Philips Healthcare - C# Coding Standard

Version 2.0

issued by the CCB Coding Standards Philips Healthcare

External Use of this Document

The C# coding standard as defined by Philips Healthcare and published via the TIOBE website (http://www.tiobe.com) may be used "as-is" by any interested party.

You may copy, adapt, and redistribute this document for non-commercial use or for your own internal use in a commercial setting. However, you may not republish this document, nor may you publish or distribute any adaptation of this document for other than non-commercial use or your own internal use, without first obtaining express written approval from Philips Healthcare. Philips Healthcare will not be liable for any direct, indirect, special or consequential damages arising out of any use of the document or the performance or implementation of the contents thereof.

Please send questions and suggestions about the C# coding standard and/or its code checker ClockSharp to info@tiobe.com.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Introduction

1.1. Objective

This Coding Standard requires or recommends certain practices for developing programs in the C# language. The objective of this coding standard is to have a positive effect on

- Avoidance of errors/bugs, especially the hard-to-find ones.
- Maintainability, by promoting some proven design principles.
- Maintainability, by requiring or recommending a certain unity of style.
- Performance, by dissuading wasteful practices.

1.2. Scope

This standard pertains to the use of the C# language. With few exceptions, it does **not** discuss the use of the .NET class libraries.

This standard does not include rules on how to layout brackets, braces, and code in general.

1.3. Rationale

Reasons to have a coding standard and to comply with it are not given here, except the objectives listed in section 1.1. In this section the origins of the rules are given and some explanation why these were chosen.

1.3.1. Sources of inspiration

Many of the rules were taken from the MSDN C# Usage Guidelines ([MS Design]). The naming guidelines in that document are identical to those found in Appendix C of the ECMA C# Language Specification ([C# Lang]).

Many other recommendations and a few design patterns were also taken from [MS Design].

Some general good practices, most of them concerning Object-Oriented programming, were copied from the Philips Healthcare C++ Coding Standard ([C++ Coding Standard]).

The numbering scheme and some of the structure have been copied from [C++ Coding Standard].

1.3.2. Contrast with C++

A considerable part of a coding standard for C or C++ could be condensed into a single rule, avoid undefined behavior, and maybe shun implementation defined behavior. Officially C# does not exhibit any of these, barring a few minor, well-defined exceptions. Most examples of undefined behavior in C++ will cause an exception to be thrown in C#. Although this is an improvement on the ?anything might happen? of C++, it is highly undesirable for post-release software.

1.4. Applicability

This coding standard applies to all C# code that is part of Philips Healthcare software products or directly supportive to these products. Third party software is constrained by this standard if this software is developed specifically for Philips Healthcare.

1.5. Notational conventions

1.5.1. Rule

A rule should be broken only for compelling reasons where no reasonable alternative can be found. The author of the violating code shall consult with at least one knowledgeable colleague and a senior designer to review said necessity. A comment in the code explaining the reason for the violation is mandatory.

1.5.2. Checkable

Rules in this coding standard are marked checkable if automatic verification of compliance is enforced by static analyzers.

1.5.3. Examples

Please note that the source code formatting in some examples has been chosen for compactness rather than for demonstrating good practice. The use of a certain compact style in some of the examples is considered suitable for tiny code fragments, but should not be emulated in ?real? code.

General rules (General)

Rules

2@105	Do not mix code from different providers in one file

Rule 2@105

Synopsis: Do not mix code from different providers in one file
Language: C#
Level: 6
Category: General

Description

In general, third party code will not comply with this coding standard, so do not put such code in the same file as code written by Philips.

Also, avoid mixing code from different Philips departments in one file, e.g., do not mix MR code with PII code. This coding standard does not specify layout rules, so code from both providers may look slightly different.

Naming conventions (Naming)

Rules

3@101	Use US-English for naming identifiers
3@102	Use Pascal and Camel casing for naming identifiers
3@103	Do not use Hungarian notation or add any other type identification to identifiers
3@104	Do not prefix member fields
3@105	Do not use casing to differentiate identifiers
3@106	Use abbreviations with care
3@107	Do not use an underscore in identifiers
3@108	Name an identifier according to its meaning and not its type
3@109	Name namespaces according to a well-defined pattern
3@110	Do not add a suffix to a `class` or `struct` name
3@111	Use a noun or a noun phrase to name a `class` or `struct`
3@112	Abbreviations with more than two letters should be cased as words
3@113	Prefix interfaces with the letter `I`
3@120	Use similar names for the default implementation of an interface
3@122	Suffix names of attributes with `Attribute`
3@201	Do not add an `enum` suffix to an enumeration type
3@202	Use singular names for enumeration types
3@203	Use a plural name for enumerations representing bitfields
3@204	Do not use letters that can be mistaken for digits, and vice versa
3@301	Add `EventHandler` to delegates related to events
3@302	Add `Callback` to delegates related to callback methods
3@303	Do not add a `Callback` or similar suffix to callback methods
3@304	Use a verb (gerund) for naming an event
3@305	Do not add an `Event` suffix (or any other type-related suffix) to the name of an event
3@306	Use an `-ing` and `-ed` form to express pre-events and post-events
3@307	Prefix an event handler with `On`
3@401	Suffix exception classes with `Exception`
3@402	Do not add code-archive related prefixes to identifiers
3@501	Name DLL assemblies after their containing namespace
3@503	Use Pascal casing for naming source files
3@504	Name the source file to the main class

Rule 3@101

Synopsis: Use US-English for naming identifiers
Language: C#
Level: 6
Category: Naming

Description

US-English means:

- magnetization, optimizing, realize, ...
- tumor, behavior, ...
- center, millimeter, ...
- ischemic, pediatric, hemodynamic, ...

Rule 3@102

Synopsis: Use Pascal and Camel casing for naming identifiers
Language: C#
Level: 9
Category: Naming

Description

In Pascal casing the first letter of each word in an identifier is capitalized, e.g., `BackColor`

In Camel casing only the first letter of the second, third, etc. word in a name is capitalized; for example, `backColor`.

The table below provides the casing for the most common types.

Identifier	Case	Example
Class	Pascal	AppDomain
Enum type	Pascal	ErrorLevel
Enum values	Pascal	FatalError
Event	Pascal	ValueChange
Exception class	Pascal	WebException
Field	camel	listItem
Const Field	Pascal	MaximumItems
Read-only Static Field	Pascal	RedValue
Interface	Pascal	IDisposable
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	camel	typeName
Property	Pascal	BackColor

For handling abbreviations see [3@112].

Rule 3@103

Synopsis: Do not use Hungarian notation or add any other type identification to identifiers

Language: C#
Level: 6
Category: Naming

Description

Use of Hungarian notation is deprecated by companies like Microsoft because it introduces a programming language-dependency and complicates maintenance activities.

Exceptions:

[3@113], [3@122], [3@301], [3@302], [3@307], [3@401].

Rule 3@104

Synopsis: Do not prefix member fields
Language: C#
Level: 9
Category: Naming

Description

Exception:

Member fields can prefixed with an "m" character.

Rule 3@105

Synopsis: Do not use casing to differentiate identifiers
Language: C#
Level: 7
Category: Naming

Description

Some programming languages (e.g. VB.NET) do not support distinguishing identifiers by case, so do not define a type called `A` and `a` in the same context.

This rule applies to namespaces, properties, methods, method parameters, and types. Please note that it **is** allowed to have identifiers that differ only in case in distinct categories, e.g. a property `BackColor` that wraps the field `backColor`.

Rule 3@106

Synopsis: Use abbreviations with care
Language: C#
Level: 10
Category: Naming

Description

Do not contract words in identifiers, but do use well-known abbreviations. For example, do not use `GetWin` instead of `GetWindow`, but **do** use a well-known abbreviation such as `UI` instead of `UserInterface`.

Rule 3@107

Synopsis: Do not use an underscore in identifiers
Language: C#
Level: 8
Category: Naming

Rule 3@108

Synopsis: Name an identifier according to its meaning and not its type
Language: C#
Level: 6
Category: Naming

Description

Avoid using language specific terminology in names of identifiers.

