA1300: Element Must Begin With Upper Case Letter

Cause

The name of a C# element does not begin with an upper-case letter.

Rule Description

A violation of this rule occurs when the names of certain types of elements do not begin with an upper-case letter. The following types of elements should use an upper-case letter as the first letter of the element name: *namespaces, classes, enums, structs, delegates, events, methods,* and *properties*.

In addition, any field which is public, internal, or marked with the const attribute should begin with an upper-case letter. Non-private readonly fields must also be named using an upper-case letter.

SA1303: Constant Field Names Must Begin With Upper Case Letter

Cause

The name of a constant C# field must begin with an upper-case letter.

Rule Description

A violation of this rule occurs when the name of a field marked with the *const* attribute does not begin with an upper-case letter.

SA1304: Non-Private Read-Only Fields Must Begin With Upper Case Letter

Cause

The name of a non-private read-only C# field must begin with an upper-case letter.

Rule Description

A violation of this rule occurs when the name of a readonly field which is not private does not begin with an upper-case letter. Non-private readonly fields must always start with an upper-case letter.

Other Standards, Not Checked By StyleCop

Casing for Identifiers

All identifiers except parameters should use Pascal-casing; parameters should be camel-cased.

Capitalization Rules for Acronyms

An acronym is a word that is formed from the letters of words in a term or phrase. For example, HTML is an acronym for Hypertext Markup Language. You should include acronyms in identifiers only when they are widely known and well understood.

Acronyms differ from abbreviations in that an abbreviation shortens a single word. For example, `ID` is an abbreviation for identifier. In general, library names should not use abbreviations. The exceptions are `ID` and `OK`. In Pascal-cased identifiers they should appear as `Id`, and `Ok`. If used as the first word in a camel-cased identifier, they should appear as `id` and `ok`, respectively.

Casing of acronyms depends on the length of the acronym and the casing of the identifier. The identifier casing rules take precedence over acronym casing rules.

For a camel-cased identifier, do not capitalize an acronym if it is the first word of the identifier (e.g. `ioChannel`). Otherwise, capitalize both letters of a two-character acronym (e.g. `mainIOChannel`), or only the first character for longer acronyms (e.g. `inputXml`).

For Pascal-cased identifiers, capitalize both letters of a two-character acronym, or only the first character for longer acronyms without regard for the position of the acronym in the identifier. (e.g. `IOChannel`, `MainIOChannel`, `InputXml`).

Capitalization Rules for Compound Words and Common Terms

Do not capitalize each word in so-called closed-form compound words. These are compound words written as a single word, such as "endpoint". For example, hashtable is a closed-form compound word that should be treated as a single word and cased accordingly. In Pascal case,

it is `Hashtable`; in camel case, it is `hashtable`. To determine whether a word is a closed-form compound word, check a current dictionary.

These are some common terms that are commonly mistaken as, but are not closed-form compound words.:

	Pascal casing	Camel casing
• Bit flag:	BitFlag	bitFlag
• File name:	FileName	fileName
• Log off:	LogOff	logOff
• Log off:	LogOn	logOn
• Sign on:	SignIn	signIn
• Sign out:	SignOut	signOut
• User name:	UserName	userName
• White space:	WhiteSpace	whiteSpace

General Naming Conventions

Choose easily readable identifier names, and favor readability over brevity. The property name `CanScrollHorizontally` is better than `ScrollableX` (an obscure reference to the X-axis).

Choose semantically meaningful names rather than language-specific keywords for type names. (`GetLength` is more meaningful than `GetInt`.)

Avoid using underscores, hyphens, or any other non-alphanumeric characters. 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.

Avoid abbreviations or contractions (e.g. use `OnButtonClick` rather than `OnBtnClick`)

Do not use any acronyms that are not widely accepted, and then only when necessary.

Avoid using identifiers that conflict with keywords of widely used programming languages. Though most keywords can be made to work as regular identifiers, doing so is confusing to read.

Names of Assemblies and DLLs

An assembly contains all or part of a reusable library and is contained in a single dynamic-link library (DLL). Assemblies and DLLs are the physical organization of a library (namespaces are a logical organization and should be factored independent of the assembly's organization).

Choose names for your assembly DLLs that suggest large chunks of functionality such as `System.Data`. Assembly and DLL names do not have to correspond to namespace names but it is reasonable to follow the namespace name when naming assemblies.

Name DLLs according to the pattern:

<Product Name>.<Functionality>[.<Component>].dll

For example: SampleProject.Configuration.DataAccess.dll

Additional levels may be added to <Functionality> or <Component> to subdivide large collections of functionality.

Names of Namespaces

The name chosen for a namespace should indicate the functionality made available by types in the namespace.

Use a stable, version-independent product name at the second level of a namespace name.

Use Pascal casing, and separate namespace components with periods.

Do not use generic type names such as Element, Node, Log, or Message, or names in the functionality or component namespaces. There is a very high probability these will cause type name conflicts in common scenarios.

Do not use the same name for a namespace and a type in that namespace. For example, do not use Debug for a namespace name and also provide a class named Debug in the same namespace. If you choose a namespace or type name that conflicts with an existing name, library users will have to qualify references to the affected items.

Names of Classes, Structures, and Interfaces

SA1302: Interface Names Must Begin With I

Cause

The name of a C# interface does not begin with the capital letter I.

Rule Description

A violation of this rule occurs when the name of an interface does not begin with the capital letter I. Interface names should always begin with I. For example, ICustomer.

Other Standards, Not Checked By StyleCop

Do not give class names a prefix (such as the letter C). Interfaces, which must begin with the letter I, are the exception to this rule.

Name classes, structures, interfaces, and value types with nouns, noun phrases, or occasionally, adjective phrases, using Pascal casing. In general, type names should be noun phrases, where the noun is the entity represented by the type.

Choose names that identify the entity from the developer's perspective; names should reflect usage scenarios.

Names of Methods

Choose method names that are verbs or verb phrases.

Typically methods act on data, so using a verb to describe the action of the method makes it easier for developers to understand what the method does. When defining the action performed by the method, be careful to select a name that provides clarity from the developer's perspective. Do not select a verb that describes how the method does what it does; in other words, do not use implementation details for your method name.

Names of Properties

Name properties using a noun, noun phrase, or an adjective. These are appropriate for properties because properties hold data.

Name Boolean properties with an affirmative phrase (`CanSeek` instead of `CantSeek`). Optionally, you can also prefix Boolean properties with `Is`, `Can`, or `Has`, but only where it adds value.

Names of Events

Name events with a verb or a verb phrase.

Give event names a concept of before and after, using the present and past tense. For example, a close event that is raised before a window is closed would be called `Closing` and one that is raised after the window is closed would be called `Closed`.

Name event handlers (delegates used as types of events) with the `EventHandler` suffix. Include two parameters named sender and `e` in event handler signatures. The sender parameter should be of type Object, and the `e` parameter should be an instance of or inherit from EventArgs.

Name event argument classes with the `EventArgs` suffix.

Names of Variables or Parameters

SA1308: Variable Names Must Not Be Prefixed

Cause

A field name in C# is prefixed with *m_* or *s_*.

