Estándar de Programación en C#.Net

Contenido

Introducción 7

Metodología 7

Herramientas 7

Convenciones 8

- Pascal Case 8
- Camel Case 8

Categorías 9

- Diseño 9
- Nomenclatura 9
- Legibilidad 9
- Orden 9
- Espacios 9
- Mantenimiento 9
- Documentación 9

Recomendaciones 10

Código Heredado 10

Reglas 11

- Diseño - Layout Rules 11
  - SA1500: CurlyBracketsForMultiLineStatementsMustNotShareLine 11
  - SA1501: StatementMustNotBeOnASingleLine 12
  - SA1502: ElementMustNotBeOnASingleLine 13
  - SA1503: CurlyBracketsMustNotBeOmitted 14
  - SA1504: AllAccessorsMustBeSingleLineOrMultiLine 15
  - SA1505: OpeningCurlyBracketsMustNotBeFollowedByBlankLine 16
  - SA1506: ElementDocumentationHeadersMustNotBeFollowedByBlankLine 17
  - SA1507: CodeMustNotContainMultipleBlankLinesInARow 18
  - SA1508: ClosingCurlyBracketsMustNotBePrecededByBlankLine 18
  - SA1509: OpeningCurlyBracketsMustNotBePrecededByBlankLine 19
  - SA1510: ChainedStatementBlocksMustNotBePrecededByBlankLine 20

SA1511: WhileDoFooterMustNotBePrecededByBlankLine 21
SA1512: SingleLineCommentsMustNotBeFollowedByBlankLine 22
SA1513: ClosingCurlyBracketsMustNotBeFollowedByBlankLine 23
SA1514: ElementDocumentationHeadersMustBePrecededByBlankLine 24
SA1515: SingleLineCommentsMustBePrecededByBlankLine 25
SA1516: ElementsMustBeSeparatedByBlankLine 27
Nomenclatura – Naming Rules. 28
SA1300: ElementMustBeginWithUpperCaseLetter 28
SA1301: ElementMustBeginWithLowerCaseLetter 28
SA1302: InterfaceNamesMustBeginWithI 29
SA1303: ConstFieldNamesMustBeginWithUpperCaseLetter 29
SA1304: NonPrivateReadonlyFieldsMustBeginWithUpperCaseLetter 30
SA1305: FieldNamesMustNotUseHungarianNotation 30
SA1306: FieldNamesMustBeginWithLowerCaseLetter 32
SA1307: AccessibleFieldsMustBeginWithUpperCaseLetter 33
SA1308: VariableNamesMustNotBePrefixed 34
SA1309: FieldNamesMustNotBeginWithUnderscore 34
SA1310: FieldNamesMustNotContainUnderscore 35
Legibilidad – Readability Rules 36
SA1100: DoNotPrefixCallsWithBaseUnlessLocalImplementationExists 36
SA1101: PrefixLocalCallsWithThis 37
SA1102: QueryClauseMustFollowPreviousClause 38
SA1103: QueryClausesMustBeOnSeparateLinesOrAllOnOneLine 39
SA1104: QueryClauseMustBeginOnNewLineWhenPreviousClauseSpansMultipleLines 40
SA1105: QueryClausesSpanningMultipleLinesMustBeginOnOwnLine 40
SA1106: CodeMustNotContainEmptyStatements 41
SA1107: CodeMustNotContainMultipleStatementsOnOneLine 42
SA1108: BlockStatementsMustNotContainEmbeddedComments 42
SA1109: BlockStatementsMustNotContainEmbeddedRegions 43
SA1110: OpeningParenthesisMustBeOnDeclarationLine 44
SA1111: ClosingParenthesisMustBeOnLineOfLastParameter 44
SA1112: ClosingParenthesisMustBeOnLineOfOpeningParenthesis 45

SA1113: CommaMustBeOnSameLineAsPreviousParameter 46
SA1114: ParameterListMustFollowDeclaration 47
SA1115: ParameterMustFollowComma 48
SA1116: SplitParametersMustStartOnLineAfterDeclaration 48
SA1117: ParametersMustBeOnSameLineOrSeparateLines 49
SA1118: ParametersMustNotSpanMultipleLines 50
SA1119: StatementMustNotUseUnnecessaryParenthesis 52
SA1120: CommentsMustContainText 52
SA1121: UseBuiltInTypeAlias 53
SA1122: UseStringEmptyForEmptyStrings 54
SA1123: DoNotPlaceRegionsWithinElements 55
SA1124: DoNotUseRegions 55
Orden – Ordering Rules 56
SA1200: UsingDirectivesMustBePlacedWithinNamespace 56
SA1201: ElementsMustAppearInTheCorrectOrder 59
SA1202: ElementsMustBeOrderedByAccess 62
SA1203: ConstantsMustAppearBeforeFields 63
SA1204: StaticElementsMustAppearBeforeInstanceElements 63
SA1207: ProtectedMustComeBeforeInternal 64
SA1208: SystemUsingDirectivesMustBePlacedBeforeOtherUsingDirectives 65
SA1209: UsingAliasDirectivesMustBePlacedAfterOtherUsingDirectives 65
SA1210: UsingDirectivesMustBeOrderedAlphabeticallyByNamespace 66
SA1211: UsingAliasDirectivesMustBeOrderedAlphabeticallyByAliasName 66
SA1212: PropertyAccessorsMustFollowOrder 67
SA1213: EventAccessorsMustFollowOrder 68
Espacios – Spacing Rules 68
SA1000: KeywordsMustBeSpacedCorrectly 68
SA1001: CommasMustBeSpaceCorrectly 69
SA1002: SemicolonsMustBeSpaceCorrectly 70
SA1003: SymbolsMustBeSpaceCorrectly 70
SA1004: DocumentationLinesMustBeginWithSingleSpace 71
SA1005: SingleLineCommentsMustBeginWithSingleSpace 72

SA1006: PreprocessorKeywordsMustNotBePrecededBySpace 73
SA1007: OperatorKeywordMustBeFollowedBySpace 74
SA1008: OpeningParenthesisMustBeSpacedCorrectly 74
SA1009: ClosingParenthesisMustBeSpacedCorrectly 75
SA1010: OpeningSquareBracketsMustBeSpacedCorrectly 75
SA1011: ClosingSquareBracketsMustBeSpacedCorrectly 76
SA1012: OpeningCurlyBracketsMustBeSpacedCorrectly 77
SA1013: ClosingCurlyBracketsMustBeSpacedCorrectly 77
SA1014: OpeningGenericBracketsMustBeSpacedCorrectly 78
SA1015: ClosingGenericBracketsMustBeSpacedCorrectly 78
SA1016: OpeningAttributeBracketsMustBeSpacedCorrectly 79
SA1017: ClosingAttributeBracketsMustBeSpacedCorrectly 79
SA1018: NullableTypeSymbolsMustNotBePrecededBySpace 80
SA1019: MemberAccessSymbolsMustBeSpacedCorrectly 80
SA1020: IncrementDecrementSymbolsMustBeSpacedCorrectly 81
SA1021: NegativeSignsMustBeSpacedCorrectly 81
SA1022: PositiveSignsMustBeSpacedCorrectly 82
SA1023: DereferenceAndAccessOfMustBeSpacedCorrectly 82
SA1024: ColonsMustBeSpacedCorrectly 83
SA1025: CodeMustNotContainMultipleWhitespaceInARow 84
SA1026: CodeMustNotContainSpaceAfterNewKeywordInImplicitlyTypedArrayAllocation 85
SA1027: TabsMustNotBeUsed 85
Mantenimiento – Maintainability Rules 86
SA1400: AccessModifierMustBeDeclared 86
SA1401: FieldsMustBePrivate 87
SA1402: FileMayOnlyContainASingleClass 87
SA1403: FileMayOnlyContainASingleNamespace 88
SA1404: CodeAnalysisSuppressionMustHaveJustification 88
SA1405: DebugAssertMustProvideMessageText 89
SA1406: DebugFailMustProvideMessageText 89
SA1407: ArithmeticExpressionsMustDeclarePrecedence 90
SA1408: ConditionalExpressionsMustDeclarePrecedence 91

SA1409: RemoveUnnecessaryCode 92
SA1410: RemoveDelegateParenthesisWhenPossible 93
Documentación – Documentation Rules 94
SA1600: ElementsMustBeDocumented 94
SA1601: PartialElementsMustBeDocumented 95
SA1602: EnumerationItemsMustBeDocumented 97
SA1603: DocumentationMustContainValidXml 98
SA1604: ElementDocumentationMustHaveSummary 99
SA1605: PartialElementDocumentationMustHaveSummary 99
SA1606: ElementDocumentationMustHaveSummaryText 101
SA1607: PartialElementDocumentationMustHaveSummaryText 102
SA1608: ElementDocumentationMustNotHaveDefaultSummary 104
SA1609: PropertyDocumentationMustHaveValue 105
SA1610: PropertyDocumentationMustHaveValueText 106
SA1611: ElementParametersMustBeDocumented 107
SA1612: ElementParameterDocumentationMustMatchElementParameters 108
SA1613: ElementParameterDocumentationMustDeclareParameterName 109
SA1614: ElementParameterDocumentationMustHaveText 110
SA1615: ElementReturnValueMustBeDocumented 111
SA1616: ElementReturnValueDocumentationMustHaveText 112
SA1617: VoidReturnValueMustNotBeDocumented 112
SA1618: GenericTypeParametersMustBeDocumented 113
SA1619: GenericTypeParametersMustBeDocumentedPartialClass 114
SA1620: GenericTypeParameterDocumentationMustMatchTypeParameters 116
SA1621: GenericTypeParameterDocumentationMustDeclareParameterName 117
SA1622: GenericTypeParameterDocumentationMustHaveText 118
SA1623: PropertySummaryDocumentationMustMatchAccessors 119
SA1624: PropertySummaryDocumentationMustOmitSetAccessorWithRestrictedAccess 123
SA1625: ElementDocumentationMustNotBeCopiedAndPasted 127
SA1626: SingleLineCommentsMustNotUseDocumentationStyleSlashes 128
SA1627: DocumentationTextMustNotBeEmpty 129
SA1628: DocumentationTextMustBeginWithACapitalLetter 129

SA1629: DocumentationTextMustEndWithAPeriod 130
SA1630: DocumentationTextMustContainWhitespace 131
SA1631: DocumentationTextMustMeetCharacterPercentage 132
SA1632: DocumentationTextMustMeetMinimumCharacterLength 133
SA1633: FileMustHaveHeader 133
SA1634: FileHeaderMustShowCopyright 135
SA1635: FileHeaderMustHaveCopyrightText 136
SA1636: FileHeaderCopyrightTextMustMatch 136
SA1637: FileHeaderMustContainFileName 137
SA1638: FileHeaderFileNameDocumentationMustMatchFileName 138
SA1639: FileHeaderMustHaveSummary 139
SA1640: FileHeaderMustHaveValidCompanyText 140
SA1641: FileHeaderCompanyNameTextMustMatch 140
SA1642: ConstructorSummaryDocumentationMustBeginWithStandardText 142
SA1643: DestructorSummaryDocumentationMustBeginWithStandardText 144
SA1644: DocumentationHeadersMustNotContainBlankLines 145
SA1645: IncludedDocumentationFileDoesNotExist 146
SA1646: IncludedDocumentationXPathDoesNotExist 147
SA1647: IncludeNodeDoesNotContainValidFileAndPath 148

Introducción

El presente documento tiene como objetivo definir un estándar de estilo de programación para el lenguaje C#.Net para las aplicaciones realizadas por el equipo de desarrolladores. La necesidad principal de definir un estándar, nace de la realidad de generar desarrollos de software modularizados, por grupos de personas que no se encuentran ubicados geográficamente en la misma oficina, ni en la misma ciudad.

Además, los estándares brindan ventajas alternativas que enriquecen la calidad del servicio brindado, mejoran la calidad del software y facilitan notablemente el mantenimiento

Metodología

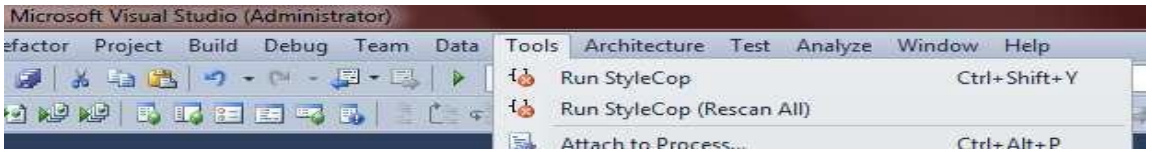
Existen diversos estándares de estilos de codificación para los lenguajes de programación, incluso muchas empresas inventan un estilo propio. Por suerte en C# los diferentes estándares existentes no difieren mucho entre si, incluso el Visual Studio realiza la indentación automática, con lo cual es fácil acostumbrarse al estándar. En nuestro caso optamos por el estándar utilizado por Microsoft, para que sea natural para los programadores que ya vienen desarrollando en este lenguaje.

Herramientas

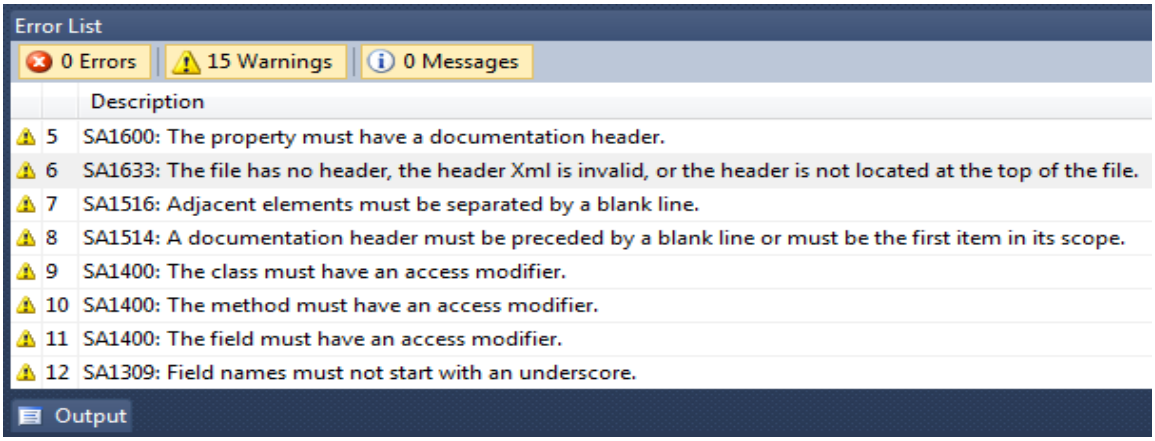
Para facilitar la tarea del programador, y que no tenga que estar leyendo este manual frecuentemente, hasta acostumbrarse al estándar, existe un herramienta muy potente que se integra al Visual Studio, la cual valida errores de codificación al momento de compilar la aplicación.

