

Coding Standards for C# Language

|  |  |
| --- | --- |
| **Project Title** | Internal |
| **Author** | M. Yousuf Bin Azhar |
| **Version** | 1.0 |
| **Confidentiality** | Internal |
| **Date** | December 2011 |
| **Distribution** | Internal |

Document History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Version | Author | Issued to | Comments |
| 21/12/06 | 0.1 | RH | Internal | Initial Document |
| 22/11/2011 | 0.1 | YBA | Internal | Revised |

**Contents**

[1. Introduction 3](#_Toc309744415)

[1.1 Importance of Coding Standards 3](#_Toc309744416)

[1.2 Confidentiality 3](#_Toc309744417)

[2. Terminology & Definitions 4](#_Toc309744418)

[2.1.1. Pascal Casing 4](#_Toc309744419)

[2.1.2. Camel Casing 4](#_Toc309744420)

[3. C# Coding Standards 5](#_Toc309744421)

[3.1 General Guidelines 5](#_Toc309744422)

[3.2 Naming Conventions 6](#_Toc309744423)

[3.2.1. Project / Solution & Assembly Files 6](#_Toc309744424)

[3.2.2. Source Files 6](#_Toc309744425)

[3.2.3. Namespaces 6](#_Toc309744426)

[3.2.4. Classes / Structs 6](#_Toc309744427)

[3.2.5. Generic Classes 7](#_Toc309744428)

[3.2.6. Interfaces 7](#_Toc309744429)

[3.2.7. Methods 7](#_Toc309744430)

[3.2.8. Properties 7](#_Toc309744431)

[3.2.9. Constants 7](#_Toc309744432)

[3.2.10. Variables 7](#_Toc309744433)

[3.2.10.1. Variable Scope Prefix 7](#_Toc309744434)

[3.2.10.2. Variable Type Prefix 8](#_Toc309744435)

[3.2.10.3. Variable Name 8](#_Toc309744436)

[3.3 Coding Style 8](#_Toc309744437)

[3.3.1. Formatting 8](#_Toc309744438)

[3.3.2. Code Commenting 9](#_Toc309744439)

[3.4 Source File Structure 9](#_Toc309744440)

[3.4.1. Guidelines 9](#_Toc309744441)

[3.4.2. Example 10](#_Toc309744442)

# Introduction

This document contains general Coding Standards for C# language. After the technology shift to.NET Framework, more & more work is being done in C# language. ASP.NET, in-process assemblies, COM+ components, Windows Services & Desktop applications are all being written using .Net languages.

As the Agilosoft code-base that contains C# code is increasing, there is a need to define and follow a common coding convention so that the code looks more consistent & readable to everyone.

Standards must be defined in such a way that they are easy to follow and convey maximum amount of information to a reviewer looking at the code without complicating matters for the developers writing the code. It is important to remember that the code could be reviewed outside the Visual Studio IDE (e.g. when reviewing code-behind files on deployment machine, external reviewers may use simple text editors) therefore the standards should specify, for example, differences between variables with respect to their usage.

## Importance of Coding Standards

Coding standards are important for many reasons. First and foremost, they specify a common format for the source code and comments. This allows developers to produce maintainable code, easily share code, and the ideas expressed within the code and comments, between each other. Instead of each developer coding how they like, they will write all code so they maintain the standards outlined in this document. This makes sure that a large project is coded in a consistent style - parts are not written differently depending on who was doing the programming. Not only does this makes the code easier to understand but ensures that anyone else who looks at the code knows what to expect throughout the entire application. It also specifies how comments (internal documentation) should be handled. More importantly, a well designed standard will also detail how certain code should be written, not just how it looks on screen.

## Confidentiality

This document and its contents is the property of the Agilosoft (the company). This document discusses the proposed standards for C# Language. All technology discussed in this document is the intellectual property of the company and protected under law.

Distribution of this document is provided through written consent of the company. If you wish to distribute this document, please obtain written permission to do so.