Example:

Do not use a definition like: void Write(double doubleValue);

Instead, use: void Write(double value);

If it is absolutely required to have a uniquely named method for every data type, use Universal Type Names in the method names. The table below provides the mapping from C# types to Universal types.

C# TYPE NAME	UNIVERSAL TYPE NAME
sbyte	`SByte`
byte	`Byte`
short	`Int16`
ushort	`UInt16`
int	`Int32`
uint	`UInt32`
long	`Int64`
ulong	`UInt64`
float	`Single`
double	`Double`
bool	`Boolean`
char	`Char`
string	`String`
object	`Object`

Based on the example above, the corresponding reading methods may look like this:

```
double ReadDouble();
long ReadInt64();
```

Rule 3@109

Synopsis: Name namespaces according to a well-defined pattern
Language: C#
Level: 8
Category: Naming

Description

Namespaces should be written in Pascal casing and named according to the following pattern:

```
<company>.<technology>.<top-level component>.<bottom-level component>
```

Rule 3@110

Synopsis: Do not add a suffix to a `class` or `struct` name
Language: C#
Level: 9
Category: Naming

Description

Do not add suffixes like `Struct` or `Class` to the name of a `class` or `struct`.

Exceptions:

[3@122] and [3@401].

Rule 3@111

Synopsis: Use a noun or a noun phrase to name a `class` or `struct`
Language: C#
Level: 8
Category: Naming

Description

Also, if the class involved is a derived class, it is a good practice to use a compound name. For example, if you have a class named `Button`, deriving from this class may result in a class named `BeveledButton`.

Rule 3@112

Synopsis: Abbreviations with more than two letters should be cased as words
Language: C#
Level: 9
Category: Naming

Description

Two-letter abbreviations in Pascal casing have both letters capitalized. In Camel casing this also holds true, except at the start of an identifier where both letters are written in lower case. With respect to capitalization in Pascal and Camel casing, abbreviations with more than two letters are treated as ordinary words.

Related to: [3@102]

Examples:

Camel Casing	Pascal Casing
newImage	NewImage
uiEntry	UIEntry
pmsMR	PmsMR
dicomType	DicomType

Rule 3@113

Synopsis: Prefix interfaces with the letter `I`
Language: C#
Level: 8
Category: Naming

Description

All interfaces should be prefixed with the letter I. Use a noun (e.g. `IComponent`), noun phrase (e.g. `ICustomAttributeProvider`), or an adjective (e.g. `IPersistable`) to name an interface.

Rule 3@120

Synopsis: Use similar names for the default implementation of an interface
Language: C#
Level: 8
Category: Naming

Description

If you provide a default implementation for a particular interface, use a similar name for the implementing class. Notice that this only applies to classes that **only** implement that interface.

Example:

A class implementing the `IComponent` interface could be called `Component` or `DefaultComponent`.

Rule 3@122

Synopsis: Suffix names of attributes with `Attribute`
Language: C#
Level: 8
Category: Naming

Description

Although this is not required by the C# compiler, this convention is followed by all built-in attributes

Rule 3@201

Synopsis: Do not add an `enum` suffix to an enumeration type
Language: C#
Level: 9
Category: Naming

Description

See also [3@103]

Rule 3@202

Synopsis: Use singular names for enumeration types
Language: C#
Level: 7
Category: Naming

Description

For example, do not name an enumeration type `Protocols` but name it `Protocol` instead. Consider the following example in which only one option is allowed.

```
public enum Protocol
{
        Tcp,
        Udp,
        Http,
        Ftp
}
```

Rule 3@203

Synopsis: Use a plural name for enumerations representing bitfields
Language: C#
Level: 7
Category: Naming

Description

Use a plural name for such enumeration types. The following code snippet is a good example of an enumeration that allows combining multiple options.

```
[Flags]
public enum SearchOptions
{
        CaseInsensitive = 0x01,
        WholeWordOnly = 0x02,
        AllDocuments = 0x04,
        Backwards = 0x08,
        AllowWildcards = 0x10
}
```

Rule 3@204

Synopsis: Do not use letters that can be mistaken for digits, and vice versa
Language: C#
Level: 7
Category: Naming

Description

To create obfuscated code, use very short, meaningless names formed from the letters `O, o, l, I` and the digits `0` and `1`. Anyone reading code like

```
bool b001 = (lo == l0) ? (I1 == 11) : (lOl != 101);
```

will marvel at your creativity.

Rule 3@301

Synopsis: Add `EventHandler` to delegates related to events
Language: C#
Level: 9
Category: Naming

Description

Delegates that are used to define an event handler for an `event` must be suffixed with `EventHandler`. For example, the following declaration is correct for a `Close` event.

```
public delegate CloseEventHandler(object sender, EventArgs arguments)
```

Rule 3@302

Synopsis: Add `Callback` to delegates related to callback methods
Language: C#
Level: 10
Category: Naming

Description

Delegates that are used to pass a reference to a callback method (so **not** an `event`) must be suffixed with `Callback`. For example:

```
public delegate AsyncIOFinishedCallback(IpcClient client, string message);
```

Rule 3@303

Synopsis: Do not add a `Callback` or similar suffix to callback methods
Language: C#
Level: 9
Category: Naming

Description

Do not add suffixes like `Callback` or `CB` to indicate that methods are going to be called through a callback delegate. You cannot make assumptions on whether methods will be called through a delegate or not. An end-user may decide to use Asynchronous Delegate Invocation to execute the method.

Rule 3@304

Synopsis: Use a verb (gerund) for naming an event
Language: C#
Level: 10
Category: Naming

Description

Good examples of events are `Closing, Minimizing`, and `Arriving`. For example, the declaration for the `Closing` event may look like this:

```
public event ClosingEventHandler Closing;
```

Rule 3@305

Synopsis: Do not add an `Event` suffix (or any other type-related suffix) to the name of an event
Language: C#

Level: 9
Category: Naming

Description

See also [3@103].

Rule 3@306

Synopsis: Use an `-ing` and `-ed` form to express pre-events and post-events
Language: C#
Level: 9
Category: Naming

Description

Do not use a pattern like `BeginXxx` and `EndXxx`. If you want to provide distinct events for expressing a point of time before and a point of time after a certain occurrence such as a validation event, do not use a pattern like `BeforeValidation` and `AfterValidation`. Instead, use a `Validat`*`ing`* and `Validate`*`d`* pattern.

Rule 3@307

Synopsis: Prefix an event handler with `On`
Language: C#
Level: 6
Category: Naming

Description

It is good practice to prefix the method that is registered as an event handler with `On`. For example, a method that handles the `Closing` event should be named `OnClosing()`.

In some situations, you might be faced with multiple classes exposing the same event name. To allow separate event handlers use a more intuitive name for the event handler, as long as it is prefixed with `On`.

Rule 3@401

Synopsis: Suffix exception classes with `Exception`
Language: C#
Level: 10
Category: Naming

Rule 3@402

Synopsis: Do not add code-archive related prefixes to identifiers
Language: C#
Level: 8
Category: Naming

Description

For example do not use code archive location (e.g. folder name) as a prefix for classes or fields. However, it is allowed to have some consistent naming scheme for related source files (e.g. belonging to a component or class hierarchy).

Rule 3@501

Synopsis: Name DLL assemblies after their containing namespace
Language: C#
Level: 8
Category: Naming

Description

To allow storing assemblies in the Global Assembly Cache, their names must be unique. Therefore, use the namespace name as a prefix of the name of the assembly. As an example, consider a group of classes organized under the namespace `Philips.PmsMR.Platform.OSInterface.` In that case, the assembly generated from those classes will be called `Philips.PmsMR.Platform.OSInterface.dll.`

If multiple assemblies are built from the same namespace, it is allowed to append a unique postfix to the namespace name.

Rule 3@503

Synopsis: Use Pascal casing for naming source files
Language: C#
Level: 9
Category: Naming

Description

Do not use the underscore character and do not use casing to differentiate names of files.

Rule 3@504

Synopsis: Name the source file to the main class
Language: C#

Level: 7
Category: Naming

Description

In addition, do not put more than one major class plus its auxiliary classes (such as `EventArgs`-derived classes) in one source file.