La aplicación a utilizar será Microsoft StyleCop, que al momento de redacción de este documento va por la versión 4.4. StyleCop se integra fácilmente a Visual Studio 2008 y 2010. Por defecto trae configuradas todos los estándares posibles, aunque algunos se pueden deshabilitar programáticamente o mediante el archivo del proyecto.

Una vez instalada, se verá una nueva opción dentro del menú “Tools” del Visual Studio, denominada “Run StyleCop”, desde la cual se puede ejecutar el análisis del código:



El resultado será un listado de advertencias en la misma ventana donde se muestran los errores de compilación. El analizador dejará compilar y ejecutar el código fuente, pero la recomendación es acostumbrarse a no dejar ninguna advertencia de StyleCop:



Convenciones

En las declaraciones donde se suelen utilizan palabras compuestas, es decir, dos o más palabras que deben unirse en una sola palabra, se utilizan dos convenciones posibles de estilos de escritura en lo que respecta a letras mayúsculas y minúsculas. Según el tipo de declaración que se esté realizando (Variables, Propiedades, Clases, etc.) se utiliza una de las dos convenciones posibles, según el estándar.

Pascal Case

La primera letra de cada palabra comienza con mayúscula y el resto de las letras en minúsculas. Ej: MiClase. Su uso más común es para declarar propiedades, métodos y clases.

Camel Case

La primera letra de cada palabra comienza con mayúscula y el resto de las letras en minúsculas, excepto la primera letra que va siempre en minúscula. Ej: miVariable. Su uso más común es para declarar variables, ya sean locales, miembros de clases o argumentos de métodos.

Categorías

El estándar de codificación se divide en varias categorías, que contienen las distintas reglas a aplicar sobre el código fuente.

Diseño

Estas reglas controlan la utilización de llaves, paréntesis, múltiples sentencias en la misma línea de código, separación entre sentencias, etc.

Nomenclatura

Fuerza la utilización de mayúsculas y minúsculas de manera correcta, dependiendo de lo que se está declarando: Variables, constantes, clases, métodos, etc.

Legibilidad

Verifica la mala utilización de prefijos, también la forma de declarar argumentos de métodos cuando son muy extensos y se necesita ubicarlos en diferentes líneas de código.

Orden

Controla el orden de declaración dentro de una clase, por ejemplo que se declaren primero los atributos privados y luego las propiedades, así como también el orden de los accesos get y set de las propiedades.

Espacios

Controla que todas las palabras estén separadas con la cantidad de espacios correcta, incluso la forma de utilización de las comas.

Mantenimiento

Controla la utilización de paréntesis para indicar precedencia de operadores o la no utilización de paréntesis innecesarios, así como declaraciones de código que pueden generar confusiones.

Documentación

Las reglas de documentación, además de forzar la documentación de distintas partes del código impone la forma de documentar la descripción de ciertos elementos.

Recomendaciones

Se recomiendo seguir todas las reglas a no ser que existe una buena razón para no hacerlo. Tal vez las reglas de particulares de documentación, son demasiado engorrosas ya que con una documentación básica en la mayoría de los casos es suficiente. Las reglas pueden desactivarse mediante código o con una configuración.

Código Heredado

Es muy fácil implementar este estándar en un proyecto nuevo en el cual que se parte desde cero. Para proyectos avanzados o terminados, se podría correr un análisis que informe la cantidad de inconsistencias y así determinar el tiempo que demandaría la corrección del código para apegarse al estándar.

Reglas

A continuación se detallan todas las reglas que controla el StyleCop separadas en las categorías correspondientes. Estas reglas fueron extraídas directamente de la documentación de la aplicación, motivo por el cual están redactadas en inglés. Cada regla tiene un código asociado, que nos servirá para buscar de qué se trata cuando no pasa la validación del StyleCop.

Diseño - Layout Rules

SA1500: CurlyBracketsForMultiLineStatementsMustNotShareLine

Cause

The opening or closing curly bracket within a C# statement, element, or expression is not placed on its own line.

Rule Description

A violation of this rule occurs when the opening or closing curly bracket within a statement, element, or expression is not placed on its own line. For example:

```
public object Method()
{
    lock (this) {
        return this.value;
    }
}
```

When StyleCop checks this code, a violation of this rule will occur because the opening curly bracket of the lock statement is placed on the same line as the lock keyword, rather than being placed on its own line, as follows:

```
public object Method()
{
    lock (this)
    {
        return this.value;
    }
```

```
}
```

A violation will also occur if the closing curly bracket shares a line with other code. For example:

```
public object Method()
{
    lock (this)
    {
        return this.value; }
}
```

How to Fix Violations

To fix a violation of this rule, ensure that both the opening and closing curly brackets are placed on their own line, and do not share the line with any other code, other than comments.

SA1501: StatementMustNotBeOnASingleLine

Cause

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

Rule Description

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

```
public object Method()
{
    lock (this) { return this.value; }
}
```

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

```
public object Method()
{
```

```
        lock (this)
        {
            return this.value;
        }
    }
```

How to Fix Violations

To fix a violation of this rule, rewrite the statement so that it expands across multiple lines.

SA1502: ElementMustNotBeOnASingleLine

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.

How to Fix Violations

To fix a violation of this rule, rewrite the element so that it expands across multiple lines.

SA1503: CurlyBracketsMustNotBeOmitted

Cause

The opening and closing curly brackets for a C# statement have been omitted.

Rule Description

A violation of this rule occurs when the opening and closing curly brackets for a statement have been omitted. In C#, some types of statements may optionally include curly brackets. Examples include if, while, and for statements. For example, an if-statement may be written without curly brackets:

```
if (true)
    return this.value;
```

Although this is legal in C#, StyleCop always requires the curly brackets to be present, to increase the readability and maintainability of the code.

When the curly brackets are omitted, it is possible to introduce an error in the code by inserting an additional statement beneath the if-statement. For example:

```
if (true)
    this.value = 2;
    return this.value;
```

Glancing at this code, it appears as if both the assignment statement and the return statement are children of the if-statement. In fact, this is not true. Only the assignment statement is a child of the if-statement, and the return statement will always execute regardless of the outcome of the if-statement.

StyleCop always requires the opening and closing curly brackets to be present, to prevent these kinds of errors:

```
if (true)
```

```
{
    this.value = 2;
    return this.value;
}
```

How to Fix Violations

To fix a violation of this rule, wrap the body of the statement in curly brackets.

SA1504: AllAccessorsMustBeSingleLineOrMultiLine

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 snaps 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;
    }
}
```

How to Fix Violations

To fix a violation of this rule, write each accessor on a single line if the accessors are short, or expand both accessors across multiple lines if the accessors are longer.

SA1505: OpeningCurlyBracketsMustNotBeFollowedByBlankLine

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.

How to Fix Violations

To fix a violation of this rule, remove the blank line following the opening curly bracket.

SA1506: ElementDocumentationHeadersMustNotBeFollowedByBlankLine

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.

How to Fix Violations

To fix a violation of this rule, remove the blank line following the documentation header.

SA1507: CodeMustNotContainMultipleBlankLinesInARow

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.

How to Fix Violations

To fix a violation of this rule, remove the extra blank lines.

SA1508: ClosingCurlyBracketsMustNotBePrecededByBlankLine

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.

How to Fix Violations

To fix a violation of this rule, remove the blank line preceding the closing curly bracket.

SA1509: OpeningCurlyBracketsMustNotBePrecededByBlankLine

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.

How to Fix Violations

To fix a violation of this rule, remove the blank line preceding the opening curly bracket.

SA1510: ChainedStatementBlocksMustNotBePrecededByBlankLine

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:

```
tr
y
{    this.SomeMethod();

}

catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```

How to Fix Violations

To fix a violation of this rule, remove any blank lines between the chained statements.

SA1511: WhileDoFooterMustNotBePrecededByBlankLine

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);
```

How to Fix Violations

To fix a violation of this rule, remove any blank lines before the while keyword.

SA1512: SingleLineCommentsMustNotBeFollowedByBlankLine

Cause

A single-line comment within C# code 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 a single-line comment is followed by a blank line. For example:

```
public bool Enabled
{
    get
    {
        // Return the value of the 'enabled' field.

        return this.enabled;
    }
}
```

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

It is allowed to place a blank line in between two blocks of single-line comments. For example:

```
public bool Enabled
{
```

```
        get
        {
            // This is a sample comment which doesn't really say anything.
            // This is another part of the comment.

            // There is a blank line above this comment but that is ok.
            return this.enabled;
        }
    }
```

If the comment is being used to comment out a line of code, place four forward slashes at the beginning of the comment, rather than two. This will cause StyleCop to ignore this violation. For example:

```
    public bool Enabled
    {
        get
        {
            ////return false;

            return this.enabled;
        }
    }
```

How to Fix Violations

To fix a violation of this rule, remove the blank line following the single-line comment.

SA1513: ClosingCurlyBracketsMustNotBeFollowedByBlankLine

Cause

A closing 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 a closing curly bracket is followed by a blank line. For example:

```
public bool Enabled
{
    get
    {
        return this.enabled;
    }

}
```

The code above would generate one instance of this violation, since there is one place where a closing curly bracket is followed by a blank line.

How to Fix Violations

To fix a violation of this rule, remove the blank line following the closing curly bracket.

SA1514: ElementDocumentationHeadersMustBePrecededByBlankLine

Cause

An element documentation header above a C# element is not 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 the element documentation header above an element is not preceded by a blank line. For example:

```
public bool Visible
{
    get { return this.visible; }
}
/// <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 not preceded by a blank line.

An exception to this rule occurs when the documentation header is the first item within its scope. In this case, the header should not be preceded by a blank line. For example:

```
public class Class1
{
    /// <summary>
    /// Gets a value indicating whether the control is enabled.
    /// </summary>
    public bool Enabled
    {
        get { return this.enabled; }
    }
}
```

In the code above, the header is the first item within its scope, and thus it should not be preceded by a blank line.

How to Fix Violations

To fix a violation of this rule, add a blank line above the documentation header.

SA1515: SingleLineCommentsMustBePrecededByBlankLine

Cause

A single-line comment within C# code is not 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 single-line comment is not preceded by a blank line. For example:

```
public bool Enabled
{
    get
    {
        Console.WriteLine("Getting the enabled flag.");
        // Return the value of the 'enabled' field.
        return this.enabled;
    }
}
```

The code above would generate an instance of this violation, since the single-line comment is not preceded by a blank line.

An exception to this rule occurs when the single-line comment is the first item within its scope. In this case, the comment should not be preceded by a blank line. For example:

```
public bool Enabled
{
    get
    {
        // Return the value of the 'enabled' field.
        return this.enabled;
    }
}
```

In the code above, the comment is the first item within its scope, and thus it should not be preceded by a blank line.

If the comment is being used to comment out a line of code, begin the comment with four forward slashes rather than two. This will cause StyleCop to ignore this violation. For example:

```
public bool Enabled
{
    get
    {
        Console.WriteLine("Getting the enabled flag.");
        ////return false;
        return this.enabled;
    }
```

```
}
```

How to Fix Violations

To fix a violation of this rule, add a blank line above the comment.

SA1516: ElementsMustBeSeparatedByBlankLine

Cause

Adjacent C# elements are not 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.

A violation of this rule occurs when two adjacent element are not separated by a blank line. For example:

```
public void Method1()
{
}
public bool Property
{
    get { return true; }
}
```

In the example above, the method and property are not separated by a blank line, so a violation of this rule would occur.

How to Fix Violations

To fix a violation of this rule, add a blank line between the adjacent elements.

Nomenclatura – Naming Rules

SA1300: ElementMustBeginWithUpperCaseLetter

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.

If the field or variable name is intended to match the name of an item associated with Win32 or COM, and thus needs to begin with a lower-case letter, place the field or variable within a special NativeMethods class. A NativeMethods class is any class which contains a name ending in NativeMethods, and is intended as a placeholder for Win32 or COM wrappers. StyleCop will ignore this violation if the item is placed within a NativeMethods class.

How to Fix Violations

To fix a violation of this rule, change the name of the element so that it begins with an upper-case letter, or place the item within a NativeMethods class if appropriate.

SA1301: ElementMustBeginWithLowerCaseLetter

Cause

There are currently no situations in which this rule will fire.

SA1302: InterfaceNamesMustBeginWithI

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.

If the field or variable name is intended to match the name of an item associated with Win32 or COM, and thus cannot begin with the letter I, place the field or variable within a special NativeMethods class. A NativeMethods class is any class which contains a name ending in NativeMethods, and is intended as a placeholder for Win32 or COM wrappers. StyleCop will ignore this violation if the item is placed within a NativeMethods class.

How to Fix Violations

To fix a violation of this rule, add the capital letter I to the front of the interface name, or place the item within a NativeMethods class if appropriate.

SA1303: ConstFieldNamesMustBeginWithUpperCaseLetter

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.

If the field or variable name is intended to match the name of an item associated with Win32 or COM, and thus needs to begin with a lower-case letter, place the field or variable within a special NativeMethods class. A NativeMethods class is any class which contains a name ending in NativeMethods, and is intended

as a placeholder for Win32 or COM wrappers. StyleCop will ignore this violation if the item is placed within a NativeMethods class.

How to Fix Violations

To fix a violation of this rule, change the name of the constant field so that it begins with an upper-case letter, or place the item within a NativeMethods class if appropriate.

SA1304: NonPrivateReadonlyFieldsMustBeginWithUpperCaseLetter

Cause

The name of a non-private readonly C# field must being 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.

If the field or variable name is intended to match the name of an item associated with Win32 or COM, and thus needs to begin with a lower-case letter, place the field or variable within a special NativeMethods class. A NativeMethods class is any class which contains a name ending in NativeMethods, and is intended as a placeholder for Win32 or COM wrappers. StyleCop will ignore this violation if the item is placed within a NativeMethods class.

How to Fix Violations

To fix a violation of this rule, change the name of the readonly field so that it begins with an upper-case letter, make the field private, or place the item within a NativeMethods class if appropriate.

SA1305: FieldNamesMustNotUseHungarianNotation

Cause

The name of a field or variable in C# uses Hungarian notation.