Rule Description

A violation of this rule occurs when a field name is prefixed by *m_* or *s_*.

By default, StyleCop disallows the use of underscores, *m_*, etc., to mark local class fields, in favor of the 'this.' prefix. The advantage of using 'this.' is that it applies equally to all element types including methods, properties, etc., and not just fields, making all calls to class members instantly recognizable, regardless of which editor is being used to view the code. Another advantage is that it creates a quick, recognizable differentiation between instance members and static members, which will not be prefixed.

SA1309: Field Names Must Not Begin With Underscore

Cause

A field name in C# begins with an underscore.

Rule Description

A violation of this rule occurs when a field name begins with an underscore.

By default, StyleCop disallows the use of underscores, *m_*, etc., to mark local class fields, in favor of the 'this.' prefix. The advantage of using 'this.' is that it applies equally to all element types including methods, properties, etc., and not just fields, making all calls to class members instantly recognizable, regardless of which editor is being used to view the code. Another advantage is that it creates a quick, recognizable differentiation between instance members and static members, which will not be prefixed.

SA1310: Field Names Must Not Contain Underscore

Cause

A field name in C# contains an underscore.

Rule Description

A violation of this rule occurs when a field name contains an underscore.

Fields and variables should be named using descriptive, readable wording which describes the function of the field or variable. Typically, these names will be written using camel case, and should not use underscores. For example, use `customerPostCode` rather than `customer_post_code`.

SA1311: Static Read-Only Fields Must Begin With Upper Case Letter

Cause

The name of a static read-only field does not begin with an upper-case letter.

Rule Description

A violation of this rule occurs when the name of a static readonly field begins with a lower-case letter.

Other Standards, Not Checked By StyleCop

Name variables or parameters with nouns or noun phrases.

Use Camel casing in variable or parameter names, except in two-character acronyms.

4) Commenting and Documentation

Both General Comments and XML Documentation Comments are encouraged in VA code. General comments are comments which are delimited by `/*` and `*/`, or `//`. XML Documentation Comments are delimited with `///`.

General Comments

General comments are meant to aid developers in further understanding code and implementation decisions. General comments should contain only information that is relevant to reading and understanding the program. Discussion of nontrivial or unobvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code.

In general, avoid any comments that are likely to get out of date as the code evolves.

Temporary comments that are expected to be changed, or removed later, should be marked with special tokens so that they can easily be found.

Ideally, all temporary comments shall be removed by the time a program is ready to be moved to production.

Comments should not be enclosed in large boxes drawn with asterisks or other characters and should not include special characters such as form-feed and backspace.

Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format. A single-line comment should be preceded by a blank line.

Here's an example of a single-line comment in .NET

```
if (bar > 1) {

           bar--;


    // Do a triple-flip.

    ...

}
```

The // comment delimiter should not be used on consecutive full lines for text comments. However, it can be used in consecutive multiple lines for commenting out sections of code.

```
//if (bar > 1) {
//          bar--;
//
//    // Do a triple-flip.
//    ...
//}
```

Trailing Comments

Trailing comments are very short comments that appear on the same line as the code they describe. Trailing comments should be shifted far enough to the right in order to separate them from the statements. Multiple trailing comments contained in a section of code should be indented to the same tab setting.

The // comment delimiter can comment out a complete line or only a partial line.

```
if (foo > 1) {
            foo--;


    // Do a double-flip.
    ...
}
else {
    return false;          // foo <=1, no double-flip
}
```

Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of a file and/or before a method or class. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line. This sets it apart from the rest of the code.

For example:

```
/*
 * Here is a block comment.
 * It extends over several lines,
 * and uses the proper formatting.
 */
```

Special Tokens in Comments

In addition to general comments, Microsoft Visual Studio allows developers to place special tokens in comments to indicate areas where there is additional work to be completed or a known issue needs to be corrected.

These types of comments indicate that the code is not complete, or is not implemented in an optimal manner.

```
if (foo > 1) {
    // Do a double-flip.
    ...
}
else {
// HACK: Would be better if bar() would accept foo value
     return false;
}


// TODO: Implement triple-flip
```

These special token comments can be viewed in the Task List in the IDE. The default special tokens are `TODO`, `HACK`, and `UNDONE`, but custom tokens may be added

Released source code shall not contain special token comments showing incomplete work.

XML Documentation Comments

In Visual C#, you can create documentation for your code by including XML elements in special comment fields (indicated by triple slashes) in the source code directly before the code block to which the comments refer, for example:

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass{}
```

When you compile with the /doc option, the compiler will search for all XML tags in the source code and create an XML documentation file. To create the final documentation based on the compiler-generated file, you can create a custom tool or use a tool such as Sandcastle.

XML documentation comments can also provide valuable pop-up tips to those using libraries where methods and classes are appropriately documented.

For usage information, see How to: Use the XML Documentation Features in the Microsoft Developer Network C# Programming Guide.

5) Style and Design

Having style and design guidelines create a consistent look to the code, so that readers can focus on content, not layout. It also enables the readers to understand the code more quickly by making assumptions based on previous experience. Once a consistent look has been established, it will facilitate the copying, changing, and maintenance of the code.

All style and design elements are checked by StyleCop.

Style

SA1502: Element Must Not Be On Single Line

Cause

A C# element containing opening and closing curly brackets is written completely on a single line.

Rule Description

A violation of this rule occurs when an element that is wrapped in opening and closing curly brackets is written on a single line. For example:

```
public object Method() { return null; }
```

When StyleCop checks this code, a violation of this rule will occur because the entire method is written on one line. The method should be written across multiple lines, with the opening and closing curly brackets each on their own line, as follows:

```
public object Method()

{

      return null;

}
```

As an exception to this rule, accessors within properties, events, or indexers are allowed to be written all on a single line, as long as the accessor is short.

SA1504: All Accessor Must Be Multi-Line or Single Line

Cause

Within a C# property, indexer or event, at least one of the child accessors is written on a single line, and at least one of the child accessors is written across multiple lines.

Rule Description

A violation of this rule occurs when the accessors within a property, indexer or event are not consistently written on a single line or on multiple lines. This rule is intended to increase the readability of the code by requiring all of the accessors within an element to be formatted in the same way.

For example, the following property would generate a violation of this rule, because one accessor is written on a single line while the other accessor spans multiple lines.

```
public bool Enabled

{

get { return this.enabled; }

set

{

this.enabled = value;

}

}
```

The violation can be avoided by placing both accessors on a single line, or expanding both accessors across multiple lines:

```
public bool Enabled

{

get { return this.enabled; }

set { this.enabled = value; }

}

public bool Enabled

{

get

{

return this.enabled;

}

set

{

this.enabled = value;

}

}
```

SA1505: Opening Curly Brackets Must Not Be Followed By Blank Line

Cause

An opening curly bracket within a C# element, statement, or expression is followed by a blank line.

Rule Description

To improve the readability of the code, StyleCop requires blank lines in certain situations, and prohibits blank lines in other situations. This results in a consistent visual pattern across the code, which can improve recognition and readability of unfamiliar code.

A violation of this rule occurs when an opening curly bracket is followed by a blank line. For example:

```
public bool Enabled

{


      get

      {

            return this.enabled;

      }

}
```

The code above would generate two instances of this violation, since there are two places where opening curly brackets are followed by blank lines.

SA1506: Element Documentation Headers Must Not Be Followed By Blank Line

Cause

An element documentation header above a C# element is followed by a blank line.

Rule Description

To improve the readability of the code, StyleCop requires blank lines in certain situations, and prohibits blank lines in other situations. This results in a consistent visual pattern across the code, which can improve recognition and readability of unfamiliar code.

A violation of this rule occurs when the element documentation header above an element is followed by a blank line. For example:

```
/// <summary>

/// Gets a value indicating whether the control is enabled.