Exception:

If a partial class is used, then the other files for this class can be named as `MainClass.PostFix.cs`, whereby Postfix is a *meaningful* name which describes the contents and not just `MainClass.2.cs`.

Example: `MyForm.cs` and `MyForm.Designer.cs`.

Comments and embedded documentation (Comments)

Rules

4@101	Each file shall contain a header block
4@103	Use // for comments
4@105	All comments shall be written in US English
4@106	Use XML tags for documenting types and members

Rule 4@101

Synopsis: Each file shall contain a header block
Language: C#
Level: 10
Category: Comments

Description

The header block must consist of a `#region` block containing the following copyright statement and the name of the file.

```
#region Copyright Koninklijke Philips Electronics N.V. 2008
//
// All rights are reserved. Reproduction or transmission in whole or in part, in
// any form or by any means, electronic, mechanical or otherwise, is prohibited
// without the prior written consent of the copyright owner.
//
// Filename: PatientAdministration.cs
//
#endregion
```

Rule 4@103

Synopsis: Use // for comments
Language: C#
Level: 9
Category: Comments

Rule 4@105

Synopsis: All comments shall be written in US English
Language: C#
Level: 10
Category: Comments

Description

See also [3@101].

Rule 4@106

Synopsis: Use XML tags for documenting types and members
Language: C#
Level: 9
Category: Comments

Description

All public and protected types, methods, fields, events, delegates, etc. shall be documented using XML tags. Using these tags will allow IntelliSense to provide useful details while using the types. Also, automatic documentation generation tooling relies on these tags.

Section tags define the different sections within the type documentation.

SECTION TAGS	DESCRIPTION	LOCATION
<summary>	Short description	type or member
<remarks>	Describes preconditions and other additional information.	type or member
<param>	Describes the parameters of a method	method
<returns>	Describes the return value of a method	method
<exception>	Lists the exceptions that a method or property can throw	method, even or property
<value>	Describes the type of the data a property accepts and/or returns	property
<example>	Contains examples (code or text) related to a member or a type	type or member
<seealso>	Adds an entry to the See Also section	type or member
<overloads>	Provides a summary for multiple overloads of a method	first method in a overload list.

Inline tags can be used within the section tags.

INLINE TAGS	DESCRIPTION
<see>	Creates a hyperlink to another member or type
<paramref>	Creates a checked reference to a parameter

Markup tags are used to apply special formatting to a part of a section.

MARKUP TAGS	DESCRIPTION
<code>	Changes the indentation policy for code examples
<c>	Changes the font to a fixed-wide font (often used with the <code> tag)
<para>	Creates a new paragraph
<list>	Creates a bulleted list, numbered list, or a table
<b>	Bold typeface
<i>	Italics typeface

Exception:

In an inheritance hierarchy, do **not** repeat the documentation but use the `<see>` tag to refer to the base class or interface member.

Exception:

Private and nested classes do not **have** to be documented in this manner.

Object lifecycle (Object lifecycle)

Rules

5@101	Declare and initialize variables close to where they are used
5@102	If possible, initialize variables at the point of declaration
5@106	Use a public static read-only field to define predefined object instances
5@107	Set a reference field to `null` to tell the garbage collector that the object is no longer needed
5@108	Do not `shadow? a name in an outer scope
5@111	Avoid implementing a destructor
5@112	If a destructor is needed, also use `GC.SuppressFinalize`
5@113	Implement `IDisposable` if a class uses unmanaged/expensive resources or owns disposable objects
5@114	Do not access any reference type members in the destructor
5@116	Always document when a member returns a copy of a reference type or array

Rule 5@101

Synopsis: Declare and initialize variables close to where they are used
Language: C#
Level: 7
Category: Object lifecycle

Rule 5@102

Synopsis: If possible, initialize variables at the point of declaration
Language: C#
Level: 7
Category: Object lifecycle

Description

Avoid the C style where all variables have to be defined at the beginning of a block, but rather define and initialize each variable at the point where it is needed.

Rule 5@106

Synopsis: Use a public static read-only field to define predefined object instances
Language: C#
Level: 4
Category: Object lifecycle

Description

For example, consider a `Color` class/struct that expresses a certain color internally as red, green, and blue components, and this class has a constructor taking a numeric value, then this class may expose several predefined colors like this.

```
public struct Color
{
  public static readonly Color Red = new Color(0xFF0000);
  public static readonly Color Black = new Color(0x000000);
  public static readonly Color White = new Color(0xFFFFFF);

  public Color(int rgb)
  {
          // implementation
  }
}
```

Rule 5@107

Synopsis: Set a reference field to `null` to tell the garbage collector that the object is no longer needed
Language: C#
Level: 4
Category: Object lifecycle

Description

Setting reference fields to `null` may improve memory usage because the object involved will be unreferenced from that point on, allowing the garbage collector (GC) to clean-up the object much earlier. Please note that this rule does not have to be followed for a variable that is about to go out of scope.

Rule 5@108

Synopsis: Do not `shadow? a name in an outer scope
Language: C#
Level: 2
Category: Object lifecycle

Description

Repeating a name that already occurs in an outer scope is seldom intended and may be surprising in maintenance, although the behaviour is well-defined.

```
int foo = something;
?
if (whatever)
{
        double foo = 12.34;
        double anotherFoo = foo; // Violation.
}
```

Exception:

In case a method parameter has the same name as a field then the following construction can be used: `this.x = x`

```
int foo = something;
?
public void SomeMethod(int foo)
{
        this.foo = foo; // No violation
        int anotherFoo = foo; // However, this again is a violation!
}
```

Rule 5@111

Synopsis: Avoid implementing a destructor
Language: C#
Level: 4
Category: Object lifecycle

Description

If a destructor is required, adhere to Rule 5@112 and Rule 5@113.

The use of destructors in C# is demoted since it introduces a severe performance penalty due to way the garbage collector works. It is also a bad design pattern to clean up any resources in the destructor since you cannot predict at which time the destructor is called (in other words, it is non-deterministic).

Notice that C# destructors are not really destructors as in C++. They are just a C# compiler feature to represent CLR Finalizers.

Rule 5@112

Synopsis: If a destructor is needed, also use `GC.SuppressFinalize`
Language: C#
Level: 3
Category: Object lifecycle

Description

If a destructor is needed to verify that a user has called certain cleanup methods such as `Close()` on a `IpcPeer` object, call `GC.SuppressFinalize` in the `Close()` method. This ensures that the destructor is ignored if the user is properly using the class. The following snippet illustrates this pattern.

```
public class IpcPeer
{
        bool connected = false;

        public void Connect()
        {
                // Do some work and then change the state of this object.
                connected = true;
        }

        public void Close()
```

```
        {
                // Close the connection, change the state, and instruct garbage collector
                // not to call the destructor.
                connected = false;
                GC.SuppressFinalize(this);
        }

        ~IpcPeer()
        {
                // If the destructor is called, then Close() was not called.
                if (connected)
                {
                        // Warning! User has not called Close(). Notice that you can?t
                        // call Close() from here because the objects involved may
                        // have already been garbage collected (see Rule 5@113).
                }
        }
}
```

Rule 5@113

Synopsis: Implement `IDisposable` if a class uses unmanaged/expensive resources or owns disposable objects
Language: C#
Level: 2
Category: Object lifecycle

Description

If a class uses unmanaged resources such as objects returned by C/C++ DLLs, or expensive resources that must be disposed of as soon as possible, you must implement the `IDisposable` interface to allow class users to explicitly release such resources.

A class should implement the IDisposable interface, in case it creates instances of objects that implement the IDisposable interfaces and a reference to that instances is kept (note that if the class transfer ownership of the create instance to another object, then it doesn't need to implement IDisposable).