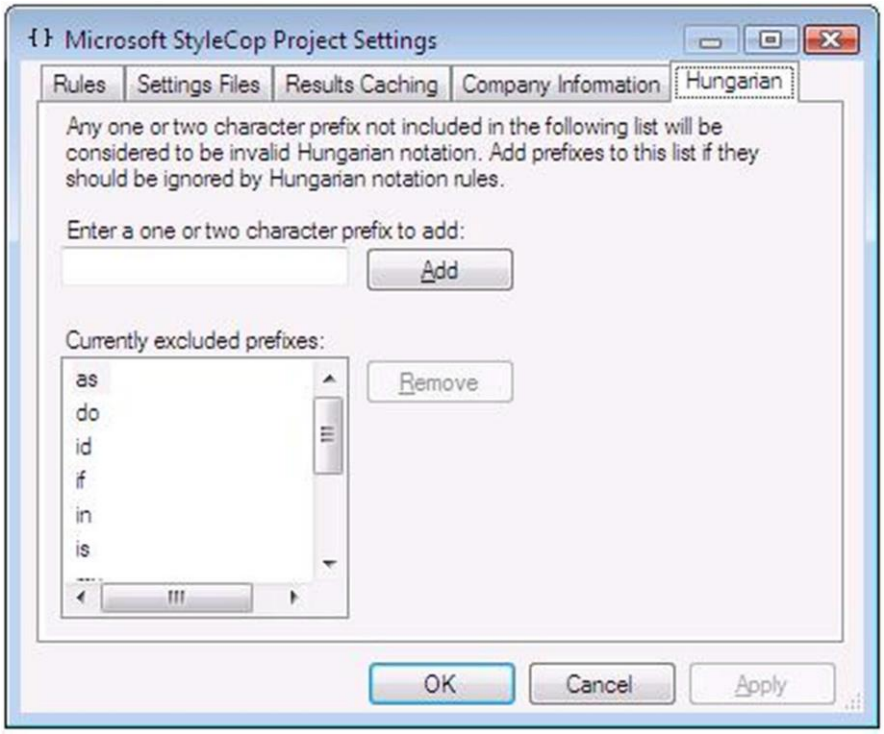
Rule Description

A violation of this rule occurs when Hungarian notation is used in the naming of fields and variables. The use of Hungarian notation has become widespread in C++ code, but the trend in C# is to use longer, more descriptive names for variables, which are not based on the type of the variable but which instead describe what the variable is used for.

In addition, modern code editors such as Visual Studio make it easy to identify type information for a variable or field, typically by hovering the mouse cursor over the variable name. This reduces the need for Hungarian notation.

StyleCop assumes that any variable name that begins with one or two lower-case letters followed by an upper-case letter is making use of Hungarian notation, and will flag a violation of this rule in each case. It is possible to declare certain prefixes as legal, in which case they will be ignored. For example, a variable named onExecute will appear to StyleCop to be using Hungarian notation, when in reality it is not. Thus, the on prefix should be flagged as an allowed prefix.

To configure the list of allowed prefixes, bring up the StyleCop settings for a project, and navigate to the Hungarian tab, as shown below:



Adding a one or two letter prefix to this list will cause StyleCop to ignore variables or fields which begin with this prefix.

If the field or variable name is intended to match the name of an item associated with Win32 or COM, and thus needs to use Hungarian notation, place the field or variable within a special NativeMethods class. A NativeMethods class is any class which contains a name ending in NativeMethods, and is intended as a placeholder for Win32 or COM wrappers. StyleCop will ignore this violation if the item is placed within a NativeMethods class.

How to Fix Violations

To fix a violation of this rule, remove the Hungarian notation from the field or variable name, or place the item within a NativeMethods class if appropriate.

SA1306: FieldNamesMustBeginWithLowerCaseLetter

Cause
The name of a field or variable in C# does not begin with a lower-case letter

Rule Description
A violation of this rule occurs when the name of a field or variable begins with an upper-case letter. Field and variable names must begin with a lower-case letter, unless the field is public or internal, const, or non-private and readonly. In these cases, the field should begin with an upper-case letter.

If the field or variable name is intended to match the name of an item associated with Win32 or COM, and thus needs to begin with an upper-case letter, place the field or variable within a special NativeMethods class. A NativeMethods class is any class which contains a name ending in NativeMethods, and is intended as a placeholder for Win32 or COM wrappers. StyleCop will ignore this violation if the item is placed within a NativeMethods class.

How to Fix Violations
To fix a violation of this rule, change the name of the field or variable so that it begins with a lower-case letter, or place the item within a NativeMethods class if appropriate.

SA1307: AccessibleFieldsMustBeginWithUpperCaseLetter

Cause
The name of a public or internal field in C# does not begin with an upper-case letter

Rule Description
A violation of this rule occurs when the name of a public or internal field begins with a lower-case letter. Public or internal fields must being with an upper-case letter.

If the field or variable name is intended to match the name of an item associated with Win32 or COM, and thus needs to start with a lower-case letter, place the field or variable within a special NativeMethods class. A NativeMethods class is any class which contains a name ending in NativeMethods, and is intended as a placeholder for Win32 or COM wrappers. StyleCop will ignore this violation if the item is placed within a NativeMethods class.

How to Fix Violations

To fix a violation of this rule, change the name of the field so that it begins with an upper-case letter, or place the item within a NativeMethods class if appropriate.

SA1308: VariableNamesMustNotBePrefixed

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.

If the field or variable name is intended to match the name of an item associated with Win32 or COM, and thus needs to begin with the prefix, place the field or variable within a special NativeMethods class. A NativeMethods class is any class which contains a name ending in NativeMethods, and is intended as a placeholder for Win32 or COM wrappers. StyleCop will ignore this violation if the item is placed within a NativeMethods class.

How to Fix Violations

To fix a violation of this rule, remove the prefix from the beginning of the field name, or place the item within a NativeMethods class if appropriate.

SA1309: FieldNamesMustNotBeginWithUnderscore

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.

If the field or variable name is intended to match the name of an item associated with Win32 or COM, and thus needs to begin with an underscore, place the field or variable within a special NativeMethods class. A NativeMethods class is any class which contains a name ending in NativeMethods, and is intended as a placeholder for Win32 or COM wrappers. StyleCop will ignore this violation if the item is placed within a NativeMethods class.

How to Fix Violations
To fix a violation of this rule, remove the underscore from the beginning of the field name, or place the item within a NativeMethods class if appropriate.

SA1310: FieldNamesMustNotContainUnderscore

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.

If the field or variable name is intended to match the name of an item associated with Win32 or COM, and thus needs to include underscores, place the field or variable within a special NativeMethods class. A NativeMethods class is any class which contains a name ending in NativeMethods, and is intended as a placeholder for Win32 or COM wrappers. StyleCop will ignore this violation if the item is placed within a NativeMethods class.

How to Fix Violations

To fix a violation of this rule, remove the underscore from the name of the field, or place the item within a NativeMethods class if appropriate.

Legibilidad – Readability Rules

SA1100: DoNotPrefixCallsWithBaseUnlessLocalImplementationExists

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.'.

How to Fix Violations

To fix a violation of this rule, change the 'base.' prefix to 'this.'.

SA1101: PrefixLocalCallsWithThis

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 be 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.

How to Fix Violations

To fix a violation of this rule, insert the 'this.' prefix before the call to the class member.

SA1102: QueryClauseMustFollowPreviousClause

Cause

A C# query clause does not begin on the same line as the previous clause, or on the next line.

Rule Description

A violation of this rule occurs when a clause within a query expression does not begin on the same line as the previous clause, or on the line after the query clause. For example:

```
object x = select a in b
    from c;
```

The query clause can correctly be written as:

```
object x = select a in b from c;
```

or:

```
object x =
    select a
    in b
    from c;
```

How to Fix Violations

To fix a violation of this rule, ensure that each clause in the query expression begins on the same line as the previous clause, or on the following line.

SA1103: QueryClausesMustBeOnSeparateLinesOrAllOnOneLine

Cause

The clauses within a C# query expression are not all placed on the same line, and each clause is not placed on its own line.

Rule Description

A violation of this rule occurs when the query clauses are not either placed all on the same line, or each on its own line. For example:

```
object x = select a in b
    from c;
```

The query clause can correctly be written as:

```
object x = select a in b from c;
```

or:

```
object x =
    select a
    in b
    from c;
```

How to Fix Violations

To fix a violation of this rule, ensure that all clauses are placed together on the same line, or each clause begins on its own line.

SA1104: QueryClauseMustBeginOnNewLineWhenPreviousClauseSpansMultipleLines

Cause

A clause within a C# query expression begins on the same line as the previous clause, when the previous clause spans across multiple lines.

Rule Description

A violation of this rule occurs when a query clause spans across multiple lines, and the next clause begins on the same line as the end of the previous clause.

```
object x =
    select a
    in b.GetCustomers(
        2, "x") from c;
```

The query clause can correctly be written as:

```
object x =
    select a
    in b.GetCustomers(
        2, "x")
    from c;
```

How to Fix Violations

To fix a violation of this rule, move the clause down to start on the next line.

SA1105: QueryClausesSpanningMultipleLinesMustBeginOnOwnLine

Cause

A clause within a C# query expression spans across multiple lines, and does not begin on its own line.

Rule Description

A violation of this rule occurs when a query clause spans across multiple lines, but does not begin on its own line. For example:

```
object x =
    select a in b from c.GetCustomers(
        2, "x");
```

The query clause can correctly be written as:

```
object x =
    select a
    in b
    from c.GetCustomers(
        2, "x");
```

How to Fix Violations

To fix a violation of this rule, move the clause down to start on the next line.

SA1106: CodeMustNotContainEmptyStatements

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.

How to Fix Violations

To fix a violation of this rule, remove the unneeded semicolon.

SA1107: CodeMustNotContainMultipleStatementsOnOneLine

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.

How to Fix Violations

To fix a violation of this rule, move each statement to begin on its own line.

SA1108: BlockStatementsMustNotContainEmbeddedComments

Cause

A C# statement contains a comment between the declaration of the statement and the opening curly bracket of the statement.

Rule Description

A violation of this rule occurs when the code contains a comment in between the declaration and the opening curly bracket. For example:

```
if (x != y)
// Make sure x does not equal y
{
}
```

The comment can legally be placed above the statement, or within the body of the statement:

```
// Make sure x does not equal y
if (x != y)
{
}

if (x != y)
{
    // Make sure x does not equal y
}
```

If the comment is being used to comment out a line of code, begin the comment with four forward slashes rather than two:

```
if (x != y)
////if (x == y)
{
}
```

How to Fix Violations

To fix a violation of this rule, move the comment above the statement, within the body of the statement, or remove the comment.

SA1109: BlockStatementsMustNotContainEmbeddedRegions

Cause

A C# statement contains a region tag between the declaration of the statement and the opening curly bracket of the statement.

Rule Description

A violation of this rule occurs when the code contains a region tag in between the declaration and the opening curly bracket. For example:

```
if (x != y)
#region
{
}
#endregion
```

This will result in the body of the statement being hidden when the region is collapsed.

How to Fix Violations

To fix a violation of this rule, remove the region or move it outside of the statement.

SA1110: OpeningParenthesisMustBeOnDeclarationLine

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]; }
}
```

How to Fix Violations

To fix a violation of this rule, ensure that the opening bracket is placed on the same line as the name of the method or indexer.

SA1111: ClosingParenthesisMustBeOnLineOfLastParameter

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]; }
}
```

How to Fix Violations

To fix a violation of this rule, ensure that the closing bracket is placed on the same line as the last parameter.

SA1112: ClosingParenthesisMustBeOnLineOfOpeningParenthesis

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 JoinName(string first, string last)
{
    string name = JoinStrings(
        first,
        last);
}

public int this[int x]
{
    get { return this.items[x]; }
}
```

How to Fix Violations

To fix a violation of this rule, ensure that the closing bracket is placed on the same line as the opening bracket.

SA1113: CommaMustBeOnSameLineAsPreviousParameter

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]; }
}
```

How to Fix Violations

To fix a violation of this rule, ensure that the comma is placed on the same line as the previous parameter.

SA1114: ParameterListMustFollowDeclaration

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)
{
}
```

How to Fix Violations

To fix a violation of this rule, ensure that the parameter list begins on the same line as the opening bracket, or on the next line.

SA1115: ParameterMustFollowComma

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)
{
}
```

How to Fix Violations

To fix a violation of this rule, ensure that the parameter begins on the same line as the previous comma, or on the next line.

SA1116: SplitParametersMustStartOnLineAfterDeclaration

Cause

The parameters to a C# method or indexer call or declaration span across multiple lines, but the first parameter does not start on the line after the opening bracket.

Rule Description

A violation of this rule occurs when the parameters to a method or indexer span across multiple lines, but the first parameter does not start on the line after the opening bracket. For example:

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

The parameters must begin on the line after the declaration, whenever the parameter span across multiple lines:

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

How to Fix Violations

To fix a violation of this rule, ensure that the first parameter starts on the line after the opening bracket, or place all parameters on the same line if the parameters are not too long.

SA1117: ParametersMustBeOnSameLineOrSeparateLines

Cause

The parameters to a C# method or indexer call or declaration are not all on the same line or each on a separate line.

Rule Description

A violation of this rule occurs when the parameters to a method or indexer are not all on the same line or each on its own line. For example:

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

The parameters can all be placed on the same line:

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

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

Alternatively, each parameter can be placed on its own line:

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

How to Fix Violations

To fix a violation of this rule, place all parameters on the same line, or place each parameter on its own line.

SA1118: ParametersMustNotSpanMultipleLines

Cause

A parameter to a C# method or indexer, other than the first parameter, spans across multiple lines.

Rule Description

To prevent method calls from becoming excessively complicated and unreadable, only the first parameter to a method or indexer call is allowed to span across multiple lines. The exception is an anonymous method passed as a parameter, which is always allowed to span multiple lines. A violation of this rule occurs

whenever a parameter other than the first parameter spans across multiple lines, and the parameter does not contain an anonymous method.

For example, the following code would violate this rule, since the second parameter spans across multiple lines:

```
return JoinStrings(
    "John",
    "Smith" +
    " Doe");
```

When parameters other than the first parameter span across multiple lines, it can be difficult to tell how many parameters are passed to the method. In general, the code becomes difficult to read.

To fix the example above, ensure that the parameters after the first parameter do not span across multiple lines. If this will cause a parameter to be excessively long, store the value of the parameter within a temporary variable. For example:

```
string last = "Smith" +
    " Doe";

return JoinStrings(
    "John",
    last);
```

In some cases, this will allow the method to be written even more concisely, such as:

```
return JoinStrings("John", last);
```

How to Fix Violations

To fix a violation of this rule, ensure that the parameters after the first parameter do not span multiple lines, unless the parameter contains an anonymous method.

SA1119: StatementMustNotUseUnnecessaryParenthesis

Cause

A C# statement contains parenthesis which are unnecessary and should be removed.

Rule Description

It is possible in C# to insert parenthesis around virtually any type of expression, statement, or clause, and in many situations use of parenthesis can greatly improve the readability of the code. However, excessive use of parenthesis can have the opposite effect, making it more difficult to read and maintain the code.

A violation of this rule occurs when parenthesis are used in situations where they provide no practical value. Typically, this happens anytime the parenthesis surround an expression which does not strictly require the use of parenthesis, and the parenthesis expression is located at the root of a statement. For example, the following lines of code all contain unnecessary parenthesis which will result in violations of this rule:

```
int x = (5 + b);
string y = (this.Method()).ToString();
return (x.Value);
```

In each of these statements, the extra parenthesis can be removed without sacrificing the readability of the code:

```
int x = 5 + b;
string y = this.Method().ToString();
return x.Value;
```

How to Fix Violations

To fix a violation of this rule, remove the unnecessary parenthesis.

SA1120: CommentsMustContainText

Cause

The C# comment does not contain any comment text.