# Terminology & Definitions

### Pascal Casing

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters. For example: BackColor

### Camel Casing

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example: backColor

# C# Coding Standards

This section contains Coding Standards and Guidelines for the Microsoft C# language.

## General Guidelines

* Think about the best structure of the program before you start coding
* Concentrate on optimal algorithms, readable and maintainable code
* Do not duplicate the same code. If you notice that you're writing the same code sequence more than once in your program, move it to a separate function
* Common code/constants between different modules/classes should be moved into a separate common module
* Code must not contain unused variables and parts of code that is not accessible
* Unused code should not be commented out but should be removed (preferably check in such a code into VSS/ TFS and then remove the code)
* Code must not contain anti-performance parts. For example avoid calling the same function returning the same result. Instead, call the function once and store the result in a variable
* Select the control structure properly
* Write secure code:
  + Protect against Buffer Overruns
  + Protect against SQL injections. For example when coding “’SELECT \* FROM operators WHERE operator\_id = ‘ + input” make sure that ‘input’ does not contain injected code, such as “a’; DROP TABLE customer”
  + Prevent In-line Scripting
* Write your application such that it does not require administrative rights
* If operations can fail, check for failure and act appropriately. For example always validate user input to handle all possibilities
* Use assertions properly.
  + Use System.Diagnostics.Debug.Assert() method in C#
  + Do not use assertions for error handling
  + Use assertions carefully because they can open security holes and undermine the runtime's mechanism for enforcing security restrictions
* Errors/Exceptions are raised where required and must be handled properly
* Code inside the exception handler should be simple and non-complex. This is to minimize the possibility of the exception handler raising further exceptions
* Don't neglect logging, tracing, error handling and reporting. Report any information that might help the developer to find the cause of an error. E.g., report “Couldn't find foo.conf, looked in /etc, /home/user, /usr/lib/foo/” instead of simply “couldn't open file”. However, don't use assertions for error handling
* Scope of variables and functions should be as reduced as possible. Avoid global data and functions whenever possible
* Use namespaces in favor of prefixes
* Use double in favor of float by default
* Use signed integers by default
* Avoid using literals in the code and use constants instead. Ideally, there should not be any literal numbers and strings in the code, especially when a number or string is being used many times
* Avoid Magic Numbers. These are numeric literal used within an expression (or to initialize a variable) that does not have an obvious or well-known meaning. This usually excludes the integers 0 or 1
* Avoid spaces, underscores and non-alphabetic characters in file names
* Do not use underscore as word separator (except for constants). Use capitalisation to separate multiple words (e.g. use BusinessLayer instead of Business\_Layer)
* Use what's available, including third-party libraries if policies permit and if you can manage the dependencies. Don't reinvent the wheel. Look in the Standard Library for the code that has already been developed and tested
* Allocate and de-allocate dynamically allocated memory blocks properly to avoid memory leaks
* If sizeof() is needed use sizeof(variable) in preference to sizeof(typeof\_variable). This helps protect against bugs when type is changed

## Naming Conventions

### Project / Solution & Assembly Files

* Always use Pascal Case
* Match Project / Solution name with Assembly and Root Namespace (e.g. Project File: Agilosoft.RetailProduct.DataLayer.csproj, Assembly File: Agilosoft.RetailProduct.DataLayer.dll, Root Namespace: Agilosoft.RetailProduct.DataLayer)
* Project / Solution file also derives the default name of the resulting binary file (dll, exe) so names like Agilosoft.ProductName.ProductComponent is an acceptable name for project / solution file.
* Choose names for your assembly DLLs that suggest large chunks of functionality such as Agilosoft.ProductName.System.Data
* Assembly and DLL names do not have to correspond to namespace names but it is reasonable to follow the namespace name when naming assemblies
* Consider naming DLLs according to the following pattern:

<Company>.<Component>.dll

where <Component> contains one or more dot-separated clauses. For example Agilosoft.WebControls.dll

* Avoid spaces, underscores and non-alphabetic characters in file names

### Source Files

* Always use Pascal Case
* Match containing class name
* Avoid more than one class in a single source file

### Namespaces

Use the following pattern:

<Company>.<ProductName>.<Component>

Where:

* Company: Usually it should be Agilosoft. However it could be the name of the client company (e.g. HH or BMG).
* ProductName: Name of the product like HRPayroll or BMT
* Component: One or more dot-separated clauses that best conveys the description of the component e.g. DataAccess or AS2.Server
* Use Pascal Case for each of the parts described above

### Classes / Structs

* Use Pascal Case
* Do not use a prefix (e.g. “C” for classes)
* Use a suffix according to the following table

|  |  |  |
| --- | --- | --- |
| **Suffix** | **Use when Deriving from / Designing a** | **Examples** |
| Array | Array | UserNameArray |
| Attribute | Attribute class | SerializableAttribute |
| Collection | Collection class | DataSetCollection |
| EventArgs | EventArgs class | CacheUpdateEventArgs |
| Exception | Exception class | WebException |
| Map | Hashtable class (key-value pair class) | UserRoleMap |

### Generic Classes

* Use Pascal Case
* Use a single capital letter as the generic parameter type (e.g. T or K)
* Avoid designing generic classes with more than two generic parameter types

### Interfaces

* Always use the prefix “I”
* Use Pascal Case

### Methods

* Use Pascal Case
* Use a Verb or Verb-Object Pair (e.g. CleanUpSessions()).
* Avoid using the returned data as the names of the functions (e.g. use GetUserRoles() instead of UserRoles())
* Do not use uncommon abbreviations or short forms (e.g. DeleteAllCompanyProperties() should be used instead of DelAllCompProp())
* Do not use names that differ only slightly (e.g. GetRole() and GetRoles())
* Prefix event handlers with ‘On’. For example OnClick()
* Use the prefix ‘Is**’**, ‘Has’, or ‘Can’ for methods that return a Boolean value. For example IsVisible(), IsOpen(), CanEvaluate()

### Properties

* Use Pascal Case
* Never prefix property names with Get or Set

### Constants

* Use Upper Case
* Use underscore (\_) as the word separator (e.g. MAX\_TYPE\_NAME, COLOR\_RED)

### Variables

* Use camel case for naming a variable.
* Do not use a separator to separate the three parts of the naming pattern (e.g. do not use underscore)
* Avoid uncommon abbreviations and short forms; use long descriptive names

## Coding Style

Coding style causes the most inconsistency and controversy between developers. Each developer has a preference, and rarely are two the same. However, consistent layout, format, and organization are keys to creating maintainable code. The following sections describe the preferred way to implement C# source code in order to create readable, clear, and consistent code that is easy to understand and maintain.

### Formatting

* Never declare more than 1 namespace per file
* Avoid putting multiple classes in a single file
* Always place curly braces ({ and }) on a new line
* Always use curly braces ({ and }) in conditional statements
* Always use a Tab and Indention size of 4
* Append folder-name to namespace for source files within sub-folders
* Recursively indent all code blocks contained within braces
* Use white space (CR/LF, Tabs, etc) liberally to separate and organize code
* Avoid declaring multiple attribute declarations within a single line. Instead stack each attribute as a separate declaration
* Place Assembly scope attribute declarations on a separate line
* Place Type scope attribute declarations on a separate line
* Place Method scope attribute declarations on a separate line
* Place Member scope attribute declarations on a separate line
* Place Parameter attribute declarations inline with the parameter

### Code Commenting

* All comments should be written in U.S. English
* Use // or /// but do not use /\* … \*/
* Do not “flowerbox” comment blocks; following is a bad example of commenting code blocks

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Code comments …

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* Use inline-comments to explain assumptions, known issues, and algorithm insights
* Do not use inline-comments to explain obvious code. Well written code is self documenting
* Only insert inline-comments for Bad Code to say “fix this code” – otherwise, rewrite it!
* Include Task-List keyword flags to enable comment-filtering e.g.:

// TODO: Implement exception handling for this function

// UNDONE: Remove P/Invoke calls due to memory errors

// HACK: Temporary fix until a solution is found

* Only use C# comment block (///) for documenting the API
* Always include <summary> comments. Include <param>, <return> and <exception> where appropriate
* Include <see cref=”” /> and <seeAlso cref=”” /> where possible
* Always add CDATA tags to comments containing code and other embedded markup in order to avoid encoding issues

## Source File Structure

### Guidelines

* Place namespace “using” statements together at the top of file
* Group .NET namespaces above custom namespaces
* Group class implementation by type in the following order:
  + Constants
  + Static member variables
  + Instance (non-static) member variables
  + Constructors
  + Finalizers
  + Properties
  + Methods
  + Nested Enums, Structs, and Classes
  + Interface implementations
* Sequence declarations within type groups based upon access modifier and visibility:
  + Private
  + Internal
  + Protected
  + Public
* Use regions (#region)to separate different parts of a class implementation
* Declare each variable independently; not in the same statement
* Use access modifiers explicitly; do not rely on default access modifiers
* Segregate interface Implementation by using #region statements
* Use finalizers (destructors) only when required e.g. when implementing “Dispose Pattern” (see MSDN for details).
* Do not make use of comments to mark trivial changes and very small fixes
* Use Visual SourceSafe to identify code changes between two versions

### Example

#region Modification History

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Module : CodingStandards.cs

// Created By : M. Yousuf Bin Azhar (yousuf)

// Created On : 31/12/2006

// Description : Class to demonstrate Agilosoft Coding Standards for C# language

//

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Modification History \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// No. Who Date Description

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 01 rhasnani 01/01/2007 Added another constructor

// 02 msaleem 04/01/2007 Used Interlocked.Increment for static member to work correctly

// for multi-threading cases

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#endregion

using System;

using System.Diagnostics;

using System.Globalization;

using System.Threading;

using System.Xml;

using Agilosoft.BusinessLayer;

using Agilosoft.DataAccess;

namespace Agilosoft.Standards.Coding

{

/// <summary>

/// This class demonstrates Agilosoft Coding Standards for C# language

/// <para>Here's how you could make a second paragraph in a description.</para>

/// <see cref="System.Console.WriteLine(System.String)"/> for help on writing to console window.

/// <seealso cref="CodingStandards.ExampleMethod"/>

/// </summary>

public class CodingStandards

{

#region Constants

public const int STANDARDS\_COUNT = 10;

#endregion

#region Static Member Variables

private static int \_classInstances;

#endregion

#region Instance Member Variables

private string \_instanceName;

private bool \_instanceActive;

private DateTime m\_lastAccessTime;

#endregion

#region Constuctors

static CodingStandards()

{

\_classInstances = 0;

}

public CodingStandards()

{

InitializeInstance();

}

// BOC rhasnani(01)

public CodingStandards(CultureInfo roCultureInfo)

{

InitializeInstance();

}

// EOC rhasnani(01)

#endregion

#region Finalizers

~CodingStandards()

{

}

#endregion

#region Properties

public bool IsActive

{

get

{

return \_isInstanceActive;

}

}

public DateTime LastAccessTime

{

get

{

return \_lastAccessTime;

}

set

{

\_lastAccessTime = value;

}

}

#endregion

#region Methods

private void InitializeInstance()

{

\_instanceName = "";

\_isInstanceActive = true;

\_lastAccessTime = DateTime.Now;

Interlocked.Increment(ref ClassInstances); // msaleem(02)

}

private bool IsInstanceActive(int index)

{

return true;

}

internal int AnInternalMethod()

{

return 0;

}

protected string AProtectedMethod()

{

return \_instanceName;

}

public int ExampleMethod()

{

int returnValue;

returnValue = 0;

if (!\_isInstanceActive)

return returnValue;

else

{

for (int index = 0; index < classInstances; index++)

{

if (IsInstanceActive(index))

{

returnValue++;

}

}

}

return returnValue;

}

#endregion

#region Enumerators

#endregion

#region Nested Structures

#endregion

#region Nested Classes

#endregion

#region Interface Implementations

#region ICloneable Implementation

#endregion

#region IComparable Implementation

#endregion

#endregion

}

}