The follow code snippet shows the pattern to use for such scenarios.

```
public class ResourceHolder : IDisposable
{
        ///<summary>
        ///Implementation of the IDisposable interface
        ///</summary>
        public void Dispose()
        {
                // Call internal Dispose(bool)
                Dispose(true);

                // Prevent the destructor from being called
                GC.SuppressFinalize(this);
        }
        ///<summary>
        /// Central method for cleaning up resources
        ///</summary>
        protected virtual void Dispose(bool disposing)
        {
                // If disposing is true, then this method was called through the
                // public Dispose()
```

```
                if (disposing)
                {
                        // Release or cleanup managed resources
                }
                // Always release or cleanup (any) unmanaged resources
        }
        ~ResourceHolder()
        {
                // Since other managed objects are disposed automatically, we
                // should not try to dispose any managed resources (see Rule 5@114).
                // We therefore pass false to Dispose()
                Dispose(false);
        }
}
```

If another class derives from this class, then this class should only override the `Dispose(bool)` method of the base class. It should not implement `IDisposable` itself, nor provide a destructor. The base class?s `destructor? is automatically called.

```
public class DerivedResourceHolder : ResourceHolder
{
        protected override void Dispose(bool disposing)
        {
                if (disposing)
                {
                        // Release or cleanup managed resources of this derived
                        // class only.
                }
                // Always release or cleanup (any) unmanaged resources.
                // Call Dispose on our base class.
                base.Dispose(disposing);
        }
}
```

Rule 5@114

Synopsis: Do not access any reference type members in the destructor
Language: C#
Level: 2
Category: Object lifecycle

Description

When the destructor is called by the garbage collector, it is very possible that some or all of the objects referenced by class members are already garbage collected, so dereferencing those objects may cause exceptions to be thrown.

Only value type members can be accessed (since they live on the stack).

Rule 5@116

Synopsis: Always document when a member returns a copy of a reference type or array
Language: C#
Level: 5
Category: Object lifecycle

Description

By default, all members that need to return an internal object or an array of objects will return a reference to that object or array. In some cases, it is safer to return a copy of an object or an array of objects. In such case, **always** clearly document this in the specification.

Control flow (Control flow)

Rules

6@101	Do not change a loop variable inside a `for` loop block
6@102	Update loop variables close to where the loop condition is specified
6@103	All flow control primitives (*if, else, while, for, do, switch*) shall be followed by a block, even if it is empty
6@105	All `switch` statements shall have a `default` label as the last `case` label
6@106	An `else` sub-statement of an `if` statement shall not be an `if` statement without an `else` part
6@109	Avoid multiple or conditional `return` statements
6@112	Do not make explicit comparisons to `true` or `false`
6@115	Do not access a modified object more than once in an expression
6@118	Do not use selection statements `(if, switch)` instead of a simple assignment or initialization

Rule 6@101

Synopsis: Do not change a loop variable inside a `for` loop block
Language: C#
Level: 2
Category: Control flow

Description

Updating the loop variable within the loop body is generally considered confusing, even more so if the loop variable is modified in more than one place. This rule also applies to `foreach` loops.

Rule 6@102

Synopsis: Update loop variables close to where the loop condition is specified
Language: C#
Level: 4
Category: Control flow

Description

This makes understanding the loop much easier.

Rule 6@103

Synopsis: All flow control primitives (*if, else, while, for, do, switch*) shall be followed by a block, even if it is empty
Language: C#
Level: 3
Category: Control flow

Description

Example 1:

```
if (DoAction())
{
    result = true;
}
```

Example 2:

```
// Count number of elements in array.
for (int i = 0; i < y; i++)
{
}
```

Exceptions:

- an "else" statement may directly followed by another "if"
- An if clause, followed by a single statement, does not have to enclose that single statement in a block, provided that the entire statement is written on a single line. Of course the exception is intended for those cases where it improves readability. Please note that the entire statement must be a one-liner (of reasonable length), so it is not applicable to complex conditions. Also note that the exception is only made for if (without else), not for while etc. Examples:

  ```
  if (failure) throw new InvalidOperationException("Failure!");
  if (x < 10) x = 0;
  ```

  Rationale for the exception: code readability can be improved because the one-liner saves vertical space (by a factor of 4). The lurking danger in later maintenance, where someone might add a statement intending it to be subject to the condition, is absent in the one-liner.

Rule 6@105

Synopsis: All `switch` statements shall have a `default` label as the last `case` label
Language: C#
Level: 2
Category: Control flow

Description

A comment such as *?no action?* is recommended where this is the explicit intention. If the default case should be unreachable, an assertion to this effect is recommended.

If the default label is always the last one, it is easy to locate.

Rule 6@106

Synopsis: An `else` sub-statement of an `if` statement shall not be an `if` statement without an `else` part
Language: C#

Level: 5
Category: Control flow

Description

The intention of this rule, which applies to `else-if` constructs, is the same as in [6@105]. Consider the following example.

```
void Foo(string answer)
{
        if ("no" == answer)
        {
                Console.WriteLine("You answered with No");
        }
        else if ("yes" == answer)
        {
                Console.WriteLine("You answered with Yes");
        }
        else
        {
                // This block is required, even though you might not care of any other
                // answers than "yes" and "no".
        }
```

Rule 6@109

Synopsis: Avoid multiple or conditional `return` statements
Language: C#
Level: 9
Category: Control flow

Description

One entry, one exit is a sound principle and keeps control flow simple. However, if some cases, such as when preconditions are checked, it may be good practice to exit a method immediately when a certain precondition is not met.

Rule 6@112

Synopsis: Do not make explicit comparisons to `true` or `false`
Language: C#
Level: 9
Category: Control flow

Description

It is usually bad style to compare a `bool`-type expression to `true` or `false`.

Example:

```
while (condition == false)      // wrong; bad style
while (condition != true)       // also wrong
while (((condition == true) == true) == true)   // where do you stop?
```

```
while (condition)       // OK
```

Rule 6@115

Synopsis: Do not access a modified object more than once in an expression
Language: C#
Level: 5
Category: Control flow

Description

The evaluation order of sub-expressions within an expression **is** defined in C#, in contrast to C or C++, but such code is hard to understand.

Example:

```
v[i] = ++c;             // right
v[i] = ++i;             // wrong: is v[i] or v[++i] being assigned to?
i = i + 1;              // right
i = ++i + 1;    // wrong and useless; i += 2 would be clearer
```

Rule 6@118

Synopsis: Do not use selection statements `(if, switch)` instead of a simple assignment or initialization
Language: C#
Level: 5
Category: Control flow

Description

Express your intentions directly. For example, rather than

```
bool pos;
if (val > 0)
{
        pos = true;
}
else
{
        pos = false;
}
```

or (slightly better)

```
bool pos = (val > 0) ? true : false;
```

write

```
bool pos;
pos = (val > 0);                // single assignment
```

or even better

```
bool pos = (val > 0);   // initialization
```

Object oriented programming (Object oriented)

Rules

7@101	Declare all fields (data members) `private`
7@102	Provide a default `private` constructor if there are only `static` methods and properties on a class
7@105	Explicitly define a `protected` constructor on an `abstract` base class
7@201	Selection statements (`if-else` and `switch`) should be used when the control flow depends on an object's value; dynamic binding should be used when the control flow depends on the object's type
7@301	All variants of an overloaded method shall be used for the same purpose and have similar behavior
7@303	If you must provide the ability to override a method, make only the most complete overload virtual and define the other operations in terms of it
7@402	Use code to describe preconditions, postconditions, exceptions, and class invariants
7@403	It shall be possible to use a reference to an object of a derived class wherever a reference to that object?s base class object is used
7@501	Do not overload any `modifying? operators on a `class` type
7@502	Do not modify the value of any of the operands in the implementation of an overloaded operator
7@504	Use a `struct` when value semantics are desired
7@520	Implement the `GetHashCode` method whenever you implement the `Equals` method
7@521	Override the `Equals` method whenever you implement the `==` operator, and make them do the same thing
7@522	Override the `Equals` method any time you implement the `IComparable` Interface
7@525	Consider implementing the `Equals` method on value types
7@526	Reference types should not override the equality operator (`==`)
7@530	Consider implementing operator overloading for the equality (`==`), not equal (`!=`), less than (`<`), and greater than (`>`) operators when you implement `IComparable`
7@531	Consider overloading the equality operator (`==`), when you overload the addition (`+`) operator and/or subtraction (`-`) operator
7@532	Consider implementing all relational operators (`<`, `<=`, `>`, `>=`) if you implement any
7@601	Allow properties to be set in any order
7@602	Use a property rather than a method when the member is a logical data member
7@603	Use a method rather than a property when this is more appropriate
7@604	Do not create a constructor that does not yield a fully initialized object
7@608	Always check the result of an `as` operation

Rule 7@101

Synopsis: Declare all fields (data members) `private`
Language: C#
Level: 2
Category: Object oriented

Description

An honored principle, stated in both [C++ Coding Standard] and [MS Design].