/// </summary>

public bool Enabled

{

get { return this.enabled; }

}
```

The code above would generate an instance of this violation, since the documentation header is followed by a blank line.

SA1507: Code Must Not Contain Multiple Blank Lines in a Row

Cause

The C# code contains multiple blank lines in a row.

Rule Description

To improve the readability of the code, StyleCop requires blank lines in certain situations, and prohibits blank lines in other situations. This results in a consistent visual pattern across the code, which can improve recognition and readability of unfamiliar code.

A violation of this rule occurs when the code contains more than one blank line in a row. For example:

```
public bool Enabled

{

get

{

Console.WriteLine("Getting the enabled flag.");




return this.enabled;

}

}
```

The code above would generate an instance of this violation, since it contains blank multiple lines in a row.

SA1508: Closing Curly Brackets Must Not Be Preceded By Blank Line

Cause

A closing curly bracket within a C# element, statement, or expression is preceded by a blank line.

Rule Description

To improve the readability of the code, StyleCop requires blank lines in certain situations, and prohibits blank lines in other situations. This results in a consistent visual pattern across the code, which can improve recognition and readability of unfamiliar code.

A violation of this rule occurs when a closing curly bracket is preceded by a blank line. For example:

```
public bool Enabled

{

get

{

return this.enabled;


}


}
```

The code above would generate two instances of this violation, since there are two places where closing curly brackets are preceded by blank lines.

SA1509: Opening Curly Brackets Must Not Be Preceded by Blank Line

Cause

An opening curly bracket within a C# element, statement, or expression is preceded by a blank line.

Rule Description

To improve the readability of the code, StyleCop requires blank lines in certain situations, and prohibits blank lines in other situations. This results in a consistent visual pattern across the code, which can improve recognition and readability of unfamiliar code.

A violation of this rule occurs when an opening curly bracket is preceded by a blank line. For example:

```
public bool Enabled
```

```
{

get


{

return this.enabled;

}

}
```

The code above would generate two instances of this violation, since there are two places where opening curly brackets are preceded by blank lines.

SA1510: Chained Statement Blocks Must Not Be Preceded By Blank Line

Cause

Chained C# statements are separated by a blank line.

Rule Description

To improve the readability of the code, StyleCop requires blank lines in certain situations, and prohibits blank lines in other situations. This results in a consistent visual pattern across the code, which can improve recognition and readability of unfamiliar code.

Some types of C# statements can only be used when chained to the bottom of another statement. Examples include catch and finally statements, which must always be chained to the bottom of a try-statement. Another example is an else-statement, which must always be chained to the bottom of an if-statement, or to another else-statement. These types of chained statements must not be separated by a blank line. For example:

```
try

{

this.SomeMethod();

}


catch (Exception ex)

{

Console.WriteLine(ex.ToString());

}
```

SA1511: While Do Footer Must Not Be Preceded By Blank Line

Cause

The while footer at the bottom of a do-while statement is separated from the statement by a blank line.

Rule Description

To improve the readability of the code, StyleCop requires blank lines in certain situations, and prohibits blank lines in other situations. This results in a consistent visual pattern across the code, which can improve recognition and readability of unfamiliar code.

A violation of this rule occurs when the while keyword at the bottom of a do-while statement is separated from the main part of the statement by one or more blank lines. For example:

```
do

{

Console.WriteLine("Loop forever");

}


while (true);
```

SA1517: Code Must Not Contain Blank Lines at Start of File

Cause

The code file has blank lines at the start.

Rule Description

To improve the layout of the code, StyleCop requires no blank lines at the start of files.

A violation of this rule occurs when one or more blank lines are at the start of the file.

SA1518: Code Must Not Contain Blank Lines at End Of File

Cause

The code file has blank lines at the end.

Rule Description

To improve the layout of the code, StyleCop requires no blank lines at the end of files.

A violation of this rule occurs when one or more blank lines are at the end of the file.

SA1000: Keywords Must Be Spaced Correctly

Cause

The spacing around a C# keyword is incorrect.

Rule Description

A violation of this rule occurs when the spacing around a keyword is incorrect.

The following C# keywords must always be followed by a single space: catch, fixed, for, foreach, from, group, if, in, into, join, let, lock, orderby, return, select, stackalloc, switch, throw, using, where, while, yield.

The following keywords must not be followed by any space: *checked, default, sizeof, typeof, unchecked*.

The *new* keyword should always be followed by a space, unless it is used to create a new array, in which case there should be no space between the *new* keyword and the opening array bracket.

SA1001: Commas Must Be Spaced Correctly

Cause

The spacing around a comma is incorrect, within a C# code file.

Rule Description

A violation of this rule occurs when the spacing around a comma is incorrect.

A comma should always be followed by a single space, unless it is the last character on the line, and a comma should never be preceded by any whitespace, unless it is the first character on the line.

SA1002: Semicolons Must Be Spaced Correctly

Cause

The spacing around a semicolon is incorrect, within a C# code file.

Rule Description

A violation of this rule occurs when the spacing around a semicolon is incorrect.

A semicolon should always be followed by a single space, unless it is the last character on the line, and a semicolon should never be preceded by any whitespace, unless it is the first character on the line.

SA1003: Symbols Must Be Spaced Correctly

Cause

The spacing around an operator symbol is incorrect, within a C# code file.

Rule Description

A violation of this rule occurs when the spacing around an operator symbol is incorrect.

The following types of operator symbols must be surrounded by a single space on either side: colons, arithmetic operators, assignment operators, conditional operators, logical operators, relational operators, shift operators, and lambda operators. For example:

```
int x = 4 + y;
```

In contrast, unary operators must be preceded by a single space, but must never be followed by any space. For example:

```
bool x = !value;
```

An exception is whenever the symbol is preceded or followed by a parenthesis or bracket, in which case there should be no space between the symbol and the bracket. For example:

```
if (!value)
{
}
```

SA1004: Documentation Lines Must Begin With Single Space

Cause

A line within a documentation header above a C# element does not begin with a single space.

Rule Description

A violation of this rule occurs when a line within a documentation header does not begin with a single space. For example:

```
///<summary>
///The summary text.
///</summary>
/// <param name="x">The document root.</param>
/// <param name="y">The Xml header token.</param>
private void Method1(int x, int y)
{
}
```

The header lines should begin with a single space after the three leading forward slashes:

```
/// <summary>
/// The summary text.
/// </summary>
/// <param name="x">The document root.</param>
/// <param name="y">The Xml header token.</param>
private void Method1(int x, int y)
{
}
```

SA1006: Preprocessor Keywords Must Not Be Preceded By Space

Cause

A C# preprocessor-type keyword is preceded by space.

Rule Description

A violation of this rule occurs when the preprocessor-type keyword in a preprocessor directive is preceded by space. For example:

```
# if Debug
```

There should not be any whitespace between the opening hash mark and the preprocessor-type keyword:

```
#if Debug
```

SA1007: Operator Keyword Must Be Followed By Space

Cause

The operator keyword within a C# operator overload method is not followed by any whitespace.

Rule Description

A violation of this rule occurs when the operator keyword within an operator overload method is not followed by any whitespace. The operator keyword should always be followed by a single space. For example:

```
public MyClass operator +(MyClass a, MyClass b)
{
}
```

SA1008: Opening Parenthesis Must Be Spaced Correctly

Cause

An opening parenthesis within a C# statement is not spaced correctly.

Rule Description

A violation of this rule occurs when the opening parenthesis within a statement is not spaced correctly. An opening parenthesis should not be preceded by any whitespace, unless it is the first character on the line, or it is preceded by certain C# keywords such as if, while, or for. In addition, an opening parenthesis is allowed to be preceded by whitespace when it follows an operator symbol within an expression.

An opening parenthesis should not be followed by whitespace, unless it is the last character on the line.

SA1009: Closing Parenthesis Must Be Spaced Correctly

Cause

A closing parenthesis within a C# statement is not spaced correctly.

Rule Description

A violation of this rule occurs when the closing parenthesis within a statement is not spaced correctly.

A closing parenthesis should never be preceded by whitespace. In most cases, a closing parenthesis should be followed by a single space, unless the closing parenthesis comes at the end of a cast, or the closing parenthesis is followed by certain types of operator symbols, such as positive signs, negative signs, and colons.

If the closing parenthesis is followed by whitespace, the next non-whitespace character must not be an opening or closing parenthesis or square bracket, or a semicolon or comma.

SA1010: Opening Square Brackets Must Be Spaced Correctly

Cause

An opening square bracket within a C# statement is not spaced correctly.

Rule Description

A violation of this rule occurs when an opening square bracket within a statement is preceded or followed by whitespace.

An opening square bracket must never be preceded by whitespace, unless it is the first character on the line, and an opening square must never be followed by whitespace, unless it is the last character on the line.

SA1011: Closing Square Brackets Must Be Spaced Correctly

Cause

A closing square bracket within a C# statement is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around a closing square bracket is not correct.

A closing square bracket must never be preceded by whitespace, unless it is the first character on the line.

A closing square bracket must be followed by whitespace, unless it is the last character on the line, it is followed by a closing bracket or an opening parenthesis, it is followed by a comma or semicolon, or it is followed by certain types of operator symbols.

SA1012: Opening Curly Brackets Must Be SpacedCorrectly

Cause

An opening curly bracket within a C# element is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around an opening curly bracket is not correct.

An opening curly bracket should always be preceded by a single space, unless it is the first character on the line, or unless it is preceded by an opening parenthesis, in which case there should be no space between the parenthesis and the curly bracket.

An opening curly bracket must always be followed by a single space, unless it is the last character on the line.

SA1013: Closing Curly Brackets Must Be SpacedCorrectly

Cause

A closing curly bracket within a C# element is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around a closing curly bracket is not correct.

A closing curly bracket should always be followed by a single space, unless it is the last character on the line, or unless it is followed by a closing parenthesis, a comma, or a semicolon.

A closing curly bracket must always be preceded by a single space, unless it is the first character on the line.

SA1014: Opening Generic Brackets Must Be Spaced Correctly

Cause

An opening generic bracket within a C# element is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around an opening generic bracket is not correct.

An opening generic bracket should never be preceded or followed by whitespace, unless the bracket is the first or last character on the line.

SA1015: Closing Generic Brackets Must Be Spaced Correctly

Cause

A closing generic bracket within a C# element is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around a closing generic bracket is not correct.

A closing generic bracket should never be preceded by whitespace, unless the bracket is the first character on the line. A closing generic bracket should be followed by an open parenthesis, a close parenthesis, a closing generic bracket, a nullable symbol, an end of line or a single whitespace (but not whitespace and an open parenthesis).

SA1016: Opening Attribute Brackets Must Be Spaced Correctly

Cause

An opening attribute bracket within a C# element is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around an opening attribute bracket is not correct.

An opening attribute bracket should never be followed by whitespace, unless the bracket is the last character on the line.

SA1017: Closing Attribute Brackets Must Be Spaced Correctly

Cause

A closing attribute bracket within a C# element is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around a closing attribute bracket is not correct.

A closing attribute bracket should never be preceded by whitespace, unless the bracket is the first character on the line.

SA1018: Nullable Type Symbols Must Not Be Preceded By Space

Cause

A nullable type symbol within a C# element is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around a nullable type symbol is not correct.

A nullable type symbol should never be preceded by whitespace, unless the symbol is the first character on the line.

SA1019: Member Access Symbols Must Be Spaced Correctly

Cause

The spacing around a member access symbol is incorrect, within a C# code file.

Rule Description

A violation of this rule occurs when the spacing around a member access symbol is incorrect. A member access symbol should not have whitespace on either side, unless it is the first character on the line.

SA1020: Increment Decrement Symbols Must Be Spaced Correctly

Cause

An increment or decrement symbol within a C# element is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around an increment or decrement symbol is not correct.

There should be no whitespace between the increment or decrement symbol and the item that is being incremented or decremented.

SA1021: Negative Signs Must Be Spaced Correctly

Cause

A negative sign within a C# element is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around a negative sign is not correct.

A negative sign should always be preceded by a single space, unless it comes after an opening square bracket, a parenthesis, or is the first character on the line.

A negative sign should never be followed by whitespace, and should never be the last character on a line.

SA1022: Positive Signs Must Be Spaced Correctly

Cause

A positive sign within a C# element is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around a positive sign is not correct.

A positive sign should always be preceded by a single space, unless it comes after an opening square bracket, a parenthesis, or is the first character on the line.

A positive sign should never be followed by whitespace, and should never be the last character on a line.

SA1023: Dereference And Access Of Symbols Must Be Spaced Correctly

Cause

A dereference symbol or an access-of symbol within a C# element is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around a dereference or access-of symbol is not correct.

The spacing around the symbol depends upon whether the symbol is used within a type declaration. If so, the symbol must always be followed by a single space, unless it is the last

character on the line, or is followed by an opening square bracket or a parenthesis. In addition, the symbol should not be preceded by whitespace, and should not be the first character on the line. An example of a properly spaced dereference symbol used within a type declaration is:

```
object* x = null;
```

When a dereference or access-of symbol is used outside of a type declaration, the opposite rule applies. In this case, the symbol must always be preceded by a single space, unless it is the first character on the line, or is preceded by an opening square bracket, a parenthesis or a symbol of the same type i.e. an equals. The symbol should not be followed by whitespace, and should not be the last character on the line. For example:

```
y = *x;
```

SA1024: Colons Must Be Spaced Correctly

Cause

A colon within a C# element is not spaced correctly.

Rule Description

A violation of this rule occurs when the spacing around a colon is not correct.

The spacing around a colon depends upon the type of colon and how it is used within the code. A colon appearing within an element declaration must always have a single space on either side, unless it is the first or last character on the line. For example all of the colons below follow this rule:

```
public class Class2<T> : Class1 where T : MyType

{

public Class2(int x) : base(x)

{

}

}
```

When the colon comes at the end of a label or case statement, it must always be followed by whitespace or be the last character on the line, but should never be preceded by whitespace. For example:

```
_label:

switch (x)

{

case 2:

return x;

}
```

Finally, when a colon is used within a conditional statement, it must always contain a single space on either side, unless the colon is the first or last character on the line. For example:

```
int x = y ? 2 : 3;
```

SA1025: Code Must Not Contain Multiple Whitespace In A Row

Cause

The code contains multiple whitespace characters in a row.

Rule Description

A violation of this rule occurs whenever the code contains multiple whitespace characters in a row, unless the characters come at the beginning or end of a line of code, following a comma or semicolon or preceding a symbol.

SA1026: Code Must Not Contain Space After New Keyword Implicitly Typed Array Allocation

Cause

An implicitly typed new array allocation within a C# code file is not spaced correctly.

Rule Description

A violation of this rule occurs whenever the code contains an implicitly typed new array allocation which is not spaced correctly. Within an implicitly typed new array allocation, there should not be any space between the new keyword and the opening array bracket. For example:

```
var a = new[] { 1, 10, 100, 1000 };
```

Design

SA1402: File May Only Contain a Single Class

Cause

A C# code file contains more than one unique class.

Rule Description

A violation of this rule occurs when a C# file contains more than one class. To increase long-term maintainability of the code-base, each class should be placed in its own file, and file names should reflect the name of the class within the file.

It is possible to place other supporting elements within the same file as the class, such as delegates, enums, etc., if they are related to the class.

It is also possible to place multiple parts of the same partial class within the same file.

SA1403: File May Only Contain a Single Namespace

Cause

A C# code file contains more than one namespace.

Rule Description

A violation of this rule occurs when a C# file contains more than one namespace. To increase long-term maintainability of the code-base, each file should contain at most one namespace.

SA1407: Arithmetic Expressions Must Declare Precedence

Cause

A C# statement contains a complex arithmetic expression which omits parenthesis around operators.

Rule Description

C# maintains a hierarchy of precedence for arithmetic operators. It is possible in C# to string multiple arithmetic operations together in one statement without wrapping any of the operations in parenthesis, in which case the compiler will automatically set the order and precedence of the operations based on these pre-established rules. For example:

```
int x = 5 + y * b / 6 % z - 2;
```

Although this code is legal, it is not highly readable or maintainable. In order to achieve full understanding of this code, the developer must know and understand the basic operator precedence rules in C#.

This rule is intended to increase the readability and maintainability of this type of code, and to reduce the risk of introducing bugs later, by forcing the developer to insert parenthesis to explicitly declare the operator precedence. The example below shows multiple arithmetic operations surrounded by parenthesis:

```
int x = 5 + (y * ((b / 6) % z)) - 2;
```

Inserting parenthesis makes the code more obvious and easy to understand, and removes the need for the reader to make assumptions about the code.

SA1408: Conditional Expressions Must Declare Precedence

Cause

A C# statement contains a complex conditional expression which omits parenthesis around operators.

Rule Description

C# maintains a hierarchy of precedence for conditional operators. It is possible in C# to string multiple conditional operations together in one statement without wrapping any of the operations in parenthesis, in which case the compiler will automatically set the order and precedence of the operations based on these pre-established rules. For example:

SA1206: Declaration Keywords Must Follow Order

Cause

The keywords within the declaration of an element do not follow a standard ordering scheme.

Rule Description

A violation of this rule occurs when the keywords within an element's declaration do not follow a standard ordering scheme.

Within an element declaration, keywords must appear in the following order:

- Access modifiers
- static
- All other keywords

Using a standard ordering scheme for element declaration keywords can make the code more readable by highlighting the access level of each element. This can help prevent elements from being given a higher access level than needed.

SA1207: Protected Must Come Before Internal

Cause

The keyword protected is positioned after the keyword internal within the declaration of a protected internal C# element.

Rule Description

A violation of this rule occurs when a protected internal element's access modifiers are written as *internal protected*. In reality, an element with the keywords *protected internal* will have the same access level as an element with the keywords *internal protected*. To make the code easier to read and more consistent, StyleCop standardizes the ordering of these keywords, so that a protected internal element will always be described as such, and never as internal protected. This can help to reduce confusion about whether these access levels are indeed the same.

SA1212: Property Accessors Must Follow Order

Cause

A get accessor appears after a set accessor within a property or indexer.

Rule Description

A violation of this rule occurs when a get accessor is placed after a set accessor within a property or indexer. To comply with this rule, the get accessor should appear before the set accessor.

For example, the following code would raise an instance of this violation:

```
public string Name

{

set { this.name = value; }

get { return this.name; }

}

The code below would not raise this violation:

public string Name

{

get { return this.name; }

set { this.name = value; }

}
```

SA1213: Event Accessors Must Follow Order

Cause

An add accessor appears after a remove accessor within an event.

Rule Description

A violation of this rule occurs when an add accessor is placed after a remove accessor within an event. To comply with this rule, the add accessor should appear before the remove accessor.

For example, the following code would raise an instance of this violation:

```
public event EventHandler NameChanged
{
remove { this.nameChanged -= value; }
add { this.nameChanged += value; }
}
```

The code below would not raise this violation:

```
public event EventHandler NameChanged
{
add { this.nameChanged += value; }
remove { this.nameChanged -= value; }
}
```

SA1100: Do Not Prefix Calls With Base

Cause

A call to a member from an inherited class begins with '`base.`', and the local class does not contain an override or implementation of the member.

Rule Description

A violation of this rule occurs whenever the code contains a call to a member from the base class prefixed with '`base.`', and there is no local implementation of the member. For example:

```
string name = base.JoinName("John", "Doe");
```

This rule is in place to prevent a potential source of bugs. Consider a base class which contains the following virtual method:

```
public virtual string JoinName(string first, string last)
{
}
```

Another class inherits from this base class but does not provide a local override of this method. Somewhere within this class, the base class method is called using `base.JoinName(...)`. This works as expected. At a later date, someone adds a local override of this method to the class:

```
public override string JoinName(string first, string last)

{

return "Bob";