Rule Description

A violation of this rule occurs whenever the code contains a C# comment which does not contain any text.

How to Fix Violations

To fix a violation of this rule, add text to the comment, or remove the comment.

SA1121: UseBuiltInTypeAlias

Cause

The code uses one of the basic C# types, but does not use the built-in alias for the type.

Rule Description

A violation of this rule occurs when one of the following types are used anywhere in the code: Array, Boolean, Byte, Char, Decimal, Double, Int16, Int32, Int64, Object, SByte, Single, String, UInt16, UInt32, UInt64.

A violation also occurs when any of these types are represented in the code using the full namespace for the type: System.Array, System.Boolean, System.Byte, System.Char, System.Decimal, System.Double, System.Int16, System.Int32, System.Int64, System.Object, System.SByte, System.Single, System.String, System.UInt16, System.UInt32, System.UInt64.

Rather than using the type name or the fully-qualified type name, the built-in aliases for these types should always be used: array, bool, byte, char, decimal, double, short, int, long, object, sbyte, single, string, ushort, uint, ulong.

The following table lists each of these types in all three formats:

Type Alias	Type	Fully Qualified Type

array	Array	System.Array
bool	Boolean	System.Boolean
byte	Byte	System.Byte
char	Char	System.Char
decimal	Decimal	System.Decimal
double	Double	System.Double
short	Int16	System.Int16
int	Int32	System.Int32
long	Int64	System.Int64
object	Object	System.Object
sbyte	SByte	System.SByte
single	Single	System.Single
string	String	System.String
ushort	UInt16	System.UInt16
uint	UInt32	System.UInt32
ulong	UInt64	System.UInt64

How to Fix Violations

To fix a violation of this rule, replace the type with the built-in alias for the type.

SA1122: UseStringEmptyForEmptyStrings

Cause

The C# code includes an empty string, written as “”.

Rule Description

A violation of this rule occurs when the code contains an empty string. For example:

```
string s = "";
```

This will cause the compiler to embed an empty string into the compiled code. Rather than including a hard-coded empty string, use the static string.Empty property:

```
string s = string.Empty;
```

How to Fix Violations

To fix a violation of this rule, replace the hard-coded empty string with string.Empty.

SA1123: DoNotPlaceRegionsWithinElements

Cause

The C# code contains a region within the body of a code element.

Rule Description

A violation of this rule occurs whenever a region is placed within the body of a code element. In many editors, including Visual Studio, the region will appear collapsed by default, hiding the code within the region. It is generally a bad practice to hide code within the body of an element, as this can lead to bad decisions as the code is maintained over time.

How to Fix Violations

To fix a violation of this rule, remove the region from the code.

SA1124: DoNotUseRegions

Cause
The C# code contains a region.

Rule Description
A violation of this rule occurs whenever a region is placed anywhere within the code. In many editors, including Visual Studio, the region will appear collapsed by default, hiding the code within the region. It is generally a bad practice to hide code, as this can lead to bad decisions as the code is maintained over time.

How to Fix Violations
To fix a violation of this rule, remove the region from the code.

Orden – Ordering Rules

SA1200: UsingDirectivesMustBePlacedWithinNamespace

Cause
A C# using directive is placed outside of a namespace element.

Rule Description
A violation of this rule occurs when a using directive or a using-alias directive is placed outside of a namespace element, unless the file does not contain any namespace elements.

For example, the following code would result in two violations of this rule.

```
using System;
using Guid = System.Guid;

namespace Microsoft.Sample
{
    public class Program
    {
    }
}
```

The following code, however, would not result in any violations of this rule:

```
namespace Microsoft.Sample
{
    using System;
    using Guid = System.Guid;

    public class Program
    {
    }
}
```

There are subtle differences between placing using directives within a namespace element, rather than outside of the namespace, including:

1. Placing using-alias directives within the namespace eliminates compiler confusion between conflicting types.

2. When multiple namespaces are defined within a single file, placing using directives within the namespace elements scopes references and aliases.

1. Eliminating Type Confusion

Consider the following code, which contains a using-alias directive defined outside of the namespace element. The code creates a new class called Guid, and also defines a using-alias directive to map the name Guid to the type System.Guid. Finally, the code creates an instance of the type Guid:

```
using Guid = System.Guid;

namespace Microsoft.Sample
{
    public class Guid
    {
        public Guid(string s)
        {
        }
    }

    public class Program
    {
        public static void Main(string[] args)
        {
```

```
            Guid g = new Guid("hello");
        }
    }
}
```

This code will compile cleanly, without any compiler errors. However, it is unclear which version of the Guid type is being allocated. If the using directive is moved inside of the namespace, as shown below, a compiler error will occur:

```
namespace Microsoft.Sample
{
    using Guid = System.Guid;

    public class Guid
    {
        public Guid(string s)
        {
        }
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            Guid g = new Guid("hello");
        }
    }
}
```

The code fails on the following compiler error, found on the line containing `Guid g = new Guid("hello");`

CS0576: Namespace 'Microsoft.Sample' contains a definition conflicting with alias 'Guid'

The code creates an alias to the System.Guid type called Guid, and also creates its own type called Guid with a matching constructor interface. Later, the code creates an instance of the type Guid. To create this instance, the compiler must choose between the two different definitions of Guid. When the using-alias directive is placed outside of the namespace element, the compiler will choose the local definition of Guid defined within the local namespace, and completely ignore the using-alias directive defined outside of the namespace. This, unfortunately, is not obvious when reading the code.

When the using-alias directive is positioned within the namespace, however, the compiler has to choose between two different, conflicting Guid types both defined within the same namespace. Both of these types provide a matching constructor. The compiler is unable to make a decision, so it flags the compiler error.

Placing the using-alias directive outside of the namespace is a bad practice because it can lead to confusion in situations such as this, where it is not obvious which version of the type is actually being used. This can potentially lead to a bug which might be difficult to diagnose.

Placing using-alias directives within the namespace element eliminates this as a source of bugs.

2. Multiple Namespaces

Placing multiple namespace elements within a single file is generally a bad idea, but if and when this is done, it is a good idea to place all using directives within each of the namespace elements, rather than globally at the top of the file. This will scope the namespaces tightly, and will also help to avoid the kind of behavior described above.

It is important to note that when code has been written with using directives placed outside of the namespace, care should be taken when moving these directives within the namespace, to ensure that this is not changing the semantics of the code. As explained above, placing using-alias directives within the namespace element allows the compiler to choose between conflicting types in ways that will not happen when the directives are placed outside of the namespace.

How to Fix Violations

To fix a violation of this rule, move all using directives and using-alias directives within the namespace element.

SA1201: ElementsMustAppearInTheCorrectOrder

Cause

An element within a C# code file is out of order in relation to the other elements in the code.

Rule Description

A violation of this rule occurs when the code elements within a file do not follow a standard ordering scheme.

To comply with this rule, elements at the file root level or within a namespace must be positioned in the following order:

- Extern Alias Directives
- Using Directives
- Namespaces
- Delegates
- Enums
- Interfaces
- Structs
- Classes

Within a class, struct, or interface, elements must be positioned in the following order:

- Fields
- Constructors
- Finalizers (Destructors)
- Delegates
- Events
- Enums
- Interfaces
- Properties
- Indexers
- Methods
- Structs
- Classes

Complying with a standard ordering scheme based on element type can increase the readability and maintainability of the file and encourage code reuse.

When implementing an interface, it is sometimes desirable to group all members of the interface next to one another. This will sometimes require violating this rule, if the interface contains elements of different types. This problem can be solved through the use of partial classes.

1. Add the partial attribute to the class, if the class is not already partial.

2. Add a second partial class with the same name. It is possible to place this in the same file, just below the original class, or within a second file.

3. Move the interface inheritance and all members of the interface implementation to the second part of the class.

For example:

```
/// <summary>
/// Represents a customer of the system.
/// </summary>
public partial class Customer
{
    // Contains the main functionality of the class.
}

/// <content>
/// Implements the ICollection class.
/// </content>
public partial class Customer : ICollection
{
    public int Count
    {
        get { return this.count; }
    }

    public bool IsSynchronized
    {
        get { return false; }
    }

    public object SyncRoot
    {
        get { return null; }
    }

    public void CopyTo(Array array, int index)
    {
        throw new NotImplementedException();
```

```
        }
    }
```

How to Fix Violations

To fix an instance of this violation, order the elements in the file in the order described above.

SA1202: ElementsMustBeOrderedByAccess

Cause

An element within a C# code file is out of order within regard to access level, in relation to other elements in the code.

Rule Description

A violation of this rule occurs when the code elements within a file do not follow a standard ordering scheme based on access level.

To comply with this rule, adjacent elements of the same type must be positioned in the following order by access level:

- public
- internal
- protected internal
- protected
- private

Complying with a standard ordering scheme based on access level can increase the readability and maintainability of the file and make it easier to identify the public interface that is being exposed from a class.

How to Fix Violations

To fix an instance of this violation, order the elements in the file in the order described above.

SA1203: ConstantsMustAppearBeforeFields

Cause

A const field is placed beneath a non-const field.

Rule Description

A violation of this rule occurs when a const field is placed beneath a non-const field. Constants must be placed above fields to indicate that the two are fundamentally different types of elements with different considerations for the compiler, different naming requirements, etc.

How to Fix Violations

To fix an instance of this violation, place all consts above all fields.

SA1204: StaticElementsMustAppearBeforeInstanceElements

Cause

A static element is positioned beneath an instance element of the same type.

Rule Description

A violation of this rule occurs when a static element is positioned beneath an instance element of the same type. All static elements must be placed above all instance elements of the same type to make it easier to see the interface exposed from the instance and static version of the class.

How to Fix Violations

To fix an instance of this violation, place all static elements above all instance elements of the same type.

Glossary Item Box

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.

How To Fix Violations

To fix an instance of this violation, order the keywords in the element's declaration as described above.

SA1207: ProtectedMustComeBeforeInternal

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.

How to Fix Violations

To fix an instance of this violation, place the protected keyword before the internal keyword.

SA1208: SystemUsingDirectivesMustBePlacedBeforeOtherUsingDirectives

Cause

A using directive which declares a member of the System namespace appears after a using directive which declares a member of a different namespace, within a C# code file.

Rule Description

A violation of this rule occurs when a using directive for the System namespace is placed after a non-System using directive. Placing all System using directives at the top of the using directives can make the code cleaner and easier to read, and can help make it easier to identify the namespaces that are being used by the code.

How to Fix Violations

To fix an instance of this violation, place the System using directive above all using directives for other namespaces.

SA1209: UsingAliasDirectivesMustBePlacedAfterOtherUsingDirectives

Cause

A using-alias directive is positioned before a regular using directive.

Rule Description

A violation of this rule occurs when a using-alias directive is placed before a normal using directive. Using-alias directives have special behavior which can alter the meaning of the rest of the code within the file or

namespace. Placing the using-alias directives together below all other using-directives can make the code cleaner and easier to read, and can help make it easier to identify the types used throughout the code.

How to Fix Violations

To fix an instance of this violation, place all using-alias directives beneath all normal using directives.

SA1210: UsingDirectivesMustBeOrderedAlphabeticallyByNamespace

Cause

The using directives within a C# code file are not sorted alphabetically by namespace.

Rule Description

A violation of this rule occurs when the using directives are not sorted alphabetically by namespace. Sorting the using directives alphabetically makes the code cleaner and easier to read, and can help make it easier to identify the namespaces that are being used by the code.

How to Fix Violations

To fix an instance of this violation, order the using directives alphabetically by namespace.

SA1211: UsingAliasDirectivesMustBeOrderedAlphabeticallyByAliasName

Cause

The using-alias directives within a C# code file are not sorted alphabetically by alias name.

Rule Description

A violation of this rule occurs when the using-alias directives are not sorted alphabetically by alias name. Sorting the using-alias directives alphabetically can make the code cleaner and easier to read, and can help make it easier to identify the namespaces that are being used by the code.

How to Fix Violations

To fix an instance of this violation, order the using-alias directives alphabetically by alias name.

SA1212: PropertyAccessorsMustFollowOrder

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:

```
publicstring Name
{
    set { this.name = value; }
get { return this.name; }
}
```

The code below would not raise this violation:

```
publicstring Name
{
get { return this.name; }

    set { this.name = value; }
}
```

How to Fix Violations

To fix an instance of this violation, place the get accessor before the set accessor.

SA1213: EventAccessorsMustFollowOrder

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:

```
publicevent 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; }
}
```

How to Fix Violations

To fix an instance of this violation, place the add accessor before the remove accessor.

Espacios – Spacing Rules

SA1000: KeywordsMustBeSpacedCorrectly

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.

How to Fix Violations
To fix a violation of this rule, add or remove a space after the keyword, according to the description above.

SA1001: CommasMustBeSpaceCorrectly

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.

How to Fix Violations
To fix a violation of this rule, ensure that the comma is followed by a single space, and is not preceded by any space.

SA1002: SemicolonsMustBeSpaceCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that the semicolon is followed by a single space, and is not preceded by any space.

SA1003: SymbolsMustBeSpaceCorrectly

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 occurs 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)
{
}
```

How to Fix Violations

To fix a violation of this rule, ensure that the spacing around the symbol follows the rule described above.

SA1004: DocumentationLinesMustBeginWithSingleSpace

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)
{
}
```

How to Fix Violations

To fix a violation of this rule, ensure that the header line begins with a single space.

SA1005: SingleLineCommentsMustBeginWithSingleSpace

Cause

A single-line comment within a C# code file does not begin with a single space.

Rule Description

A violation of this rule occurs when a single-line comment does not begin with a single space. For example:

```
private void Method1()
{
    //A single-line comment.
    //   A single-line comment.
}
```

The comments should begin with a single space after the leading forward slashes:

```
private void Method1()
{
    // A single-line comment.
    // A single-line comment.
}
```

An exception to this rule occurs when the comment is being used to comment out a line of code. In this case, the space can be omitted if the comment begins with four forward slashes to indicate out-commented code. For example:

```
private void Method1()
{
    ////int x = 2;
    ////return x;
}
```

How to Fix Violations

To fix a violation of this rule, ensure that the comment begins with a single space. If the comment is being used to comment out a line of code, ensure that the comment begins with four forward slashes, in which case the leading space can be omitted.

SA1006: PreprocessorKeywordsMustNotBePrecededBySpace

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
```

How to Fix Violations

To fix a violation of this rule, ensure that there is no whitespace between the opening hash mark and the preprocessor-type keyword.

SA1007: OperatorKeywordMustBeFollowedBySpace

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)
{
}
```

How to Fix Violations

To fix a violation of this rule, add a single space after the operator keyword.

SA1008: OpeningParenthesisMustBeSpacedCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that the spacing around the opening parenthesis follows the rule described above.

SA1009: ClosingParenthesisMustBeSpacedCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that the spacing around the closing parenthesis follows the rule described above.

SA1010: OpeningSquareBracketsMustBeSpacedCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that there is no whitespace on either side of the opening square bracket.