Exceptions to this rule are `static readonly` fields and `const` fields, which may have any accessibility deemed appropriate. See also [5@106].

Rule 7@102

Synopsis: Provide a default `private` constructor if there are only `static` methods and properties on a class
Language: C#
Level: 5
Category: Object oriented

Description

Instantiating such a class is pointless.

Exceptions:

- In case the class is defined as `static`, then the private constructor is not required.
- In case the class is defined as `abstract`, then the protected constructor is required, see [7@105].

Rule 7@105

Synopsis: Explicitly define a `protected` constructor on an `abstract` base class
Language: C#
Level: 3
Category: Object oriented

Description

Of course an abstract class cannot be instantiated, so a public constructor should be harmless. However, [MS Design] states:

Many compilers will insert a `public` or `protected` constructor if you do not. Therefore, for better documentation and readability of your source code, you should explicitly define a `protected` constructor on all abstract classes.

Rule 7@201

Synopsis: Selection statements (`if-else` and `switch`) should be used when the control flow depends on an object's value; dynamic binding should be used when the control flow depends on the object's type
Language: C#
Level: 9
Category: Object oriented

Description

This is a general OO principle. Please note that it is usually a design error to write a selection statement that queries the type of an object (keywords `typeof`, `is`).

Exception:

Using a selection statement to determine if some object implements one or more optional interfaces **is** a valid construct though.

Rule 7@301

Synopsis: All variants of an overloaded method shall be used for the same purpose and have similar behavior
Language: C#
Level: 3
Category: Object oriented

Description

Doing otherwise is against the *Principle of Least Surprise.*

Rule 7@303

Synopsis: If you must provide the ability to override a method, make only the most complete overload virtual and define the other operations in terms of it
Language: C#
Level: 6
Category: Object oriented

Description

Using the pattern illustrated below requires a derived class to only override the virtual method. Since all the other methods are implemented by calling the most complete overload, they will automatically use the new implementation provided by the derived class.

```
public class MultipleOverrideDemo
{
        private string someText;

        public MultipleOverrideDemo(string s)
        {
                this.someText = s;
        }

        public int IndexOf(string s)
        {
                return IndexOf(s, 0);
        }

        public int IndexOf(string s, int startIndex)
        {
                return IndexOf(s, startIndex, someText.Length - startIndex );
        }
```

```
        // Use virtual for this one.
        public virtual int IndexOf(string s, int startIndex, int count)
        {
                return someText.IndexOf(s, startIndex, count);
        }
}
```

An even better approach, **not** required by this coding standard, is to refrain from making `virtual` methods `public`, but to give them `protected` accessibility, changing the sample above into:

```
public class MultipleOverrideDemo
{
        // same as above ...
        public int IndexOf(string s, int startIndex, int count)
        {
                return InternalIndexOf(s, startIndex, count);
        }

        // Use virtual for this one.
        protected virtual int InternalIndexOf(string s, int startIndex, int count)
        {
                return someText.IndexOf(s, startIndex, count);
        }
}
```

Rule 7@402

Synopsis: Use code to describe preconditions, postconditions, exceptions, and class invariants
Language: C#
Level: 10
Category: Object oriented

Description

Compilable preconditions etc. are testable and longer lasting than just comments.

The exact form (e.g. assertions, special Design By Contract functions such as *require* and *ensure*) is not discussed here.

Rule 7@403

Synopsis: It shall be possible to use a reference to an object of a derived class wherever a reference to that object?s base class object is used
Language: C#
Level: 3
Category: Object oriented

Description

This rule is known as the *Liskov Substitution Principle*, (see [Liskov 88]), often abbreviated to LSP. Please note that an `interface` is also regarded as a base class in this context.

Rule 7@501

Synopsis: Do not overload any `modifying? operators on a `class` type
Language: C#
Level: 6
Category: Object oriented

Description

In this context the `modifying? operators are those that have a corresponding assignment operator, i.e. the non-unary versions of +, -, *, /, %, &, |, ^, << and >>.

There is very little literature regarding operator overloading in C#. Therefore it is wise to approach this feature with some caution.

Overloading operators on a `struct` type is good practice, since it is a value type. The ,code>class is a reference type and users will probably expect reference semantics, which are not provided by most operators.

Consider a `class Foo` with an overloaded `operator+(int)`, and thus an impicitly overloaded `operator+=(int)`. If we define the function `AddTwenty` as follows:

```
public static void AddTwenty (Foo f)
{
        f += 20;
}
```

Then this function has **no** net effect:

```
{
        Foo bar = new Foo(5);
        AddTwenty (bar);
        // note that `bar? is unchanged
        // the Foo object with value 25 is on its way to the GC...
}
```

The exception to this rule is a `class` type that has complete value semantics, like `System.String.`

Rule 7@502

Synopsis: Do not modify the value of any of the operands in the implementation of an overloaded operator
Language: C#
Level: 1
Category: Object oriented

Description

This rule can be found in a non-normative clause of [C# Lang], section 17.9.1. Breaking this rule gives counter-intuitive results.

Rule 7@504

Synopsis: Use a `struct` when value semantics are desired
Language: C#
Level: 6
Category: Object oriented

Description

More precisely, a `struct` should be considered for types that meet any of the following criteria:

- Act like primitive types.
- Have an instance size under ±16 bytes.
- Are immutable.
- Value semantics are desirable.

Remember that a `struct` cannot be derived from.

Rule 7@520

Synopsis: Implement the `GetHashCode` method whenever you implement the `Equals` method
Language: C#
Level: 1
Category: Object oriented

Description

This keeps `GetHashCode` and `Equals` synchronized.

Rule 7@521

Synopsis: Override the `Equals` method whenever you implement the `==` operator, and make them do the same thing
Language: C#
Level: 1
Category: Object oriented

Description

This allows infrastructure code such as `Hashtable` and `ArrayList`, which use the `Equals` method, to behave the same way as user code written using the equality operator.

Note:

For value types, the other way around applies also, i.e., whenever you override the `Equals` method, then also also implement the equality operator.

Rule 7@522

Synopsis: Override the `Equals` method any time you implement the `IComparable` Interface
Language: C#
Level: 1
Category: Object oriented

Rule 7@525

Synopsis: Consider implementing the `Equals` method on value types
Language: C#
Level: 3
Category: Object oriented

Description

On value types the default implementation on System.ValueType will not perform as well as your custom implementation.

Rule 7@526

Synopsis: Reference types should not override the equality operator (==)
Language: C#
Level: 1
Category: Object oriented

Description

The default implementation is sufficient.

Rule 7@530

Synopsis: Consider implementing operator overloading for the equality (==), not equal (!=), less than (<), and greater than (>) operators when you implement `IComparable`
Language: C#
Level: 3
Category: Object oriented

Rule 7@531

Synopsis: Consider overloading the equality operator (==), when you overload the addition (+) operator and/or subtraction (-) operator
Language: C#
Level: 2
Category: Object oriented

Rule 7@532

Synopsis: Consider implementing all relational operators (<, <=, >, >=) if you implement any
Language: C#
Level: 2
Category: Object oriented

Rule 7@601

Synopsis: Allow properties to be set in any order
Language: C#
Level: 4
Category: Object oriented

Description

Properties should be stateless with respect to other properties, i.e. there should not be an observable difference between first setting property A and then B and its reverse.