}
```

At this point, the local call to `base.JoinName(...)` most likely introduces a bug into the code. This call will always call the base class method and will cause the local override to be ignored.

For this reason, calls to members from a base class should not begin with '`base.`', unless a local override is implemented, and the developer wants to specifically call the base class member. When there is no local override of the base class member, the call should be prefixed with '`this.`' rather than '`base.`'.

SA1101: Prefix Local Calls With "`this.`"

Cause

A call to an instance member of the local class or a base class is not prefixed with '`this.`', within a C# code file.

Rule Description

A violation of this rule occurs whenever the code contains a call to an instance member of the local class or a base class which is not prefixed with '`this.`'. An exception to this rule occurs when there is a local override of a base class member, and the code intends to call the base class member directly, bypassing the local override. In this case the call can be prefixed with '`base.`' rather than '`this.`'.

By default, StyleCop disallows the use of underscores or *m_* to mark local class fields, in favor of the '`this.`' prefix. The advantage of using '`this.`' is that it applies equally to all element types including methods, properties, etc., and not just fields, making all calls to class members instantly recognizable, regardless of which editor is being used to view the code. Another advantage is that it creates a quick, recognizable differentiation between instance members and static members, which are not prefixed.

A final advantage of using the '`this.`' prefix is that typing '`this.`' will cause Visual Studio to show the IntelliSense popup, making it quick and easy for the developer to choose the class member to call.

SA1106: Code Must Not Contain Empty Statements

Cause

The C# code contains an extra semicolon.

Rule Description

A violation of this rule occurs when the code contain an extra semicolon. Syntactically, this results in an extra, empty statement in the code.

SA1107: Code Must Not Contain Multiple Statements On One Line

Cause

The C# code contains more than one statement on a single line.

Rule Description

A violation of this rule occurs when the code contain more than one statement on the same line. Each statement must begin on a new line.

SA1110: Opening Parenthesis Must Be On Declaration Line

Cause

The opening parenthesis or bracket in a call to a C# method or indexer, or the declaration of a method or indexer, is not placed on the same line as the method or indexer name.

Rule Description

A violation of this rule occurs when the opening bracket of a method or indexer call or declaration is not placed on the same line as the method or indexer. The following examples show correct placement of the opening bracket:

```
public string JoinName(string first, string last)

{

return JoinStrings(

first, last);

}

public int this[int x]

{

get { return this.items[x]; }

}
```

SA1111: Closing Parenthesis Must Be On Line Of Last Parameter

Cause

The closing parenthesis or bracket in a call to a C# method or indexer, or the declaration of a method or indexer, is not placed on the same line as the last parameter.

Rule Description

A violation of this rule occurs when the closing bracket of a method or indexer call or declaration is not placed on the same line as the last parameter. The following examples show correct placement of the bracket:

```
public string JoinName(string first, string last)

{

string name = JoinStrings(

first,

last);

}

public int this[int x]

{

get { return this.items[x]; }

}
```

SA1112: Closing Parenthesis Must Be On Line Of Opening Parenthesis

Cause

The closing parenthesis or bracket in a call to a C# method or indexer, or the declaration of a method or indexer, is not placed on the same line as the opening bracket when the element does not take any parameters.

Rule Description

A violation of this rule occurs when a method or indexer does not take any parameters and the closing bracket of a call or declaration for the method or indexer is not placed on the same line as the opening bracket. The following example shows correct placement of the closing parenthesis:

```
public string GetName()

{

return this.name.Trim();

}
```

SA1113: Comma Must Be On Same Line As Previous Parameter

Cause

A comma between two parameters in a call to a C# method or indexer, or in the declaration of a method or indexer, is not placed on the same line as the previous parameter.

Rule Description

A violation of this rule occurs when a comma between two parameters to a method or indexer is not placed on the same line as the previous parameter. The following examples show correct placement of the comma:

```
public string JoinName(string first, string last)

{

string name = JoinStrings(

first,

last);

}

public int this[int x,

int y]

{

get { return this.items[x, y]; }

}
```

SA1114: Parameter List Must Follow Declaration

Cause

The start of the parameter list for a method or indexer call or declaration does not begin on the same line as the opening bracket or on the line after the opening bracket.

Rule Description

A violation of this rule occurs when there are one or more blank lines between the opening bracket and the start of the parameter list. For example:

```
public string JoinName(


string first, string last)

{

}
```

The parameter list must begin on the same line as the opening bracket, or on the next line. For example:

```
public string JoinName(string first, string last)

{

}

public string JoinName(

string first, string last)

{

}
```

SA1115: Parameter Must Follow Comma

Cause

A parameter within a C# method or indexer call or declaration does not begin on the same line as the previous parameter, or on the next line.

Rule Description

A violation of this rule occurs when there are one or more blank lines between a parameter and the previous parameter. For example:

```
public string JoinName(

string first,

string last)

{

}
```

The parameter must begin on the same line as the previous comma, or on the next line. For example:

```
public string JoinName(string first, string last)

{

}

public string JoinName(

string first,

string last)