SA1011: ClosingSquareBracketsMustBeSpacedCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that the spacing around the closing square bracket follows the rule described above.

SA1012: OpeningCurlyBracketsMustBeSpacedCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that the spacing around the opening curly bracket follows the rule described above.

SA1013: ClosingCurlyBracketsMustBeSpacedCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that the spacing around the closing curly bracket follows the rule described above.

SA1014: OpeningGenericBracketsMustBeSpacedCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that there is no whitespace on either side of the opening generic bracket.

SA1015: ClosingGenericBracketsMustBeSpacedCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that there is no whitespace before the closing generic bracket.

SA1016: OpeningAttributeBracketsMustBeSpacedCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that there is no whitespace after the opening attribute bracket.

SA1017: ClosingAttributeBracketsMustBeSpacedCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that there is no whitespace before the closing attribute bracket.

SA1018: NullableTypeSymbolsMustNotBePrecededBySpace

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.

How to Fix Violations

To fix a violation of this rule, ensure that there is no whitespace before the nullable type symbol.

SA1019: MemberAccessSymbolsMustBeSpacedCorrectly

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 or last character on the line.

How to Fix Violations
To fix a violation of this rule, ensure that the member access symbol is not surrounded by any whitespace.

SA1020: IncrementDecrementSymbolsMustBeSpacedCorrectly

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.

How to Fix Violations
To fix a violation of this rule, ensure that there is no whitespace between the increment or decrement symbol and the item that is being incremented or decremented.

SA1021: NegativeSignsMustBeSpacedCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that the spacing around the negative sign follows the rule described above.

SA1022: PositiveSignsMustBeSpacedCorrectly

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.

How to Fix Violations

To fix a violation of this rule, ensure that the spacing around the positive sign follows the rule described above.

SA1023: DereferenceAndAccessOfMustBeSpacedCorrectly

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 or a parenthesis. The symbol should not be followed by whitespace, and should not be the last character on the line. For example:

```
y = *x;
```

How to Fix Violations

To fix a violation of this rule, ensure that the spacing around the dereference or address-of symbol follows the rule described above.

SA1024: ColonsMustBeSpacedCorrectly

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;
```

How to Fix Violations

To fix a violation of this rule, ensure that the spacing around the colon follows the rule described above.

SA1025: CodeMustNotContainMultipleWhitespaceInARow

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, or come after a comma or semicolon.

How to Fix Violations
To fix a violation of this rule, remove the extra whitespace characters are leave only a single space.

SA1026: CodeMustNotContainSpaceAfterNewKeywordInImplicitlyTypedArrayAllocation

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 };
```

How to Fix Violations
To fix a violation of this rule, remove any whitespace between the new keyword and the opening array bracket.

SA1027: TabsMustNotBeUsed

Cause
The C# code contains a tab character.

Rule Description
A violation of this rule occurs whenever the code contains a tab character.

Tabs should not be used within C# code, because the length of the tab character can vary depending upon the editor being used to view the code. This can cause the spacing and indexing of the code to vary from the developer's original intention, and can in some cases make the code difficult to read.

For these reasons, tabs should not be used, and each level of indentation should consist of four spaces. This will ensure that the code looks the same no matter which editor is being used to view the code.

How to Fix Violations

To fix a violation of this rule, remove the tab character from the code.

Mantenimiento – Maintainability Rules

SA1400: AccessModifierMustBeDeclared

Cause

The access modifier for a C# element has not been explicitly defined.

Rule Description

C# allows elements to be defined without an access modifier. Depending upon the type of element, C# will automatically assign an access level to the element in this case.

This rule requires an access modifier to be explicitly defined for every element. This removes the need for the reader to make assumptions about the code, improving the readability of the code.

How to Fix Violations

To fix a violation of this rule, add an access modifier to the declaration of the element.

SA1401: FieldsMustBePrivate

Cause

A field within a C# class has an access modifier other than private.

Rule Description

A violation of this rule occurs whenever a field in a class is given non-private access. For maintainability reasons, properties should always be used as the mechanism for exposing fields outside of a class, and fields should always be declared with private access. This allows the internal implementation of the property to change over time without changing the interface of the class.

Fields located within C# structs are allowed to have any access level.

How to Fix Violations

To fix a violation of this rule, make the field private and add a property to expose the field outside of the class.

SA1402: FileMayOnlyContainASingleClass

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.

How to Fix Violations

To fix an instance of this violation, move each class into its own file.

SA1403: FileMayOnlyContainASingleNamespace

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.

How to Fix Violations

To fix a violation of this rule, ensure that the file only contains a single namespace.

SA1404: CodeAnalysisSuppressionMustHaveJustification

Cause

A Code Analysis SuppressMessage attribute does not include a justification.

Rule Description

A violation of this rule occurs when the code contains a Code Analysis SuppressMessage attribute, but a justification for the suppression has not been provided within the attribute. Whenever a Code Analysis rule is suppressed, a justification should be provided. This can increase the long-term maintainability of the code.

```
    [SuppressMessage("Microsoft.Performance",    "CA1804:RemoveUnusedLocals",
Justification = "Used during unit testing")]
    public bool Enable()
    {
```

```
}
```

How to Fix Violations

To fix an instance of this violation, add a Justification tag and justification text to the SuppressMessage attribute describing the reason for the suppression.

SA1405: DebugAssertMustProvideMessageText

Cause

A call to Debug.Assert in C# code does not include a descriptive message.

Rule Description

A violation of this rule occurs when the code contains a call to Debug.Assert which does not provide a description for the end-user. For example, the following assert includes a description message:

```
Debug.Assert(value != true, "The value must always be true.");
```

How to Fix Violations

To fix a violation of this rule, add a descriptive message to the assert which will appear to the end user when the assert is fired.

SA1406: DebugFailMustProvideMessageText

Cause

A call to Debug.Fail in C# code does not include a descriptive message.

Rule Description

A violation of this rule occurs when the code contains a call to Debug.Fail which does not provide a description for the end-user. For example, the following call includes a description message:

```
Debug.Fail("The code should never reach this point.");
```

How to Fix Violations

To fix an instance of this violation, add a descriptive message to the Debug.Fail call which will appear to the end user when the assert is fired.

SA1407: ArithmeticExpressionsMustDeclarePrecedence

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.

How to Fix Violations

To fix a violation of this rule, insert parenthesis within the arithmetic expression to declare the precedence of the operations.

SA1408: ConditionalExpressionsMustDeclarePrecedence

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:

```
if (x || y && z && a || b)
{
}
```

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. For example, a developer could write this code as:

```
if ((x || y) && z && (a || b))
{
}
```

or

```
if (x || (y && z && a) || b)
```

```
{
}
```

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

How to Fix Violations

Insert parenthesis within the conditional expression to declare the precedence of the operations.

SA1409: RemoveUnnecessaryCode

Cause

A C# file contains code which is unnecessary and can be removed without changing the overall logic of the code.

Rule Description

A violation of this rule occurs when the file contains code which can be removed without changing the overall logic of the code.

For example, the following try-catch statement could be removed completely since the try and catch blocks are both empty.

```
try
{
}
catch (Exception ex)
{
}
```

The try-finally statement below does contain code within the try block, but it does not contain any catch blocks, and the finally block is empty. Thus, the try-finally is not performing any useful function and can be removed.

```
try
```

```
{
    this.Method();
}
finally
{
}
```

As a final example, the unsafe statement below is empty, and thus provides no value.

```
unsafe
{
}
```

How to Fix Violations

The fix a violation of this rule, remove the unnecessary code, or fill in the code with additional statements.

SA1410: RemoveDelegateParenthesisWhenPossible

Cause

A call to a C# anonymous method does not contain any method parameters, yet the statement still includes parenthesis.

Rule Description

When an anonymous method does not contain any method parameters, the parenthesis around the parameters are optional.

A violation of this rule occurs when the parenthesis are present on an anonymous method call which takes no method parameters. For example:

```
this.Method(delegate() { return 2; });
```

The parenthesis are unnecessary and should be removed:

```
this.Method(delegate { return 2; });
```

How to Fix Violations

Remove the unnecessary parenthesis after the delegate keyword.

Documentación – Documentation Rules

SA1600: ElementsMustBeDocumented

Cause

A C# code element is missing a documentation header.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if an element is completely missing a documentation header, or if the header is empty. In C# the following types of elements can have documentation headers: classes, constructors, delegates, enums, events, finalizers, indexers, interfaces, methods, properties, and structs.

How to Fix Violations

To fix a violation of this rule, add or fill-in a documentation header for the element.

For example, the following example shows a method with a valid documentation header:

```
/// <summary>
/// Joins a first name and a last name together into a single string.
/// </summary>
/// <param name="firstName">The first name to join.</param>
/// <param name="lastName">The last name to join.</param>
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
```

```
        return firstName + " " + lastName;
    }
```

SA1601: PartialElementsMustBeDocumented

Cause

A C# partial element is missing a documentation header.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if a partial element (an element with the partial attribute) is completely missing a documentation header, or if the header is empty. In C# the following types of elements can be attributed with the partial attribute: classes, methods.

When documentation is provided on more than one part of the partial class, the documentation for the two classes may be merged together to form a single source of documentation. For example, consider the following two parts of a partial class:

```
/// <summary>
/// Documentation for the first part of Class1.
/// </summary>
public partial class Class1
{
}

/// <summary>
/// Documentation for the second part of Class1.
/// </summary>
public partial class Class1
{
}
```

These two different parts of the same partial class each provide different documentation for the class. When the documentation for this class is built into an SDK, the tool building the documentation will either

choose to use only one part of the documentation for the class and ignore the other parts, or, in some cases, it may merge the two sources of documentation together, to form a string like: “Documentation for the first part of Class1. Documentation for the second part of Class1.”

For these reasons, it can be problematic to provide SDK documentation on more than one part of the partial class. However, it is still advisable to document each part of the class, to increase the readability and maintainability of the code, and StyleCop will require that each part of the class contain header documentation.

This problem is solved through the use of the <content> tag, which can replace the <summary> tag for partial classes. The recommended practice for documenting partial classes is to provide the official SDK documentation for the class on the main part of the partial class. This documentation should be written using the standard <summary> tag. All other parts of the partial class should omit the <summary> tag completely, and replace it with a <content> tag. This allows the developer to document all parts of the partial class while still centralizing all of the official SDK documentation for the class onto one part of the class. The <content> tags will be ignored by the SDK documentation tools.

How to Fix Violations

To fix a violation of this rule, add or fill-in a documentation header for the element.

For example, the following example shows two parts of a partial class, one containing a <summary> header and another containing a <content> header.

```
/// <summary>
/// Represents a customer in the database.
/// </summary>
public partial class Customer
{
}

/// <content>
/// Contains auto-generated functionality for the Customer class.
/// </content>
public partial class Customer
{
}
```

SA1602: EnumerationItemsMustBeDocumented

Cause

An item within a C# enumeration is missing an Xml documentation header.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when an item within an enumeration is missing a header. For example:

```
/// <summary>
/// Types of animals.
/// </summary>
public enum Animals
{
    Dog,
    Cat,
    Horse
}
```

How to Fix Violations

To fix a violation of this rule, add a documentation header for each item within the enum. For example:

```
/// <summary>
/// Types of animals.
/// </summary>
public enum Animals
{
    /// <summary>
    /// Represents a dog.
    /// </summary>
    Dog,

    /// <summary>
    /// Represents a cat.
    /// </summary>
    Cat,

    /// <summary>
    /// Represents a horse.
```

```
        /// </summary>
        Horse
    }
```

SA1603: DocumentationMustContainValidXml

Cause

The Xml within a C# element's document header is badly formed.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when the documentation Xml is badly formed and cannot be parsed. This can occur if the Xml contains invalid characters, or if an Xml node is missing a closing tag, for example.

How to Fix Violations

To fix a violation of this rule, replace the badly formed Xml with valid Xml that can be parsed by a standard Xml parser.

The following example shows a class containing invalid Xml within its documentation header. The closing tag for the <summary> node is invalid.

```
/// <summary>
/// An example of badly formed Xml.
/// </summa3ry>
public class Example
{
}
```

SA1604: ElementDocumentationMustHaveSummary

Cause

The Xml header documentation for a C# element is missing a <summary> tag.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when the element documentation is missing a <summary> tag.

How to Fix Violations

To fix a violation of this rule, add and fill-in a <summary> tag for the element, containing a description of the element.

The following example shows a class containing invalid Xml within its documentation header. The closing tag for the <summary> node is invalid.

```
/// <summary>
/// Represents a customer in the database.
/// </summary>
public class Customer
{
}
```

SA1605: PartialElementDocumentationMustHaveSummary

Cause

The <summary> or <content> tag within the documentation header for a C# code element is missing or empty.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when the documentation header for a partial element (an element with the partial attribute) is missing a <summary> or <content> tag, or contains an empty <summary> or <content> tag which does not contain a description of the element. In C# the following types of elements can be attributed with the partial attribute: classes, methods.

When documentation is provided on more than one part of the partial class, the documentation for the two classes may be merged together to form a single source of documentation. For example, consider the following two parts of a partial class:

```
/// <summary>
/// Documentation for the first part of Class1.
/// </summary>
public partial class Class1
{
}

/// <summary>
/// Documentation for the second part of Class1.
/// </summary>
public partial class Class1
{
}
```

These two different parts of the same partial class each provide different documentation for the class. When the documentation for this class is built into an SDK, the tool building the documentation will either choose to use only one part of the documentation for the class and ignore the other parts, or, in some cases, it may merge the two sources of documentation together, to form a string like: “Documentation for the first part of Class1. Documentation for the second part of Class1.”

For these reasons, it can be problematic to provide SDK documentation on more than one part of the partial class. However, it is still advisable to document each part of the class, to increase the readability and maintainability of the code, and StyleCop will require that each part of the class contain header documentation.

This problem is solved through the use of the <content> tag, which can replace the <summary> tag for partial classes. The recommended practice for documenting partial classes is to provide the official SDK documentation for the class on the main part of the partial class. This documentation should be written using the standard <summary> tag. All other parts of the partial class should omit the <summary> tag completely, and replace it with a <content> tag. This allows the developer to document all parts of the partial class while still centralizing all of the official SDK documentation for the class onto one part of the class. The <content> tags will be ignored by the SDK documentation tools.

How to Fix Violations

To fix a violation of this rule, add and fill-in a <summary> or <content> tag with a description of the code element.

The following example shows a partial class with a fill-in <summary> tag.

```
/// <summary>
/// Represents a customer in the database.
/// </summary>
public partial class Customer
{
}
```

SA1606: ElementDocumentationMustHaveSummaryText

Cause

The <summary> tag within the documentation header for a C# code element is empty.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when the documentation header for an element contains an empty <summary> tag which does not contain a description of the element.

How to Fix Violations

To fix a violation of this rule, fill-in the <summary> tag with a description of the code element.