Rule 7@602

Synopsis: Use a property rather than a method when the member is a logical data member
Language: C#
Level: 9
Category: Object oriented

Rule 7@603

Synopsis: Use a method rather than a property when this is more appropriate
Language: C#
Level: 9
Category: Object oriented

Description

In some cases a method is better than a property:

- The operation is a conversion, such as `Object.ToString`.
- The operation is expensive enough that you want to communicate to the user that they should consider caching the result.
- Obtaining a property value using the get accessor would have an observable side effect.
- Calling the member twice in succession produces different results.
- The order of execution is important. See [7@601].
- The member is `static` but returns a value that can be changed.
- The member returns a copy of an internal array or other reference type.
- Only a `set` accessor would be supplied. Write-only properties tend to be confusing.

Rule 7@604

Synopsis: Do not create a constructor that does not yield a fully initialized object
Language: C#
Level: 2
Category: Object oriented

Description

Only create constructors that construct objects that are fully initialized. There shall be no need to set additional properties. A `private` constructor is exempt from this rule.

Rule 7@608

Synopsis: Always check the result of an `as` operation
Language: C#
Level: 2
Category: Object oriented

Description

If you use `as` to obtain a certain interface reference from an object, always ensure that this operation does not return `null`. Failure to do so may cause a `NullReferenceException` at a later stage if the object did not implement that interface.

Exceptions (Exceptions)

Rules

8@101	Only throw exceptions in exceptional situations
8@102	Do not throw exceptions from unexpected locations
8@103	Only re-throw exceptions when you want to specialize the exception
8@104	List the explicit exceptions a method or property can throw
8@105	Always log that an exception is thrown
8@106	Allow callers to prevent exceptions by providing a method or property that returns the object?s state
8@107	Use standard exceptions
8@108	Throw informational exceptions
8@109	Throw the most specific exception possible
8@110	Only catch the exceptions explicitly mentioned in the documentation
8@202	Provide common constructors for custom exceptions
8@203	Avoid side-effects when throwing recoverable exceptions
8@204	Do not throw an exception from inside an exception constructor

Rule 8@101

Synopsis: Only throw exceptions in exceptional situations
Language: C#
Level: 3
Category: Exceptions

Description

Do not throw exceptions in situations that are normal or expected (e.g. end-of-file). Use return values or status enumerations instead. In general, try to design classes that do not throw exceptions in the normal flow of control. However, **do** throw exceptions that a user is not allowed to catch when a situation occurs that may indicate a design error in the way your class is used.

Rule 8@102

Synopsis: Do not throw exceptions from unexpected locations
Language: C#
Level: 1
Category: Exceptions

Description

Throwing an exception from some locations are unexpected and can cause problems. For example when you call an exception from inside a destructor, the CLR will stop executing the destructor, and pass the exception to the base class destructor (if any). If there is no base class, then the destructor is discarded.

Do not throw exceptions from the following locations:

Location	Note
Event accessor methods	The followings exceptions are allowed: System.InvalidOperationException, System.NotSupportedException and System.ArgumentException. This also includes their derivates.
Equals methods	An Equals method should return true or false. Return false instead of an exception if the arguments to not match.
GetHashCode() methods	GetHashCode() should always return a value, otherwise you lose values in a hash table.
ToString methods	This method is also used by the debugger to display information about objects in a string format. Therefore it should not raise an exception.
Static constructors	A type becomes unusable if an exception is thrown from its static constructor.
Finalizers (destructors)	Throwing an exception from a finalizer can cause a process to crash.
Dispose methods	Dispose methods are often called in finally clauses as part of cleanup. Also Dispose(false) is called from a finalizer, which in itself should not throw an exception als.
Equality Operators (==, !=)	Like the Equals methods, the operators should always return true or false.
Implicit cast operators	A user is usually unaware that an implicit cast operators is called, therefore throwing an exception from them is unexpected and should not be done.
Exception constructor	Calling a exception constructor is done to throw an exception. If the constructor throws an exception, then this is confusing.

Rule 8@103

Synopsis: Only re-throw exceptions when you want to specialize the exception
Language: C#
Level: 3
Category: Exceptions

Description

Only catch and re-throw exceptions if you want to add additional information and/or change the type of the exception into a more specific exception. In the latter case, set the `InnerException` property of the new exception to the caught exception.

Rule 8@104

Synopsis: List the explicit exceptions a method or property can throw
Language: C#
Level: 8
Category: Exceptions

Description

Describe the recoverable exceptions using the `<exception>` tag.

Explicit exceptions are the ones that a method or property explicitly throws from its implementation and which users are allowed to catch. Exceptions thrown by .NET framework classes and methods used by this implementation do not have to be listed here.

Rule 8@105

Synopsis: Always log that an exception is thrown
Language: C#
Level: 8
Category: Exceptions

Description

Logging ensures that if the caller catches your exception and discards it, traces of this exception can be recovered at a later stage.

Rule 8@106

Synopsis: Allow callers to prevent exceptions by providing a method or property that returns the object?s state
Language: C#
Level: 8
Category: Exceptions

Description

For example, consider a communication layer that will throw an `InvalidOperationException` when an attempt is made to call `Send()` when no connection is available. To allow preventing such a situation, provide a property such as `Connected` to allow the caller to determine if a connection is available before attempting an operation.

Rule 8@107

Synopsis: Use standard exceptions
Language: C#
Level: 3
Category: Exceptions

Description

The following list of exceptions are too generic and should not be raised directly by your code:

- `System.Exception`
- `System.ApplicationException`
- Any exception which is reserved for use by the CLR only (check MSDN for this)

The .NET framework already provides a set of common exceptions. The table below summarizes the most common exceptions that are available for applications.

EXCEPTION	CONDITION
IndexOutOfRangeException	Indexing an array or indexable collection outside its valid range.
InvalidOperationException	An action is performed which is not valid considering the object?s current state.
NotSupportedException	An action is performed which is may be valid in the future, but is not supported.
ArgumentException	An incorrect argument is supplied.
ArgumentNullException	A null reference is supplied as a method?s parameter that does not allow null.
ArgumentOutOfRangeException	An argument is not within the required range.

Rule 8@108

Synopsis: Throw informational exceptions
Language: C#
Level: 6
Category: Exceptions

Description

When you instantiate a new exception, set its `Message` property to a descriptive message that will help the caller to diagnose the problem. For example, if an argument was incorrect, indicate which argument was the cause of the problem. Also mention the name (if available) of the object involved.

Also, if you design a new exception class, note that it is possible to add custom properties that can provide additional details to the caller.

Rule 8@109

Synopsis: Throw the most specific exception possible
Language: C#
Level: 6
Category: Exceptions

Description

Do not throw a generic exception if a more specific one is available (related to [Rec:8@108]).

Rule 8@110

Synopsis: Only catch the exceptions explicitly mentioned in the documentation
Language: C#
Level: 1

Category: Exceptions

Description

Moreover, do not catch the base class `Exception`, `SystemException` or `ApplicationException`. Exceptions of those classes generally mean that a non-recoverable problem has occurred.

Exception:

It is allowed to catch one of the mentioned base exceptions, if it is rethrown at the end of the catch block. In order to preserve the stack details, use "`throw;`".

Example:

```
try
{
    ...
} catch (Exception e)
{
    ... // For example do some logging here or close some resource.
    throw;
}
```

Rule 8@202

Synopsis: Provide common constructors for custom exceptions
Language: C#
Level: 5
Category: Exceptions

Description

It is advised to provide the three common constructors that all standard exceptions provide as well. These include:

- `XxxException()`
- `XxxException(string message)`
- `XxxException(string message, Exception innerException)`

Rule 8@203

Synopsis: Avoid side-effects when throwing recoverable exceptions
Language: C#
Level: 1
Category: Exceptions

Description

When you throw a recoverable exception, make sure that the object involved stays in a usable and predictable state. With *usable* it is meant that the caller can catch the exception, take any necessary actions, and continue to use the object again. With *predictable* is meant that the caller can make logical assumptions on the state of

the object.

For instance, if during the process of adding a new item to a list, an exception is raised, then the caller may safely assume that the item has not been added, and another attempt to re-add it is possible.