{

}
```

6) VA Development Policies

Software Assurance (SwA) Verification with Fortify

Fortify Training

Fortify Training can be found at: OIS Software Assurance Site

Fortify Installation

Once training is complete, individuals must open a ticket with the National Service Desk (NSD) to request a Fortify license. Instructions are on the VA Software Assurance (SwA) Program Office (SAPO) FAQ SharePoint page. The license file will be emailed to you. A link to the software download instructions should also be included with the email (as of this writing, the instructions are here).

Installation will require administrative access to your local machine, but is otherwise straightforward. Be sure to install the plugin(s) for the version(s) of Visual Studio you have installed, and allow the software to update the security rule packs during the install. If you need to reinstall a Visual Studio plugin, or need a different version later, you can re-run the installer.

Rule packs can be updated from within the software, and must be current for any scan submitted to the SAPO for verification.

Fortify Static Code Analysis Scan

During Development

VA .NET developers should scan their code periodically during development to detect and address any potential issues early, and to make any code changes, if necessary, easier to implement. Once a scan is complete, each finding must be evaluated by the developer. Fortify will show an explanation of why it flagged the code, along with a snippet of the code itself. In the Visual Studio plugin, you can click to go directly to that code in the same way as with build errors. Fortify itself often has excellent suggestions on how to resolve any potential issues it detects in code. The process of evaluating Fortify findings is called “auditing.”

For example, a common finding is “Password Management: Password in Comment.” This finding will occur whenever the word ‘password’ is found in a comment. Fortify cannot determine if an actual password is shown in the comment, so the developer determines if there is an issue during the audit. Often, this finding will be set to “Not an Issue”, with a comment stating that only the word ‘password’ was used, and no actual password exists in the comment.

Another example is the finding of “Poor Error Handling: Empty Catch Block.” This is usually a valid issue, and the developer doing the audit should determine if the issue should be corrected, and another Fortify scan completed, or if a status such as “Bad Practice” should be assigned, along with a comment suggesting when the code should be rewritten to resolve the issue.

The VA .NET Improvement Group (a .NET developer community group) is working to create a reference of appropriate comments for a variety of Fortify findings. Suggestions for, and use of, the listing are encouraged.

You may audit any findings flagged by Fortify, and carry those entries through your development by loading the same audit (.fpr) file before auditing or performing another scan. So, you won't have to audit the same finding over again. Audit files worked on by multiple developers may be merged by using the "Merge Audit Projects…" option in the Visual Studio plugin, or the Audit Workbench.

During Release

As part of the Assessment and Authorization process to receive "Authority to Operate" on the VA network, a final Fortify scan must be performed, and the results submitted to the SAPO. This is what they call "Verification and Validation."

To satisfy this requirement, a developer must:

1. Ensure the project is registered with the SAPO. You'll need the Application ID for the rest of the process.
    a. You can search to determine if your project is registered on the SAPO Application Registrations SharePoint page.
    b. If you need to register your project, the instructions are on the SAPO "How to open an NSD ticket to register a VA application" SharePoint page.
2. Perform a final Fortify scan using the latest version of Fortify SCA and latest version of all rule packs.
    a. All errors/exceptions/warnings reported by Fortify during the scan(s) must be resolved. The scan must be "clean", with no errors during the scan.
    b. All findings reported by Fortify must be audited in the FPR file(s).
    c. All remaining critical and high findings must be "false positives".
    d. If a finding is a false positive, it must have been analyzed as "Not an Issue," with comments added to the FPR audit file stating the reason it is considered a false positive
    e.
3. Open an NSD ticket to request validation. Instructions, and a template for the email to NSD, are available on the SAPO SharePoint page.
    a. Provide the SAPO with the .fpr file from the final scan, and any other information requested by the SwA Program Office
    b. Schedule code review validation with the SAPO
4. Resolve any issues identified during validation and resubmit validation request.
    a. Note that resolution of issues may only require clarification of comments on "Not an Issue" findings. Detailed explanations are best to help the SAPO understand the logic behind your decision to mark a finding as "Not an Issue".
    b. If there is some risk in the finding, a properly documented Risk-Based Decision may satisfy the SAPO that the project's customer and development team are aware of the risk, and have other mitigation methods in place.

508 Compliance Standards

508 Compliance Training

Section 508 Support eLearning Courses can be found at:
http://vaww.section508.va.gov/Training.asp.

To ensure up to date information on how to test and correct applications to achieve 508 compliance take the (Testing Software Applications and Operating Systems for 508 Compliance eLearning class located at: http://www.section508.va.gov/support/sw/course.asp.

508 Compliance Testing Tools

A list of TRM approved 508 Compliance testing tools can be found at:
http://vaww.section508.va.gov/SECTION508/Accessibility_Testing_Tools.

During Development

508 compliance, via implementation of the VA 508 Compliance Checklists, is a mandatory non-functional requirement of all VA projects, and should be incorporated in design, development, and testing activities (see appendix_D for the 508 Standards Checklists). VA .NET developers should test their code periodically during development to detect and address any potential issues early and to make any code or design changes needed.

For example, a common finding is “Tab order must proceed logically and reflect the normal flow of use.” This finding will occur whenever the tab order does not flow from top left of the screen to bottom right. Another example is the finding of “Meaningful accessible names must be provided for all form elements.” This can be an issue on search buttons that only have an image or an ellipsis. Ensure all controls have a textual representation of their function.

Any findings of non-compliant 508 standards must be resolved prior to release. Refer to appendix_D for the checklists needed for compliance.

7) Sensitive Information Prohibited From Public Distribution

The table below is a "living" list of security sensitive information that is prohibited from inclusion in artifacts published *external* to the VA.

These data elements *must not be* included in VA artifacts for public distribution, such as the VA Software Document Library (VDL), Freedom of Information Act (FOIA), Open Source Electronic Health Record Agent (OSEHRA) and other open source organizations (Code-In-Flight), and any other non-VA external organization.

Security Sensitive Data Elements

DATA ELEMENT	DESCRIPTION/EXAMPLES
Internet Protocol (IP) Addresses	Internal VA server IP addresses. Sometimes it is difficult to create screen captures without also capturing security sensitive data (e.g., IP Addresses, Port Numbers, etc.). Technical writers need software to be able to remove sensitive information from screen capture while avoiding destroying the image or wipe out resolution.
Port Numbers	Internal VA server port numbers.
Server Uniform Resource Locators (URLs)	Internal VA server URL locations, such as Domain Server Names (DNS).
Artifact URLs	Links to internal artifact repositories (e.g., TSPR Project Notebooks, Product Support Anonymous Directories, SharePoint sites, ClearCase, Intranet Websites, etc.). Embedded or linked files.
Personally Identifiable Information (PII)	"The term 'PII,' as defined in OMB Memorandum M-07-1616 refers to information that can be used to distinguish or trace an individual's identity, either alone or when combined with other personal or identifying information that is linked or linkable to a specific individual. The definition of PII is not anchored to any single category of information or technology. Rather, it requires a case-by-case assessment of the specific risk that an individual can be identified. In performing this assessment, it is important for an agency to recognize that non-PII can become PII whenever additional information is made publicly available — in any medium and from any source — that, when combined with other available information, could be used to identify an individual." The following list contains examples of information that

DATA ELEMENT	DESCRIPTION/EXAMPLES
	may be considered PII1: Name, such as full name, maiden name, mother's maiden name, or alias. Personal identification number, such as social security number (SSN), passport number, driver's license number, taxpayer identification number, patient identification number, and financial account or credit card number. Address information, such as street address or email address. Asset information, such as Internet Protocol (IP) or Media Access Control (MAC) address or other host-specific persistent static identifier that consistently links to a particular person or small, well-defined group of people. Telephone numbers, including mobile, business, and personal numbers. Personal characteristics, including photographic image (especially of face or other distinguishing characteristic), x-rays, fingerprints, or other biometric image or template data (e.g., retina scan, voice signature, facial geometry). Information identifying personally owned property, such as vehicle registration number or title number nd related information. Information about an individual that is linked or linkable to one of the above (e.g., date of birth, place of birth, race, religion, weight, activities, geographical indicators, employment information, medical information, education information, financial information).
Contact Information	Names, such as document authors/editors developer or

DATA ELEMENT	DESCRIPTION/EXAMPLES
	other project team member names. Stakeholder and/or project team member information. Telephone Numbers. Email Addresses. Mail Groups. Signature Blocks. Physical Location Information.
Product Licenses or licensed content	For example, software installation license numbers.
Encryption Keys/Logic	For example, Kernel authentication encryption keys.
VistA Site IDs	For example, VA "Falling Waters" backup servers located in Martinsburg, WV.
Authentication Information	User IDs/Access Codes Passwords/Verify Codes
Copyrighted Files	Just for clarification, the redaction process does not physically redact copyrights on a bit-by-bit replacement of what goes out. If they are copyrighted components, the whole file is typically removed from the source prior to release.
Vendor Proprietary Data	This includes software where VA is not licensed to redistribute: Explanations of functionality in artifacts. Non-redistributable library components in source code. Source developed in VA, but not shareable outside of VA due to license constraints (e.g., VA modified Delphi VCL components).

8) Appendices

A) StyleCop Installation

Download the latest version of StyleCop here: http://stylecop.codeplex.com/releases/view/79972 and follow the installation instructions (at the time of writing this document, the current version of StyleCop is 4.7).

Include the `Settings.StyleCop` custom VA rule file from TRM in the top directory level of your solution or project. StyleCop will automatically use the custom rule file when checking your project/solution.

B) StyleCop Rules to be Enforced in VA

VA StyleCop Rule	Explanation
SA1500 Layout Rules (STYLE)	Rules which enforce code layout and line spacing. (link each table entry to appropriate StyleCop page) SA1502: ElementMustNotBeOnSingleLine SA1504: AllAccessorMustBeMultiLineOrSingleLine SA1505: OpeningCurlyBracketsMustNotBeFollowedByBlankLine SA1506: ElementDocumentationHeadersMustNotBeFollowedByBlankLine SA1507: CodeMustNotContainMultipleBlankLinesInARow SA1508: ClosingCurlyBracketsMustNotBePrecededByBlankLine SA1509: OpeningCurlyBracketsMustNotBePrecededByBlankLine SA1510: ChainedStatementBlocksMustNotBePrecededByBlankLine SA1511: WhileDoFooterMustNotBePrecededByBlankLine SA1517: CodeMustNotContainBlankLinesAtStartOfFile SA1518: CodeMustNotContainBlankLinesAtEndOfFile

SA1400 Maintainability Rules (DESIGN)	Rules which improve code maintainability. SA1402: FileMayOnlyContainASingleClass SA1403: FileMayOnlyContainASingleNamespace SA1407: ArithmeticExpressionsMustDeclarePrecedence SA1408: ConditionalExpressionsMustDeclarePrecendence
SA1300 Naming Rules (NAMING CONVENTIONS)	Rules which enforce naming requirements for members, types, and variables. SA1300: ElementMustBeginWithUpperCaseLetter SA1302: InterfaceNamesMustBeginWithI SA1303: ConstFieldNamesMustBeginWithUpperCaseLetter SA1304: NonPrivateReadonlyFieldsMustBeginWithUpperCaseLetter SA1308: VariableNamesMustNotBePrefixed SA1309: FieldNamesMustNotBeginWithUnderscore SA1310: FieldNamesMustNotContainUnderscore SA1311: StaticReadonlyFieldsMustBeginWithUpperCaseLetter
SA1200 Ordering Rules (DESIGN)	Rules which enforce a standard ordering scheme for code contents. SA1206: DeclarationKeywordsMustFollowOrder SA1207: ProtectedMustComeBeforeInternal SA1212: PropertyAccessorsMustFollowOrder SA1213: EventAccessorsMustFollowOrder
SA1100 Readability Rules (DESIGN)	Rules which ensure that the code is well-formatted and readable. SA1100: DoNotPrefixCallsWithBaseUnlessLocalImplementationExists SA1101: PrefixLocalCallsWithThis SA1106: CodeMustNotContainEmptyStatements SA1107: CodeMustNotContainMultipleStatementsOnOneLine SA1110: OpeningParenthesisMustBeOnDeclarationLine

	SA1111: ClosingParenthesisMustBeOnLineOfOpeningParenthesis SA1112: ClosingParenthesisMustBeOnLineOfOpeningParenthesis SA1113: CommaMustBeOnSameLineAsPreviousParameter SA1114: ParameterListMustFollowDeclaration SA1115: ParameterMustFollowComma
SA1000 Spacing Rules (STYLE)	Rules which enforce spacing requirements around keywords and symbols in the code. SA1000: KeywordsMustBeSpacedCorrectly SA1001: CommasMustBeSpacedCorrectly SA1002: SemicolonsMustBeSpacedCorrectly SA1003: SymbolsMustBeSpacedCorrectly SA1004: DocumentationLinesMustBeginWithSingleSpace SA1006: PreprocessorKeywordsMustNotBePrecededBySpace SA1007: OperatorKeywordMustBeFollowedBySpace SA1008: OpeningParenthesisMustBeSpacedCorrectly SA1009: ClosingParenthesisMustBeSpacedCorrectly SA1010: OpeningSquareBracketsMustBeSpacedCorrectly SA1011: ClosingSquareBracketsMustBeSpacedCorrectly SA1012: OpeningCurlyBracketsMustBeSpacedCorrectly SA1013: ClosingCurlyBracketsMustBeSpacedCorrectly SA1014: OpeningGenericBracketsMustBeSpacedCorrectly SA1015: ClosingGenericBracketsMustBeSpacedCorrectly SA1016: OpeningAttributeBracketsMustBeSpacedCorrectly SA1017: ClosingAttributeBracketsMustBeSpacedCorrectly SA1018: NullableTypeSymbolsMustNotBePrecededBySpace SA1019: MemberAccessSymbolsMustBeSpacedCorrectly SA1020: IncrementDecrementSymbolsMustBeSpacedCorrectly SA1021: NegativeSignsMustBeSpacedCorrectly SA1022: PositiveSignsMustBeSpacedCorrectly

	SA1023: DereferenceAndAccessOfSymbolsMustBeSpacedCorrectly SA1024: ColonsMustBeSpacedCorrectly SA1025: CodeMustNotContainMultipleWhitespaceInARow SA1026: CodeMustNotContainSpaceAfterNewKeywordInImplicitlyTypedArrayAllocation

C) StyleCop Rule Suppression

It is possible to suppress the reporting of rule violations by adding suppression attributes within the source code.

For more information about Code Analysis suppressions, see the following article: In Source Suppressions Overview.

StyleCop Rule Suppressions are registered in code using the `SuppressMessage` attribute. The `SuppressMessage` attribute is a conditional attribute, which is included in the IL metadata of your managed code assembly only if the CODE_ANALYSIS compilation symbol is defined at compile time.

The `SuppressMessage` attribute has the following format:

```
[SuppressMessage("Rule Category", "Rule Id", Justification =
"Justification")]
```

Where:

`Rule Category`	The StyleCop rule class in which the rule is defined. For example, *StyleCop.CSharp.DocumentationRules*
`Rule Id`	The identifier for the rule, using the format *shortname:longname*. For example, *SA1600:ElementsMustBeDocumented*
`Justification`	The text that is used to document the reason for suppressing the message.

The `SuppressMessage` attribute also takes the following optional parameters. These parameters are completely ignored by StyleCop and do not need to be filled in for StyleCop suppressions.

- `Message Id`
- `Scope`
- `Target`

`SuppressMessage` Usage:

StyleCop violations are suppressed at the level to which an instance of the `SuppressMessage` attribute is applied. The purpose of this is to tightly couple the suppression information to the code where the violation occurs.

For example, a StyleCop `SuppressMessage` attribute placed on a class will suppress the rule for all contents of the class. The same attribute placed on a method will only suppress the rule within the method.

Global Suppressions

StyleCop supports suppression at the namespace level:

```
[SuppressMessage("StyleCop.CSharp.DocumentationRules",
"SA1600:ElementsMustBeDocumented", Justification = "This is OK
here.")]

public namespace StyleCopExample

{

public class MyUndocumentedClass

{

public void MyUndocumentedMethod

{

}

}

}
```

D) Web Resources

XML Documentation Comments – Microsoft Developer Network (C# Programming Guide)

Guidelines for Names - Microsoft Developer Network

StyleCop Rules Documentation – All rules included in StyleCop

508 Standards Checklists – All of the checklists needed for 508 Compliance.