Example

The following example shows a method which contains an empty <summary> tag.

```
/// <summary>  </summary>
/// <param name="customerId">The ID of the customer to find.</param>
/// <returns>The customer, or null if the customer could not be
/// found.</returns>
public Customer FindCustomer(int customerId)
{
    // ... finds the customer ...
}
```

SA1607: PartialElementDocumentationMustHaveSummaryText

Cause

The <summary> or <content> tag within the documentation header for a C# code element is empty.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when the documentation header for a partial element (an element with the partial attribute) contains an empty <summary> tag or <content> tag which does not contain a description of the element. In C# the following types of elements can be attributed with the partial attribute: classes, methods.

When documentation is provided on more than one part of the partial class, the documentation for the two classes may be merged together to form a single source of documentation. For example, consider the following two parts of a partial class:

```
/// <summary>
/// Documentation for the first part of Class1.
/// </summary>
public partial class Class1
{
}

/// <summary>
/// Documentation for the second part of Class1.
/// </summary>
public partial class Class1
{
}
```

These two different parts of the same partial class each provide different documentation for the class. When the documentation for this class is built into an SDK, the tool building the documentation will either choose to use only one part of the documentation for the class and ignore the other parts, or, in some cases, it may merge the two sources of documentation together, to form a string like: “Documentation for the first part of Class1. Documentation for the second part of Class1.”

For these reasons, it can be problematic to provide SDK documentation on more than one part of the partial class. However, it is still advisable to document each part of the class, to increase the readability and maintainability of the code, and StyleCop will require that each part of the class contain header documentation.

This problem is solved through the use of the <content> tag, which can replace the <summary> tag for partial classes. The recommended practice for documenting partial classes is to provide the official SDK documentation for the class on the main part of the partial class. This documentation should be written using the standard <summary> tag. All other parts of the partial class should omit the <summary> tag completely, and replace it with a <content> tag. This allows the developer to document all parts of the partial class while still centralizing all of the official SDK documentation for the class onto one part of the class. The <content> tags will be ignored by the SDK documentation tools.

How to Fix Violations

To fix a violation of this rule, fill-in the contents of the summary tag or content tag with a description of the code element.

The following example shows a method which contains an empty <summary> tag.

```
/// <summary>  </summary>
/// <param name="customerId">The ID of the customer to find.</param>
/// <returns>The customer, or null if the customer could not be found.</returns>
public Customer FindCustomer(int customerId)
{
    // ... finds the customer ...
}
```

To fix the violation, add valid summary text. For example:

```
/// <summary>Attempts to locate a record for the customer with the given ID.</summary>
/// <param name="customerId">The ID of the customer to find.</param>
/// <returns>The customer, or null if the customer could not be found.</returns>
public Customer FindCustomer(int customerId)
{
    // ... finds the customer ...
}
```

SA1608: ElementDocumentationMustNotHaveDefaultSummary

Cause

The <summary> tag within an element's Xml header documentation contains the default text generated by Visual Studio during the creation of the element.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

Visual Studio provides helper functionality for adding new elements such as classes to a project. Visual Studio will create a default documentation header for the new class and fill in this header with default documentation text.

A violation of this rule occurs when the <summary> tag for a code element still contains the default documentation text generated by Visual Studio.

How to Fix Violations

To fix a violation of this rule, replace the default documentation text with new text describing the contents of the code element.

The following example shows a class which contains the default summary text generated by Visual Studio.

```
/// <summary>
/// Summary description for the Example class.
/// </summary>
public class Example
{
}
```

SA1609: PropertyDocumentationMustHaveValue

Cause

The Xml header documentation for a C# property does not contain a <value> tag.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

The documentation for properties may include a <value> tag, which describes the value held by the property.

A violation of this rule occurs when the <value> tag for a property is missing.

How to Fix Violations

To fix a violation of this rule, add and fill-in a <value> tag within the documentation header for the property.

The following example shows a property which contains a <value> tag within its documentation header.

```
/// <summary>
/// Gets the name of the customer.
/// </summary>
/// <value>The name of the customer.</value>
public bool Name
{
    get { return this.name; }
}
```

SA1610: PropertyDocumentationMustHaveValueText

Cause

The Xml header documentation for a C# property contains an empty <value> tag.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

The documentation for properties may include a <value> tag, which describes the value held by the property.

A violation of this rule occurs when the <value> tag for a property is empty.

How to Fix Violations

To fix a violation of this rule, fill-in a description of the value held by the property within the <value> tag.

Example

The following example shows a property which contains a <value> tag within its documentation header.

```
/// <summary>
/// Gets the name of the customer.
/// </summary>
/// <value>The name of the customer.</value>
public bool Name
{
    get { return this.name; }
}
```

SA1611: ElementParametersMustBeDocumented

Cause

A C# method, constructor, delegate or indexer element is missing documentation for one or more of its parameters.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if an element containing parameters is missing documentation for one or more of its parameters.

How to Fix Violations

To fix a violation of this rule, add or fill-in documentation text within a <param> tag for each parameter within the element.

The following example shows a method with a valid documentation header:

```
/// <summary>
/// Joins a first name and a last name together into a single string.
/// </summary>
/// <param name="firstName">The first name to join.</param>
/// <param name="lastName">The last name to join.</param>
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
    return firstName + " " + lastName;
}
```

SA1612: ElementParameterDocumentationMustMatchElementParameters

Cause

The documentation describing the parameters to a C# method, constructor, delegate or indexer element does not match the actual parameters on the element.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if the documentation for an element's parameters does not match the actual parameters on the element, or if the parameter documentation is not listed in the same order as the element's parameters.

How to Fix Violations

To fix a violation of this rule, correct the parameter documentation so that the <param> tags in the documentation appear in the same order as the element's parameters, and so that there is one <param> tag for each parameter on the element.

The following example shows a method with a valid documentation header:

```
/// <summary>
/// Joins a first name and a last name together into a single string.
/// </summary>
/// <param name="firstName">The first name to join.</param>
/// <param name="lastName">The last name to join.</param>
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
    return firstName + " " + lastName;
}
```

SA1613: ElementParameterDocumentationMustDeclareParameterName

Cause

A <param> tag within a C# element's documentation header is missing a name attribute containing the name of the parameter.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if the documentation for an element contains a <param> tag which is missing a name attribute, or which contains an empty name attribute.

How to Fix Violations

To fix a violation of this rule, add or fill-in the name attribute for the <param> tag to indicate the name of the parameter that the documentation is for.

The following example shows a method with a valid documentation header:

```
/// <summary>
/// Joins a first name and a last name together into a single string.
/// </summary>
/// <param name="firstName">The first name to join.</param>
/// <param name="lastName">The last name to join.</param>
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
    return firstName + " " + lastName;
}
```

SA1614: ElementParameterDocumentationMustHaveText

Cause

A <param> tag within a C# element's documentation header is empty.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if the documentation for an element contains a <param> tag which is empty and does not contain a description of the parameter.

How to Fix Violations

To fix a violation of this rule, fill-in a description of the parameter within the <param> tag.

The following example shows a method with a valid documentation header:

```
/// <summary>
```

```
/// Joins a first name and a last name together into a single string.
/// </summary>
/// <param name="firstName">The first name to join.</param>
/// <param name="lastName">The last name to join.</param>
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
    return firstName + " " + lastName;
}
```

SA1615: ElementReturnValueMustBeDocumented

Cause

A C# element is missing documentation for its return value.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if an element containing a return value is missing a <returns> tag.

How to Fix Violations

To fix a violation of this rule, add and fill-in documentation text within a <returns> tag describing the value returned from the element.

The following example shows a method with a valid documentation header:

```
/// <summary>
/// Joins a first name and a last name together into a single string.
/// </summary>
/// <param name="firstName">The first name to join.</param>
/// <param name="lastName">The last name to join.</param>
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
    return firstName + " " + lastName;
```

```
}
```

SA1616: ElementReturnValueDocumentationMustHaveText

Cause

The <returns> tag within a C# element's documentation header is empty.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if an element contains an empty <returns> tag.

How to Fix Violations

To fix a violation of this rule, fill-in documentation text within the <returns> tag describing the value returned from the element.

The following example shows a method with a valid documentation header:

```
/// <summary>
/// Joins a first name and a last name together into a single string.
/// </summary>
/// <param name="firstName">The first name to join.</param>
/// <param name="lastName">The last name to join.</param>
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
    return firstName + " " + lastName;
}
```

SA1617: VoidReturnValueMustNotBeDocumented

Cause

A C# code element does not contain a return value, or returns void, but the documentation header for the element contains a <returns> tag.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if an element which returns void contains a <returns> tag within its documentation header.

How to Fix Violations

To fix a violation of this rule, remove the <returns> tag from the element.

The following example shows a method with a valid documentation header:

```
/// <summary>
/// Prints the given name.
/// </summary>
/// <param name="firstName">The first name.</param>
/// <param name="lastName">The last name.</param>
public void PrintNames(string firstName, string lastName)
{
    Console.WriteLine(firstName + " " + lastName);
}
```

SA1618: GenericTypeParametersMustBeDocumented

Cause

A generic C# element is missing documentation for one or more of its generic type parameters.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if an element containing generic type parameters is missing documentation for one or more of its generic type parameters.

How to Fix Violations

To fix a violation of this rule, add or fill-in documentation text within a <typeparam> tag for each generic type parameter on the element.

The following example shows a method with a valid documentation header:

```
/// <summary>
/// A sample generic class.
/// </summary>
/// <typeparam name="S">The first generic type parameter.</typeparam>
/// <typeparam name="T">The second generic type parameter.</typeparam>
public class Class1<S, T>
{
}
```

SA1619: GenericTypeParametersMustBeDocumentedPartialClass

Cause

A generic, partial C# element is missing documentation for one or more of its generic type parameters, and the documentation for the element contains a <summary> tag.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when a generic, partial element is missing documentation for one or more of its generic type parameters, and the documentation for the element contains a <summary> tag rather than a <content> tag.

When documentation is provided on more than one part of the partial class, the documentation for the two classes may be merged together to form a single source of documentation. For example, consider the following two parts of a partial class:

```
/// <summary>
/// Documentation for the first part of Class1.
/// </summary>
public partial class Class1
{
}

/// <summary>
/// Documentation for the second part of Class1.
/// </summary>
public partial class Class1
{
}
```

These two different parts of the same partial class each provide different documentation for the class. When the documentation for this class is built into an SDK, the tool building the documentation will either choose to use only one part of the documentation for the class and ignore the other parts, or, in some cases, it may merge the two sources of documentation together, to form a string like: “Documentation for the first part of Class1. Documentation for the second part of Class1.”

For these reasons, it can be problematic to provide SDK documentation on more than one part of the partial class. However, it is still advisable to document each part of the class, to increase the readability and maintainability of the code, and StyleCop will require that each part of the class contain header documentation.

This problem is solved through the use of the <content> tag, which can replace the <summary> tag for partial classes. The recommended practice for documenting partial classes is to provide the official SDK documentation for the class on the main part of the partial class. This documentation should be written using the standard <summary> tag. All other parts of the partial class should omit the <summary> tag completely, and replace it with a <content> tag. This allows the developer to document all parts of the

partial class while still centralizing all of the official SDK documentation for the class onto one part of the class. The <content> tags will be ignored by the SDK documentation tools.

When a generic element contains a <summary> tag within its documentation header, StyleCop assumes that this is the main part of the class, and requires the header to contain <typeparam> tags for each of the generic type parameters on the class. However, if the documentation header for the class contains a <content> tag rather than a <summary> tag, StyleCop will assume that the generic type parameters are defined on another part of the class, and will not require <typeparam> tags on this part of the class.

How to Fix Violations

To fix a violation of this rule, add or fill-in documentation text within a <typeparam> tag for each generic type parameter on the element, or change the <summary> tag to a <content> tag if this is not the main part of the partial class.

The following example shows a method with a valid documentation header:

```
/// <summary>
/// A sample generic class.
/// </summary>
/// <typeparam name="S">The first generic type parameter.</typeparam>
/// <typeparam name="T">The second generic type parameter.</typeparam>
public class Class1<S, T>
{
}
```

SA1620: GenericTypeParameterDocumentationMustMatchTypeParameters

Cause

The <typeparam> tags within the Xml header documentation for a generic C# element do not match the generic type parameters on the element.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a

description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if the <typeparam> tags within the element's header documentation do not match the generic type parameters on the element, or do not appear in the same order as the element's type parameters.

How to Fix Violations

To fix a violation of this rule, add and fill-in one <typeparam> tag for each generic type parameter on the element, and make sure that the <typeparam> tags appear in the same order as the element's type parameters.

The following example shows a method with a valid documentation header:

```
/// <summary>
/// A sample generic class.
/// </summary>
/// <typeparam name="S">The first generic type parameter.</typeparam>
/// <typeparam name="T">The second generic type parameter.</typeparam>
public class Class1<S, T>
{
}
```

SA1621: GenericTypeParameterDocumentationMustDeclareParameterName

Cause

A <typeparam> tag within the Xml header documentation for a generic C# element is missing a name attribute, or contains an empty name attribute.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if the element contains a <typeparam> tag within its Xml header documentation which does not declare the name of the type parameter.

How to Fix Violations

To fix a violation of this rule, add or fill-in the name attribute for each <typeparam> tag, indicating the name of the type parameter that the documentation is for.

The following example shows a method with a valid documentation header:

```
/// <summary>
/// A sample generic class.
/// </summary>
/// <typeparam name="S">The first generic type parameter.</typeparam>
/// <typeparam name="T">The second generic type parameter.</typeparam>
public class Class1<S, T>
{
}
```

SA1622: GenericTypeParameterDocumentationMustHaveText

Cause

A <typeparam> tag within the Xml header documentation for a generic C# element is empty.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if the element contains an empty <typeparam> tag within its Xml header documentation.

How to Fix Violations

To fix a violation of this rule, fill-in each <typeparam> tag within a description of the generic type parameter.

The following example shows a method with a valid documentation header:

```
/// <summary>
/// A sample generic class.
/// </summary>
/// <typeparam name="S">The first generic type parameter.</typeparam>
/// <typeparam name="T">The second generic type parameter.</typeparam>
public class Class1<S, T>
{
}
```

SA1623: PropertySummaryDocumentationMustMatchAccessors

Cause

The documentation text within a C# property's <summary> tag does not match the accessors within the property.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs if a property's summary documentation does not match the accessors within the property.

The property's summary text must begin with wording describing the types of accessors exposed within the property. If the property contains only a get accessor, the summary must begin with the word "Gets". If the property contains only a set accessor, the summary must begin with the word "Sets". If the property exposes both a get and set accessor, the summary text must begin with "Gets or sets".