Rule 8@204

Synopsis: Do not throw an exception from inside an exception constructor
Language: C#
Level: 1
Category: Exceptions

Description

Throwing an exception from inside an exception's constructor will stop the construction of the exception being built, and hence, preventing the exception from getting thrown. The other exception is thrown, but this can be confusing to the user of the class or method concerned.

Delegates and events (Delegates and events)

Rules

9@101	Do not make assumptions on the object's state after raising an event
9@102	Always document from which thread an event handler is called
9@103	Raise events through a protected virtual method
9@104	Use the sender/arguments signature for event handlers
9@105	Implement add/remove accessors if the number of handlers for an event must be limited
9@106	Consider providing property-changed events
9@107	Consider an `interface` instead of a `delegate`
9@108	Use delegate inference instead of explicit delegate instantiation when possible
9@110	Each subscribe must have a corresponding unsubscribe

Rule 9@101

Synopsis: Do not make assumptions on the object's state after raising an event
Language: C#
Level: 2
Category: Delegates and events

Description

Prepare for any changes to the current object's state while executing an event handler. The event handler may have called other methods or properties that changed the object?s state (e.g. it may have disposed objects referenced through a field).

Rule 9@102

Synopsis: Always document from which thread an event handler is called
Language: C#
Level: 9
Category: Delegates and events

Description

Some classes create a dedicated thread or use the Thread Pool to perform some work, and then raise an event. The consequence of that is that an event handler is executed from another thread than the main thread. For such an event, the event handler must synchronize (ensure thread-safety) access to shared data (e.g. instance members).

Rule 9@103

Synopsis: Raise events through a protected virtual method

Language: C#
Level: 9
Category: Delegates and events

Description

If a derived class wants to intercept an event, it can override such a virtual method, do its own work, and then decide whether or not to call the base class version (whether or not this should be done, is mentioned by the base class documentation). Since the derived class may decide not to call the base class method, ensure that it does not do any work required for the base class to function properly.

Name this method `OnEventName`, where *EventName* should be replaced with the name of the event. Notice that an event handler uses the same naming scheme but has a different signature. The following snippet (most parts left out for brevity) illustrates the difference between the two.

```
///<summary>An example class</summary>
public class Connection
{
    // Event definition
    public event EventHandler Closed;

    // Method that causes the event to occur
    public void Close()
    {
        // Do something and then raise the event
        OnClosed(EventArgs.Empty);
    }

    // Method that raises the Closed event.
    protected virtual OnClosed(EventArgs args)
    {
        if (Closed != null)
        {
                Closed(this, args);
        }
    }
}
///<summary>Main entrypoint.</summary>
public static void Main()
{
        Connection connection = new Connection();
        connection.Closed += new EventHandler(OnClosed);
        // For .NET 2
        // connection.Closed += OnClosed;
}
///<summary>Event handler for the Closed event</summary>
private static void OnClosed(object sender, EventArgs args)
{
    ...
}
```

Rule 9@104

Synopsis: Use the sender/arguments signature for event handlers
Language: C#
Level: 6
Category: Delegates and events

Description

The goal of this rule is to have a consistent signature for all event handlers. In general, the event handler?s signature should look like this

```
public delegate void MyEventHandler(object sender, EventArgs arguments)
```

Using the base class as the sender type allows derived classes to reuse the same event handler.

The same applies to the arguments parameter. It is recommended to derive from the .NET Framework?s `EventArgs` class and add your own event data. Using such a class prevents cluttering the event handler?s signature, allows extending the event data without breaking any existing users, and can accommodate multiple return values (instead of using reference fields). Moreover, all event data should be exposed through properties, because that allows for verification and preventing access to data that is not always valid in all occurrences of a certain event.

Note: If possible use the generic EventHandler instead of defining your own EventHandler delegate.

Rule 9@105

Synopsis: Implement add/remove accessors if the number of handlers for an event must be limited
Language: C#
Level: 8
Category: Delegates and events

Description

If you implement the `add` and `remove` accessors of an event, then the CLR will call those accessors when an event handler is added or removed. This allows limiting the number of allowed event handlers, or to check for certain preconditions.

Rule 9@106

Synopsis: Consider providing property-changed events
Language: C#
Level: 9
Category: Delegates and events

Description

Consider providing events that are raised when certain properties are changed. Such an event should be named *`Property`*`Changed`, where *Property* should be replaced with the name of the property with which this event is associated.

Rule 9@107

Synopsis: Consider an `interface` instead of a `delegate`

Language: C#
Level: 9
Category: Delegates and events

Description

If you provide a method as the target for a delegate, the compiler will only ensure that the method signature matches the delegate's signature.

This means that if you have two classes providing a delegate with the same signature and the same name, and each class has a method as a target for that delegate, it is possible to provide the method of the first class as a target for the delegate in the other class, even though they might not be related at all.

Therefore, it is sometimes better to use interfaces. The compiler will ensure that you cannot accidentally provide a class implementing a certain interface to a method that accepts another interface that happens to have to same name.

Rule 9@108

Synopsis: Use delegate inference instead of explicit delegate instantiation when possible
Language: C#
Level: 9
Category: Delegates and events

Description

Using delegate inference for subscribing to and unsubscribing from event, code can be made much more elegant than the old previous way, which was like:

```
someClass.SomeEvent += new EventHandler(OnHandleSomeEvent);
private void OnHandleSomeEvent(object sender, EventArgs e)
{...}
```

This can now be replaced by:

```
someClass.SomeEvent += OnHandleSomeEvent;
private void OnHandleSomeEvent(object sender, EventArgs e)
{...}
}
```

Note: this only applies to code written in C# 2.0 and higher.

Rule 9@110

Synopsis: Each subscribe must have a corresponding unsubscribe
Language: C#
Level: 2
Category: Delegates and events

Description

Subscribing to an event gives the object that sends the event, a reference to the subscribed object. If the subscribed object does not unsubscribe once that is not needed, then it will still be called. If for example, the

subscribed object is disposed, then the event still is called on that disposed object (which usually is not intended), and also it is not garbage collected. Therefore it is good to ensure that for each subscribe that is done, also an unsubscribe is done, once listening to that event is no longer needed. The `Dispose()` implementation could be used to ensuring that all unsubscribes are done.

Various data types (Data types)

Rules

10@201	Use an `enum` to strongly type parameters, properties, and return types
10@202	Use the default type `Int32` as the underlying type of an `enum` unless there is a reason to use `Int64`
10@203	Use the `[Flags]` attribute on an `enum` if a bitwise operation is to be performed on the numeric values
10@301	Do not use ?magic numbers?
10@401	Floating point values shall not be compared using either the `==`, `!=`, `>=`, `<=` operators and `Equals` method.
10@403	Do not cast types where a loss of precision is possible
10@404	Only implement casts that operate on the complete object
10@405	Do not generate a semantically different value with a cast
10@406	When using composite formatting, do supply all objects referenced in the format string
10@407	When using composite formatting, do not supply any object unless it is referenced in the format string

Rule 10@201

Synopsis: Use an `enum` to strongly type parameters, properties, and return types
Language: C#
Level: 6
Category: Data types

Description

This enhances clarity and type-safety. Try to avoid casting between enumerated types and integral types.

Exception:

In some cases, such as when databases or PII interfaces that store values as `int`s are involved, using `enum`s will result in an unacceptable amount of casting. In that case, it is better to use a `const int` construction.

Rule 10@202

Synopsis: Use the default type `Int32` as the underlying type of an `enum` unless there is a reason to use `Int64`
Language: C#
Level: 5
Category: Data types

Description

If the `enum` represents flags and there are currently more than 32 flags, or the `enum` might grow to that many flags in the future, use `Int64`.

Do not use any other underlying type because the Operating System will try to align an `enum` on 32-bit or 64-bit boundaries (depending on the hardware platform). Using a 8-bit or 16-bit type may result in a performance loss.

Rule 10@203

Synopsis: Use the `[Flags]` attribute on an `enum` if a bitwise operation is to be performed on the numeric values
Language: C#
Level: 7
Category: Data types

Description

It is good practice to use the `Flags` attribute for documenting that the enumeration is intended for combinations. Also using this attribute provides an implementation of the `ToString` method, which displays the values in their original names instead of the values.

Example:

```
FileInfo file = new FileInfo(fileName);
file.Attributes = FileAttributes.Hidden | FileAttributes.ReadOnly;
Console.WriteLine("file.Attributes = {0}", file.Attributes.ToString());
```

The printed result will be `ReadOnly|Hidden`.

Use an `enum` with the `flags` attribute only if the value can be completely expressed as a set of bit flags. Do not use an `enum` for open sets (such as the operating system version). Use a plural name for such an `enum`, as stated in [3@203].

Example:

```
[Flags]
public enum AccessPrivileges
{
        Read   = 0x1,
        Write  = 0x2,
        Append = 0x4,
        Delete = 0x8,
        All    = Read | Write | Append | Delete
}
```

Rule 10@301

Synopsis: Do not use ?magic numbers?
Language: C#
Level: 7
Category: Data types

Description

Do not use literal values, either numeric or strings, in your code other than to define symbolic constants. Use the following pattern to define constants:

```
public class Whatever
{
        public static readonly Color PapayaWhip = new Color(0xFFEFD5);
        public const int MaxNumberOfWheels = 18;
}
```

There are exceptions: the values `0`, `1` and `null` can nearly always be used safely. Very often the values `2` and `-1` are OK as well. Strings intended for logging or tracing are exempt from this rule. Literals are allowed when their meaning is clear from the context, and not subject to future changes.

```
mean = (a + b) / 2;                                     // okay
WaitMilliseconds(waitTimeInSeconds * 1000);             // clear enough
```

If the value of one constant depends on the value of another, do attempt to make this explicit in the code, so do **not** write

```
public class SomeSpecialContainer
{
        public const int MaxItems = 32;
        public const int HighWaterMark = 24;    // at 75%
        ...
}
```

but rather **do** write

```
public class SomeSpecialContainer
{
        public const int MaxItems = 32;
        public const int HighWaterMark = 3 * MaxItems / 4;      // at 75%
        ...
}
```

Please note that an `enum` can often be used for certain types of symbolic constants

Rule 10@401

Synopsis: Floating point values shall not be compared using either the `==`, `!=`, `>=`, `<=` operators and `Equals` method.
Language: C#
Level: 2
Category: Data types

Description

Most floating point values have no exact binary representation and have a limited precision. Use the following instead: `Math.Abs(x - y) < Single.Epsilon`

Exception:

When a floating point variable is explicitly initialized with a value such as 1.0 or 0.0, and then checked for a

change at a later stage.

Rule 10@403

Synopsis: Do not cast types where a loss of precision is possible
Language: C#
Level: 1
Category: Data types

Description

For example, do not cast a `long` (64-bit) to an `int` (32-bit), unless you can guarantee that the value of the `long` is small enough to fit in the `int`.

Rule 10@404

Synopsis: Only implement casts that operate on the complete object
Language: C#
Level: 2
Category: Data types

Description

In other words, do not cast one type to another using a member of the source type. For example, a `Button` class has a `string` property `Name`. It is valid to cast the `Button` to the `Control` (since `Button` is a `Control`), but it is not valid to cast the `Button` to a `string` by returning the value of the `Name` property.

Rule 10@405

Synopsis: Do not generate a semantically different value with a cast
Language: C#
Level: 2
Category: Data types

Description

For example, it is appropriate to convert a `Time` or `TimeSpan` into an `Int32`. The `Int32` still represents the time or duration. It does not, however, make sense to convert a file name string such as `c:\mybitmap.gif` into a `Bitmap` object.

Rule 10@406

Synopsis: When using composite formatting, do supply all objects referenced in the format string
Language: C#

Level: 1
Category: Data types

Description

Composite formatting, e.g. in String.Format, uses indexed placeholders that must correspond to elements in the list of values. A runtime exception results if a parameter specifier designates an item outside the bounds of the list of values, and we prefer not to have runtime exceptions.

Example:

```
Console.WriteLine("The value is {0} and not {1}", i);
```

where the `{1}` specifier designates a missing parameter.

Rule 10@407

Synopsis: When using composite formatting, do not supply any object unless it is referenced in the format string
Language: C#
Level: 4
Category: Data types

Description

Composite formatting, e.g. in String.Format, uses indexed placeholders that must correspond to elements in the list of values. It is not an error to supply objects in that list that are not referenced in the format string, but it very likely a mistake.

Example:

```
Console.WriteLine("The value is {0} and not {0}", i, j);
```

where the second specifier was probably intended to be `{1}` to refer to `j`.

Coding style (Coding style)

Rules

11@101	Do not mix coding styles within a group of closely related classes or within a module
11@403	The `public`, `protected`, and `private` sections of a `class` or `struct` shall be declared in that order
11@407	Write unary, increment, decrement, function call, subscript, and access operators together with their operands
11@409	Use spaces instead of tabs
11@411	Do not create overly long source lines

Rule 11@101

Synopsis: Do not mix coding styles within a group of closely related classes or within a module
Language: C#
Level: 9
Category: Coding style

Description

This coding standard gives you some room in choosing a certain style. Do keep the style consistent within a certain scope. That scope is not rigidly defined here, but is at least as big as a source file.

Rule 11@403

Synopsis: The `public`, `protected`, and `private` sections of a `class` or `struct` shall be declared in that order
Language: C#
Level: 9
Category: Coding style

Description

Although C# does not have the same concept of accessibility sections as C++, **do** group them in the given order. However, keep the fields at the top of the class (preferably inside their own `#region`). The `protected internal` section goes before the `protected` section, and the `internal` section before the `private` section.

Rule 11@407

Synopsis: Write unary, increment, decrement, function call, subscript, and access operators together with their operands
Language: C#
Level: 10
Category: Coding style

Description

This concerns the following operators:

unary:	& * + - ~ !
increment and decrement:	-- ++
function call and subscript:	() []
access:	.

It is not allowed to add spaces in between these operators and their operands.

It is not allowed to separate a unary operator from its operand with a newline.

Note: this rule does **not** apply to the **binary** versions of the & * + - operators.

Example:

```
a = -- b;       // wrong
a = --c;        // right

a = -b - c;     // right
a = (b1 + b2) +
       (c1 - c2) +
        d - e - f;      // also fine: make it as readable as possible
```

Rule 11@409

Synopsis: Use spaces instead of tabs
Language: C#
Level: 9
Category: Coding style

Description

Different applications interpret tabs differently. Always use spaces instead of tabs. You should change the settings in Visual Studio .NET (or any other editor) for that.

Rule 11@411

Synopsis: Do not create overly long source lines
Language: C#
Level: 8
Category: Coding style

Description

Long lines are hard to read. Many applications, such as printing and difference views, perform poorly with long lines. A maximum line length of 80 characters has proven workable for C and C++. However, C# tends to be more verbose and have deeper nesting compared to C++, so the limit of 80 characters will often cause a statement to be split over multiple lines, thus making it somewhat harder to read. This standard does not set any explicit limit on the length of a source line, thus leaving the definition of `too long? to groups or projects.

Literature

C# Lang
Title: C# Language Specification
Author: TC39/TG2/TG3
Year: 2001
Publisher: ecma
ISBN: ECMA-334
http://www.ecma-international.org/publications/standards/Ecma-334.htm

C++ Coding Standard
Title: Philips Healthcare C++ Coding Standard
Author: Philips Healthcare CCB Coding Standards
Year: 2008
Publisher: Philps Healthcare
http://tics/codingstandards/CPP/viewer

Liskov 88
Title: Data Abstraction and Hierarchy
Author: Barbara Liskov
Year: 1988
Publisher: SIGPLAN Notices, 23,5 (May, 1988)
http://portal.acm.org/citation.cfm?id=62141

MS Design
Title: Design Guidelines for Developing Class Libraries
Author: Microsoft, MSDN
http://msdn.microsoft.com/en-us/library/ms229042(VS.80).aspx

Meyer 88
Title: Object Oriented Software Construction
Author: Betrand Meyer
Year: 1988
Publisher: Prentice Hall