For example, consider the following property, which exposes both a get and set accessor. The summary text begins with the words “Gets or sets”.

```
/// <summary>
/// Gets or sets the name of the customer.
/// </summary>
public string Name
{
    get { return this.name; }
    set { this.name = value; }
}
```

If the property returns a Boolean value, an additional rule is applied. The summary text for Boolean properties must contain the words “Gets a value indicating whether”, “Sets a value indicating whether”, or “Gets or sets a value indicating whether”. For example, consider the following Boolean property, which only exposes a get accessor:

```
/// <summary>
/// Gets a value indicating whether the item is enabled.
/// </summary>
public bool Enabled
{
    get { return this.enabled; }
}
```

In some situations, the set accessor for a property can have more restricted access than the get accessor. For example:

```
/// <summary>
/// Gets the name of the customer.
/// </summary>
public string Name
{
    get { return this.name; }
    private set { this.name = value; }
}
```

In this example, the set accessor has been given private access, meaning that it can only be accessed by local members of the class in which it is contained. The get accessor, however, inherits its access from the parent property, thus it can be accessed by any caller, since the property has public access.

In this case, the documentation summary text should avoid referring to the set accessor, since it is not visible to external callers.

StyleCop applies a series of rules to determine when the set accessor should be referenced in the property's summary documentation. In general, these rules require the set accessor to be referenced whenever it is visible to the same set of callers as the get accessor, or whenever it is visible to external classes or inheriting classes.

The specific rules for determining whether to include the set accessor in the property's summary documentation are:

1. The set accessor has the same access level as the get accessor. For example:

```
/// <summary>
/// Gets or sets the name of the customer.
/// </summary>
protected string Name
{
    get { return this.name; }
    set { this.name = value; }
}
```

2. The property is only internally accessible within the assembly, and the set accessor also has internal access. For example:

```
internal class Class1
{
    /// <summary>
    /// Gets or sets the name of the customer.
    /// </summary>
    protected string Name
    {
        get { return this.name; }
        internal set { this.name = value; }
    }
}
```

```
internal class Class1
{
    public class Class2
```

```
    {
        /// <summary>
        /// Gets or sets the name of the customer.
        /// </summary>
        public string Name
        {
            get { return this.name; }
            internal set { this.name = value; }
        }
    }
}
```

3. The property is private or is contained beneath a private class, and the set accessor has any access modifier other than private. In the example below, the access modifier declared on the set accessor has no meaning, since the set accessor is contained within a private class and thus cannot be seen by other classes outside of Class1. This effectively gives the set accessor the same access level as the get accessor.

```
public class Class1
{
    private class Class2
    {
        public class Class3
        {
            /// <summary>
            /// Gets or sets the name of the customer.
            /// </summary>
            public string Name
            {
                get { return this.name; }
                internal set { this.name = value; }
            }
        }
    }
}
```

4. Whenever the set accessor has protected or protected internal access, it should be referenced in the documentation. A protected or protected internal set accessor can always been seen by a class inheriting from the class containing the property.

```
internal class Class1
{
    public class Class2
    {
        /// <summary>
        /// Gets or sets the name of the customer.
        /// </summary>
        internal string Name
        {
```

```
            get { return this.name; }
            protected set { this.name = value; }
        }
    }

    private class Class3 : Class2
    {
        public Class3(string name) { this.Name = name; }
    }
}
```

How to Fix Violations

To fix a violation of this rule, update the property's summary text so that the description begins with the proper wording, depending upon the type of the property and the types of accessors within the property.

SA1624: PropertySummaryDocumentationMustOmitSetAccessorWithRestrictedAccess

Cause

The documentation text within a C# property's <summary> tag takes into account all of the accessors within the property, but one of the accessors has limited access.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when one of the accessors within the property has limited access (usually the set accessor), but the summary documentation text for the property still refers to both accessors.

Normally, a property's summary text must begin with wording describing the types of accessors exposed within the property. If the property contains only a get accessor, the summary must begin with the word "Gets". If the property contains only a set accessor, the summary must begin with the word "Sets". If the property exposes both a get and set accessor, the summary text must begin with "Gets or sets".

However, when an accessor within the property is given an access level which is more limited than the access level of the property, this accessor should be omitted from the summary documentation.

It can sometimes be non-obvious whether the set accessor within a property is actually less accessible than the get accessor. For example, consider the case where a public property is contained within an internal class, and the set accessor is given internal accessor. In effect, both the get and set accessors have the same access level. In this case, the summary documentation should refer to both the get and set accessors, since they effectively have the same access level.

In some situations, the set accessor for a property can have more restricted access than the get accessor. For example:

```
/// <summary>
/// Gets the name of the customer.
/// </summary>
public string Name
{
    get { return this.name; }
    private set { this.name = value; }
}
```

In this example, the set accessor has been given private access, meaning that it can only be accessed by local members of the class in which it is contained. The get accessor, however, inherits its access from the parent property, thus it can be accessed by any caller, since the property has public access.

In this case, the documentation summary text should avoid referring to the set accessor, since it is not visible to external callers.

StyleCop applies a series of rules to determine when the set accessor should be referenced in the property's summary documentation. In general, these rules require the set accessor to be referenced whenever it is visible to the same set of callers as the get accessor, or whenever it is visible to external classes or inheriting classes.

The specific rules for determining whether to include the set accessor in the property's summary documentation are:

1. The set accessor has the same access level as the get accessor. For example:

```
/// <summary>
/// Gets or sets the name of the customer.
/// </summary>
protected string Name
{
    get { return this.name; }
    set { this.name = value; }
}
```

2. The property is only internally accessible within the assembly, and the set accessor also has internal access. For example:

```
internal class Class1
{
    /// <summary>
    /// Gets or sets the name of the customer.
    /// </summary>
    protected string Name
    {
        get { return this.name; }
        internal set { this.name = value; }
    }
}
```

```
internal class Class1
{
    public class Class2
    {
        /// <summary>
        /// Gets or sets the name of the customer.
        /// </summary>
        public string Name
        {
            get { return this.name; }
            internal set { this.name = value; }
        }
    }
}
```

3. The property is private or is contained beneath a private class, and the set accessor has any access modifier other than private. In the example below, the access modifier declared on the set accessor has no meaning, since the set accessor is contained within a private class and thus cannot be seen by other classes outside of Class1. This effectively gives the set accessor the same access level as the get accessor.

```
public class Class1
{
    private class Class2
    {
        public class Class3
        {
            /// <summary>
            /// Gets or sets the name of the customer.
            /// </summary>
            public string Name
            {
                get { return this.name; }
                internal set { this.name = value; }
            }
        }
    }
}
```

4. Whenever the set accessor has protected or protected internal access, it should be referenced in the documentation. A protected or protected internal set accessor can always been seen by a class inheriting from the class containing the property.

```
internal class Class1
{
    public class Class2
    {
        /// <summary>
        /// Gets or sets the name of the customer.
        /// </summary>
        internal string Name
        {
            get { return this.name; }
            protected set { this.name = value; }
        }
    }

    private class Class3 : Class2
    {
        public Class3(string name) { this.Name = name; }
    }
}
```

How to Fix Violations

To fix a violation of this rule, update the property's summary text and remove wording which refers to the limited access accessor.

SA1625: ElementDocumentationMustNotBeCopiedAndPasted

Cause

The Xml documentation for a C# element contains two or more identical entries, indicating that the documentation has been copied and pasted. This can sometimes indicate invalid or poorly written documentation.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when an element contains two or more identical documentation texts. For example:

```
/// <summary>
/// Joins a first name and a last name together into a single string.
/// </summary>
/// <param name="firstName">Part of the name.</param>
/// <param name="lastName">Part of the name.</param>
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
    return firstName + " " + lastName;
}
```

In some cases, a method may contain one or more parameters which are not used within the body of the method. In this case, the documentation for the parameter can be set to "The parameter is not used." StyleCop will allow multiple parameters to contain identical documentation as long as the documentation string is "The parameter is not used."

How to Fix Violations

To fix a violation of this rule, edit the documentation for the element and ensure that each of the individual documentation texts are unique. For example:

```
    /// <summary>
    /// Joins a first name and a last name together into a single string.
    /// </summary>
    /// <param name="firstName">The first name to join.</param>
    /// <param name="lastName">The last name to join.</param>
    /// <returns>The joined names.</returns>
    public string JoinNames(string firstName, string lastName)
    {
        return firstName + " " + lastName;
    }
```

SA1626: SingleLineCommentsMustNotUseDocumentationStyleSlashes

Cause

The C# code contains a single-line comment which begins with three forward slashes in a row.

Rule Description

A violation of this rule occurs when the code contains a single-line comment which begins with three slashes. Comments beginning with three slashes are reserved for Xml documentation headers. Single-line comments should begin with only two slashes. When commenting out lines of code, it is advisable to begin the comment with four slashes to differentiate it from normal comments. For example:

```
    /// <summary>
    /// Joins a first name and a last name together into a single string.
    /// </summary>
    /// <param name="firstName">Part of the name.</param>
    /// <param name="lastName">Part of the name.</param>
    /// <returns>The joined names.</returns>
    public string JoinNames(string firstName, string lastName)
    {
A legal comment beginning with two slashes:
        // Join the names together.
        string fullName = firstName + " " + lastName;

An illegal comment beginning with three slashes:
        /// Trim the name.
        fullName = fullName.Trim();

A line of commented-out code beginning with four slashes:
        ////fullName = asfd;

        return fullName;
    }
```

How to Fix Violations

To fix a violation of this rule, remove a slash from the beginning of the comment so that it begins with only two slashes.

SA1627: DocumentationTextMustNotBeEmpty

Cause

The Xml header documentation for a C# code element contains an empty tag.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when the documentation header for an element contains an empty tag. For example:

```
/// <summary>
/// Joins a first name and a last name together into a single string.
/// </summary>
/// <param name="firstName"> </param>
/// <param name="lastName">Part of the name.</param>
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
    ...
}
```

How to Fix Violations

To fix a violation of this rule, add documentation text within the empty tag.

SA1628: DocumentationTextMustBeginWithACapitalLetter

Cause

A section of the Xml header documentation for a C# element does not begin with a capital letter.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when part of the documentation does not begin with a capital letter. For example, the summary text in the documentation below begins with a lower-case letter:

```
/// <summary>
/// joins a first name and a last name together into a single string.
/// </summary>
/// <param name="firstName">The first name.</param>
/// <param name="lastName">The last name.</param>
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
    ...
}
```

How to Fix Violations

To fix a violation of this rule, ensure that all sections of the documentation begin with a capital letter.

SA1629: DocumentationTextMustEndWithAPeriod

Cause

A section of the Xml header documentation for a C# element does not end with a period.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when part of the documentation does not end with a period. For example, the summary text in the documentation below does not end with a period:

```
/// <summary>
/// Joins a first name and a last name together into a single string
/// </summary>
/// <param name="firstName">The first name.</param>
/// <param name="lastName">The last name.</param>
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
    ...
}
```

How to Fix Violations

To fix a violation of this rule, ensure that all sections of the documentation end with a period.

SA1630: DocumentationTextMustContainWhitespace

Cause

A section of the Xml header documentation for a C# element does not contain any whitespace between words.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when part of the documentation does contain any whitespace between words. This can indicate poorly written or poorly formatted documentation. For example:

```
/// <summary>
/// Joinsnames
/// </summary>
```

```
/// <param name="firstName">First</param>
/// <param name="lastName">Last</param>
/// <returns>Name</returns>
public string JoinNames(string firstName, string lastName)
{
    ...
}
```

How to Fix Violations

To fix a violation of this rule, ensure that all sections of the documentation contain at least one instance of whitespace between words.

SA1631: DocumentationTextMustMeetCharacterPercentage

Cause

A section of the Xml header documentation for a C# element does not contain enough alphabetic characters.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when part of the documentation does contain enough characters. This rule is calculated by counting the number of alphabetic characters and numbers within the documentation text, and comparing it against the number of symbols and other non-alphabetic characters. If the percentage of non-alphabetic characters is too high, this generally indicates poorly formatted documentation which will be difficult to read. For example, consider the follow summary documentation:

```
/// <summary>
/// @)$(*A name--------
/// </summary>
public class Name
{
    ...
}
```

How to Fix Violations

To fix a violation of this rule, rewrite the documentation text using grammatically proper language, and ensure that the ratio of symbols versus characters in the text is not too great.

SA1632: DocumentationTextMustMeetMinimumCharacterLength

Cause

A section of the Xml header documentation for a C# element is too short.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when part of the documentation is too short. This can often indicate that the documentation is not descriptive. For example:

```
/// <summary>
/// A name
/// </summary>
public class Name
{
    ...
}
```

How to Fix Violations

To fix a violation of this rule, rewrite the documentation text using grammatically proper and descriptive language. In most cases, doing so will cause the length of the documentation text to be greater than the minimum length which causes this rule to fire.

SA1633: FileMustHaveHeader

Cause

A C# code file is missing a standard file header.

Rule Description

A violation of this rule occurs when a C# source file is missing a file header. The file header must begin on the first line of the file, and must be formatted as a block of comments containing Xml, as follows:

```
//-----------------------------------------------------------------------
// <copyright file="NameOfFile.cs" company="CompanyName">
//     Company copyright tag.
// </copyright>
//-----------------------------------------------------------------------
```

For example, a file called Widget.cs from a fictional company called Sprocket Enterprises should contain a file header similar to the following:

```
//-----------------------------------------------------------------------
// <copyright file="Widget.cs" company="Sprocket Enterprises">
//     Copyright (c) Sprocket Enterprises. All rights reserved.
// </copyright>
//-----------------------------------------------------------------------
```

The dashed lines at the top and bottom of the header are not strictly necessary, so the header could be written as:

```
// <copyright file="Widget.cs" company="Sprocket Enterprises">
//     Copyright (c) Sprocket Enterprises. All rights reserved.
// </copyright>
```

It is possible to add additional tags, although they will not be checked or enforced by StyleCop:

```
//-----------------------------------------------------------------------
// <copyright file="Widget.cs" company="Sprocket Enterprises">
//     Copyright (c) Sprocket Enterprises. All rights reserved.
// </copyright>
// <author>John Doe</author>
//-----------------------------------------------------------------------
```

A file that is completely auto-generated by a tool, and which should not be checked or enforced by StyleCop, can include an “auto-generated” header rather than the standard file header. This will cause StyleCop to ignore the file. This type of header should never be placed on top of a manually written code file.

```
// <auto-generated />
namespace Sample.Something
{
    // The contents of this file are completely auto-generated by a tool.
}
```

How to Fix Violations

To fix a violation of this rule, add a standard file header at the top of the file.

SA1634: FileHeaderMustShowCopyright

Cause

The file header at the top of a C# code file is missing a copyright tag.

Rule Description

A violation of this rule occurs when the file header at the top of a C# file is missing a copyright tag. For example:

```
//-----------------------------------------------------------------------
// <Tag>A file header which does not contain a copyright tag</Tag>
//-----------------------------------------------------------------------
```

A file header should include a copyright tag, as follows:

```
//-----------------------------------------------------------------------
// <copyright file="Widget.cs" company="Sprocket Enterprises">
//     Copyright (c) Sprocket Enterprises. All rights reserved.
// </copyright>
//-----------------------------------------------------------------------
```

How to Fix Violations

To fix a violation of this rule, add a standard copyright tag to the file header.

SA1635: FileHeaderMustHaveCopyrightText

Cause

The file header at the top of a C# code file is missing copyright text.

Rule Description

A violation of this rule occurs when the file header at the top of a C# file does not contain text within its copyright tag. For example:

```
//-----------------------------------------------------------------------
// <copyright file="Widget.cs" company="Sprocket Enterprises">
// </copyright>
//-----------------------------------------------------------------------
```

A file header should include copyright text, as follows:

```
//-----------------------------------------------------------------------
// <copyright file="Widget.cs" company="Sprocket Enterprises">
//     Copyright (c) Sprocket Enterprises. All rights reserved.
// </copyright>
//-----------------------------------------------------------------------
```

How to Fix Violations

To fix a violation of this rule, add your company's standard copyright text to the copyright tag.

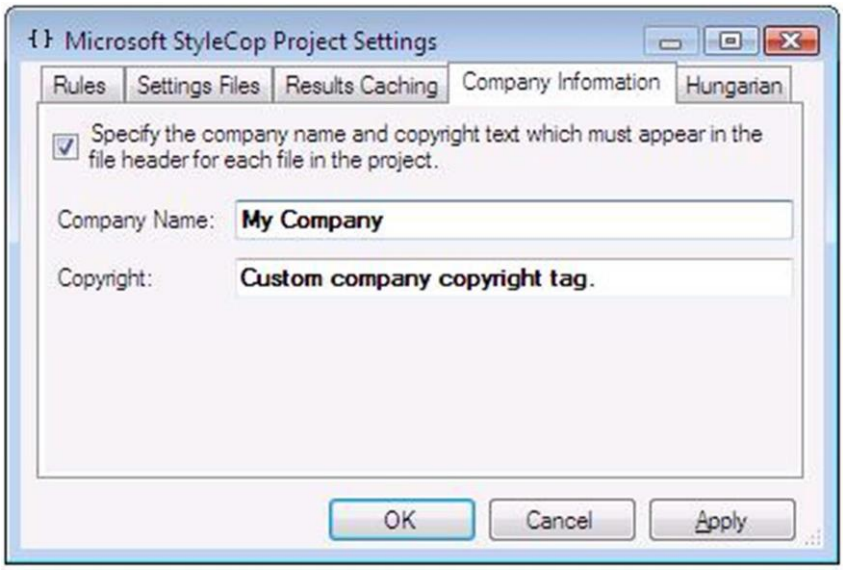
SA1636: FileHeaderCopyrightTextMustMatch

Cause

The file header at the top of a C# code file does not contain the appropriate copyright text.

Rule Description

A violation of this rule occurs when the file header at the top of a C# file does not contain the copyright text that has been specified for the project. To enable this rule, navigate to the StyleCop settings for the project and change to the Company Information tab, as shown below:



Check the checkbox at the top of the settings page, and fill in the required copyright text for your company. Click OK to save the settings. With these settings in place, every file within the project must contain the required copyright text within its file header copyright tag, as shown in the example below:

```
//-----------------------------------------------------------------------
// <copyright file="Widget.cs" company="My Company">
//     Custom company copyright tag.
// </copyright>
//-----------------------------------------------------------------------
```

How to Fix Violations

To fix a violation of this rule, add your company's standard copyright text to the file header copyright tag.

SA1637: FileHeaderMustContainFileName

Cause

The file header at the top of a C# code file is missing the file name.

Rule Description

A violation of this rule occurs when the file header at the top of a C# file does not contain a valid file name tag. For example:

```
//-----------------------------------------------------------------------
// <copyright company="Sprocket Enterprises">
//     Copyright (c) Sprocket Enterprises. All rights reserved.
// </copyright>
//-----------------------------------------------------------------------
```

A file header should include a file tag containing the name of the file, as follows:

```
//-----------------------------------------------------------------------
// <copyright file="Widget.cs" company="Sprocket Enterprises">
//     Copyright (c) Sprocket Enterprises. All rights reserved.
// </copyright>
//-----------------------------------------------------------------------
```

How to Fix Violations

To fix a violation of this rule, add a file tag containing the name of the file.

SA1638: FileHeaderFileNameDocumentationMustMatchFileName

Cause

The file tag within the file header at the top of a C# code file does not contain the name of the file.

Rule Description

A violation of this rule occurs when the file tag within the file header at the top of a C# file does not contain the name of the file. For example, consider a C# source file named File1.cs, with the following header:

```
//-----------------------------------------------------------------------
```

```
// <copyright file="File2.cs" company="Sprocket Enterprises">
//     Copyright (c) Sprocket Enterprises. All rights reserved.
// </copyright>
//-----------------------------------------------------------------------
```

A violation of this rule would occur, since the file tag does not contain the name of the file. The header should be written as:

```
//-----------------------------------------------------------------------
// <copyright file="File1.cs" company="Sprocket Enterprises">
//     Copyright (c) Sprocket Enterprises. All rights reserved.
// </copyright>
//-----------------------------------------------------------------------
```

How to Fix Violations

To fix a violation of this rule, add the name of the file to the file tag.

SA1639: FileHeaderMustHaveSummary

Cause

The file header at the top of a C# code file does not contain a filled-in summary tag.

Rule Description

A violation of this rule occurs when the file header at the top of a C# file does not contain a valid summary tag. This rule is disabled by default.

For example:

```
//-----------------------------------------------------------------------
// <copyright file="Widget.cs" company="Sprocket Enterprises">
//     Copyright (c) Sprocket Enterprises. All rights reserved.
// </copyright>
//-----------------------------------------------------------------------
```

If this rule is enabled, the file header should contain a summary tag. For example:

```
//-----------------------------------------------------------------------
// <copyright file="Widget.cs" company="Sprocket Enterprises">
//     Copyright (c) Sprocket Enterprises. All rights reserved.
// </copyright>
// <summary>Defines the Widget class.</summary>
//-----------------------------------------------------------------------
```

How to Fix Violations

To fix a violation of this rule, add and fill-in a summary tag describing the contents of the file.

SA1640: FileHeaderMustHaveValidCompanyText

Cause

The file header at the top of a C# code file does not contain company name text.

Rule Description

A violation of this rule occurs when the file header at the top of a C# file does not contain a company tag with company name text. For example:

```
//-----------------------------------------------------------------------
// <copyright file="Widget.cs" company="Sprocket Enterprises">
//     Copyright (c) Sprocket Enterprises. All rights reserved.
// </copyright>
//-----------------------------------------------------------------------
```

How to Fix Violations

To fix a violation of this rule, add and fill-in a company attribute containing the name of the company.

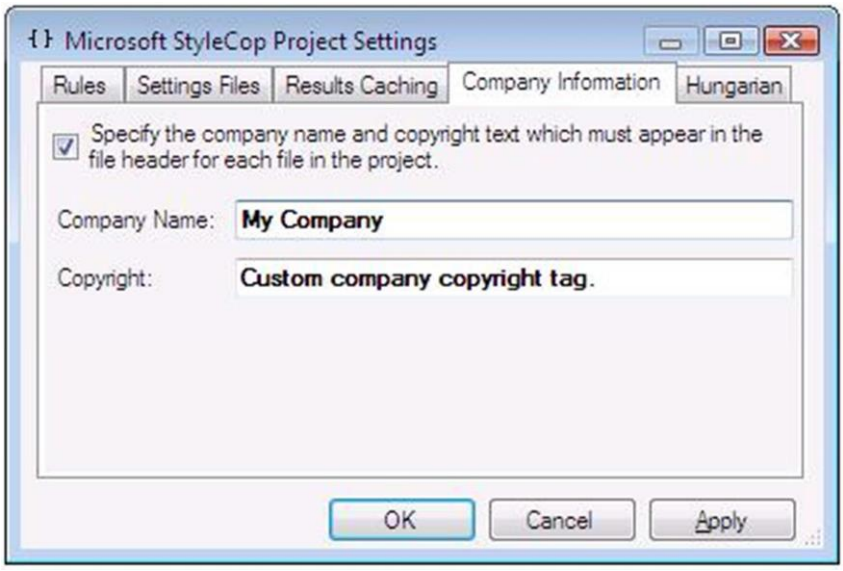
SA1641: FileHeaderCompanyNameTextMustMatch

Cause

The file header at the top of a C# code file does not contain the appropriate company name text.

Rule Description

A violation of this rule occurs when the file header at the top of a C# file does not contain the company name text that has been specified for the project. To enable this rule, navigate to the StyleCop settings for the project and change to the Company Information tab, as shown below:



Check the checkbox at the top of the settings page, and fill in the required company name text for your company. Click OK to save the settings. With these settings in place, every file within the project must contain the required company name text within its file header copyright tag, as shown in the example below:

```
//-----------------------------------------------------------------------
// <copyright file="Widget.cs" company="My Company">
//     Custom company copyright tag.
// </copyright>
//-----------------------------------------------------------------------
```

How to Fix Violations

To fix a violation of this rule, add your company's standard company name text to the file header copyright tag.

SA1642: ConstructorSummaryDocumentationMustBeginWithStandardText

Cause

The Xml documentation header for a C# constructor does not contain the appropriate summary text.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when the summary tag within the documentation header for a constructor does not begin with the proper text.

The rule is intended to standardize the summary text for a constructor based on the access level of the constructor. The summary for a non-private instance constructor must begin with “Initializes a new instance of the {class name} class.” For example, the following shows the constructor for the Customer class.

```
/// <summary>
/// Initializes a new instance of the Customer class.
/// </summary>
public Customer()
{
}
```

It is possible to embed other tags into the summary text. For example:

```
/// <summary>
/// Initializes a new instance of the <see cref="Customer"/> class.
/// </summary>
public Customer()
{
}
```

If the class contains generic parameters, these can be annotated within the cref link using either of the following two formats:

```
/// <summary>
/// Initializes a new instance of the <see cref="Customer`1"/> class.
/// </summary>
public Customer()
{
}

/// <summary>
/// Initializes a new instance of the <see cref="Customer{T}"/> class.
/// </summary>
public Customer()
{
}
```

If the constructor is static, the summary text should begin with “Initializes static members of the {class name} class.” For example:

```
/// <summary>
/// Initializes static members of the Customer class.
/// </summary>
public static Customer()
{
}
```

Private instance constructors must use the summary text “Prevents a default instance of the {class name} class from being created.”

```
/// <summary>
/// Prevents a default instance of the Customer class from being created.
/// </summary>
private Customer()
{
}
```

How to Fix Violations

To fix a violation of this rule, edit the summary text for the constructor as described above.

SA1643: DestructorSummaryDocumentationMustBeginWithStandardText

Cause

The Xml documentation header for a C# finalizer does not contain the appropriate summary text.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when the summary tag within the documentation header for a finalizer does not begin with the proper text.

The rule is intended to standardize the summary text for a finalizer. The summary for a finalizer must begin with “Finalizes an instance of the {class name} class.” For example, the following shows the finalizer for the Customer class.

```
/// <summary>
/// Finalizes an instance of the Customer class.
/// </summary>
~Customer()
{
}
```

It is possible to embed other tags into the summary text. For example:

```
/// <summary>
/// Finalizes an instance of the <see cref="Customer"/> class.
/// </summary>
~Customer()
{
}
```

If the class contains generic parameters, these can be annotated within the cref link using either of the following two formats:

```
/// <summary>
/// Initializes a new instance of the <see cref="Customer`1"/> class.
/// </summary>
public Customer()
{
}

/// <summary>
/// Initializes a new instance of the <see cref="Customer{T}"/> class.
/// </summary>
public Customer()
{
}
```

How to Fix Violations

To fix a violation of this rule, edit the summary text for the finalizer as described above.

SA1644: DocumentationHeadersMustNotContainBlankLines

Cause

A section within the Xml documentation header for a C# element contains blank lines.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

A violation of this rule occurs when the documentation header contains one or more blank lines within a section of documentation. For example:

```
/// <summary>
/// Joins a first name and a last name together into a single string.
///
/// Uses a simple form of string concatenation.
/// </summary>
/// <param name="firstName">The first name to join.</param>
/// <param name="lastName">The last name to join.</param>
```

```
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
    return firstName + " " + lastName;
}
```

Rather than inserting blank lines into the documentation, use the <para> tag to denote paragraphs. For example:

```
/// <summary>
/// <para>
/// Joins a first name and a last name together into a single string.
/// </para><para>
/// Uses a simple form of string concatenation.
/// </para>
/// </summary>
/// <param name="firstName">The first name to join.</param>
/// <param name="lastName">The last name to join.</param>
/// <returns>The joined names.</returns>
public string JoinNames(string firstName, string lastName)
{
    return firstName + " " + lastName;
}
```

How to Fix Violations

To fix a violation of this rule, remove the blank lines from the documentation header, and optionally replace them with <para/> tags.

SA1645: IncludedDocumentationFileDoesNotExist

Cause

An included Xml documentation file does not exist.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

As an alternative to authoring documentation directly within the code file, it is possible to place documentation for multiple elements within a separate Xml file, and then reference a section of that file within an element's documentation header. This causes the compiler to import the documentation for that element from the included document. For example:

```
///<include file="IncludedDocumentation.xml" path="root/EnabledMethodDocs" />
public bool Enabled(bool true)
{
}
```

A violation of this rule occurs when the included file does not exist at the given location, or cannot be loaded.

How to Fix Violations

To fix a violation of this rule, correct the path to the included documentation file to point to a valid file location.

SA1646: IncludedDocumentationXPathDoesNotExist

Cause

An included Xml documentation link contains an invalid path.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

As an alternative to authoring documentation directly within the code file, it is possible to place documentation for multiple elements within a separate Xml file, and then reference a section of that file within an element's documentation header. This causes the compiler to import the documentation for that element from the included document. For example:

```
///<include file="IncludedDocumentation.xml" path="root/EnabledMethodDocs" />
public bool Enabled(bool true)
{
}
```

A violation of this rule occurs when the path attribute does not link to a valid path within the included documentation file.

How to Fix Violations

To fix a violation of this rule, correct the value of the path attribute to point to a valid path within the file.

SA1647: IncludeNodeDoesNotContainValidFileAndPath

Cause

An include tag within an Xml documentation header does not contain valid file and path attribute.

Rule Description

C# syntax provides a mechanism for inserting documentation for classes and elements directly into the code, through the use of Xml documentation headers. For an introduction to these headers and a description of the header syntax, see the following article: http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

As an alternative to authoring documentation directly within the code file, it is possible to place documentation for multiple elements within a separate Xml file, and then reference a section of that file within an element's documentation header. This causes the compiler to import the documentation for that element from the included document. For example:

```
///<include file="IncludedDocumentation.xml" path="root/EnabledMethodDocs" />
public bool Enabled(bool true)
{
}
```

A violation of this rule occurs when the include tag is missing a file or path attribute, or contains an improperly formatted file or path attribute.

How to Fix Violations

To fix a violation of this rule, add or correct the file and path attributes.