MS Solutions Team

 C#.Net Code Conventions

Version 1.01

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 11/07/03 | 1.0 | Initial version | Yaroslav Popov |
| 12/10/03 | 1.01 | Discussed in the team and updated | Yaroslav Popov |
|  |  |  |  |

Table of Contents

[1. Introduction 4](#_Toc59259482)

[2. Names 4](#_Toc59259483)

[2.1 Namespaces 4](#_Toc59259484)

[2.2 Classes 5](#_Toc59259485)

[2.3 Interfaces 5](#_Toc59259486)

[2.4 Methods and Functions 5](#_Toc59259487)

[2.5 Properties, fields, parameters and variables 5](#_Toc59259488)

[2.6 Structures 6](#_Toc59259489)

[2.7 Events and delegates 6](#_Toc59259490)

[2.8 Files 6](#_Toc59259491)

[3. Comments 6](#_Toc59259492)

[4. Format 7](#_Toc59259493)

[References 9](#_Toc59259494)

# Introduction

The goal of this document is to provide a standard coding technique for C#.Net projects hold by the members of MS Solutions team. The techniques defined here are not proposed to form an inflexible set of coding standards. They are rather meant to serve as a guide for developing a coding standard for a specific software project.

# Names

The naming rules have key importance to understanding the logic of the application.

* The main general rule is: “A name should tell ‘what’ rather then ‘how’” [1].
* Make names long enough to be meaningful but short enough to avoid verbosity.
* Provide names that a reader can comprehend.
* Avoid redundant class names while naming properties and methods (List.ListItem should be named as List.Item).
* Components must be fully usable from both case-sensitive and case-insensitive languages. Therefore don’t create namespaces, class names, property names, method names, parameters names etc. that differ only by case.
* Avoid using class names that duplicate commonly used .NET Framework namespaces, such as: System, Collections, Forms, UI, etc.

## Namespaces

The general rule for naming namespaces is to use the company name followed by the technology name and optionally the feature and design as follows.

CompanyName.TechnologyName[.Feature][.Design]

For example:

Baseshop.Media

Baseshop.Media.Design

* Use a stable, recognized technology name at the second level of a hierarchical name. Use organizational hierarchies as the basis for namespace hierarchies. For example, the System.Windows.Forms.Design namespace contains designers and related classes used to design System.Windows.Forms based applications.
* A nested namespace should have a dependency on types in the containing namespace. For example, the classes in the System.Web.UI.Design depend on the classes in System.Web.UI. However, the classes in System.Web.UI do not depend on the classes in System.UI.Design.
* You should use Pascal case[[1]](#footnote-1) for namespaces, and separate logical components with periods, as in Microsoft.Office.PowerPoint. If your brand employs nontraditional casing, follow the casing defined by your brand, even if it deviates from the prescribed Pascal case.
* Use plural namespace names if it is semantically appropriate. For example, use System.Collections rather than System.Collection. Exceptions to this rule are brand names and abbreviations. For example, use System.IO rather than System.IOs.
* Do not use the same name for a namespace and a class.

## Classes

* Use a noun or concatenated nouns to name a class. Use Pascal case.
* Do not use a type prefix, such as C for class, on a class name. For example, use the class name FileStream rather than CFileStream.
* Do not use the underscore character (\_).

## Interfaces

* Name interfaces with nouns or concatenated nouns, or adjectives that describe behavior. For example, the interface name IComponent uses a descriptive noun. The interface name ICustomAttributeProvider uses concatenated nouns. The name IPersistable uses an adjective. Use Pascal case.
* Prefix interface names with the letter I, to indicate that the type is an interface.

## Methods and Functions

* Names of methods and functions must be constructed by concatenating several words. Use Pascal case to name public and protected methods: CalculateTotal(). Use camel case[[2]](#footnote-2) to name private functions: getAttribute().
* Use verb-noun construction to name a function or method.
* Avoid names that are open to subjective interpretation: OpenThis.
* Avoid creating method bodies of a very big size (generally not more than 25 - 50 lines of code). Use private functions to break down the business logic into sub-modules.

## Properties, fields, parameters and variables

* Property name begins with capital letter: JobId (Pascal case).
* Use a read-only property when the user cannot change the property's logical data member. Do not use write-only properties.
* Field name begins with small letter. If a field is a property's logical data member, it gets the same name as the property (if property name is JobId the field name for this property is jobId).
* Do not use fields that are public. A field can be declared as public for a global constant (instead of using property with get accessor). In the last case capitalize the name of the field.
* Interface classes should not have protected fields.
* Local variable names and parameter names begins with small letter. Use camel case.
* Name of a constant is capitalized. The concatenated words are separated by “\_” sign: APPLICATION\_NAME.
* Boolean property/field/variable/parameter name should start with prefix “is”, “has” or “does” which implies True/False values (doesFileExist instead of fileExists).
* Even for short-lived local variables still use a meaningful name. Use single-character names: ‘i’, ‘j’, ’k’, ’l’, ’m’, ’n’ for loops variables; ‘x’, ‘y’, ‘z' for coordinates; ‘r’, ‘g’, ‘b’ for colors, ‘e’ for event argument.
* Consider using named constants in conditions instead of numbers. The exceptions are 0, 1 and –1. ( *for i=0; i<NUM\_DAYS\_IN\_WEEK; i++* instead of *for i=0; i<7; i++*  ). However it is possible to use number if it is used in one place only and this does not make sense to define it as a constant. In the case a number is used in condition a comment has to be added to explain meaning of the number used.
* Avoid using hard coded strings for messages that are displayed to user. Use a named constant, a database record or resource file item instead.
* While building strings for custom messages use formatted strings:

MES\_DELETE = "File {0} deleted.";

. . .

res = String.Format(MES\_DELETE, drawFile.Name);

## Structures

* Structures can include public fields.
* Do not create structure with 2 or more constructors. Use classes instead.

## Events and delegates

* Use an EventHandler suffix on delegates names.
* Specify two parameters named sender and e. The sender parameter represents the object that raised the event. The sender parameter is always of type object, even if it is possible to use a more specific type. The state associated with the event is encapsulated in an instance of an event class named e. Use an appropriate and specific event class for the e parameter type.
* Name an event argument class with the EventArgs suffix.
* Consider naming events with a verb.
* Do not use a prefix or suffix in the event declaration. For example, use Close instead of OnClose.
* Function name that is passed to the delegate has to be named as following: name of the object that has raised the event, underscore symbol, name of the raised event. For example: Circle\_ColorChanged.

public delegate void ColorEventHandler( object sender, ShapeEventArgs e);

public event ColorEventHandler ColorChanged;

Circle1.ColorChanged += new Shape.ColorEventHandler(Circle1\_ColorChanged);

## Files

* Each class/interface has to be placed into a separate file. The file name has to be the same as the class/interface name it contains.
* A file may contain several classes in the case of nested classes or if just one class is considered to be inherited from.

# Comments

While developing in C# use XML Documentation feature, described in [2].

Few general rules on commenting:

* At the beginning of every construction part of code (class, method, property, function or protected field declaration, etc.) use “<summary>” XML commenting tag [2] (in order to generate the commenting tags automatically type “///” in the beginning of the construction part of code). In the case of method or function provide the description of parameters and return value in the corresponding tags. While commenting code construction provide text indicating the purpose, assumptions and limitations of the block.

/// <summary>

/// Checks that stored procedure exists in the database

/// </summary>

/// <param name="stProcName">Name of the stored procedure</param>

/// <returns>true if the procedure exists</returns>

public bool CheckSPExists (string stProcName)

{

………

}

* Use complete sentences when writing comments.
* Comment as you code or right after finishing a certain logical portion of code (method, function etc.).
* When modifying code, always keep the commenting up to date.
* Avoid adding comments at the end of a line of code except when annotating local variable declarations. In this case use // for commenting.
* In the code body comment anything that is not readily obvious.
* Use comments on important loops and logic branches.
* Comment all private field declarations. Use // for commenting.

# Format

* Establish a standard size for an indent and use it consistently through the project. The default indent is the default tab size (4 space characters).
* Line of code has to be less than 80 characters long.
* Align open and close braces vertically where brace pairs align:

for (i = 0; i < NUM\_OBJECTS; i++)

{

...

}

* Indent code along lines of logical construction:

if (reportId != BaseTable.INVALID\_PK)

{

try

{

recReport = RepManager.GetRecordByPK(reportId);

}

catch (Exception ex)

{

HandleException(ex);

}

}

else

{

recReport = new RecReports();

}

* In switch statements use indenting on two levels:

switch (e.Button.Text)

{

case "Triangle" :

Triangle triangle1 = new Triangle("Triangle1", true);

IPointy iptTr = (IPointy) triangle1;

Triangle1.Draw();

break;

case "Hexagon" :

Hexagon hexagon1 = new Hexagon("Hexagon1", false);

IPointy iptTr = (IPointy) hexagon1;

hexagon1.Draw();

break;

}

* Use spaces before and after most operators (except ++, --, ?…:) : isDisposing = false instead of isDisposing=false.
* Avoid placing more than one statement per line (except ++, --, ?…:).
* Use line breaks and empty lines to create “paragraphs” of code to separate the logical segments. Use line breaks and empty lines before code commenting segments.
* Constant variables group initialization code and enumeration fields initialization code has to be lined up on the equality sign:

public const int DBVERSION = 4;

public const int TINYINT\_OWERFLOW = 8115;

public const int TRIGGER\_EXCEPT = 50000;

public enum StatusMode

{

Planned = 1,

Active = 2,

InActive = 4,

All = 7

};

* In case of long statement, break it to several lines and use double indenting in next lines. For example:

if (Member.Address.Room != null && Member.Address.Room != "" &&

(Member.Address.Sect > 0 || Member.Address.BuildNo > 0))

Member.Address.Normalize();

* In case of long statement, break it in accordance with logical code structure. It is allowed to add additional indent to indicate level of parts in expression’s structure. Following example is wrong formatting:

if (Address.Room != null && Address.Room != "" && (Address.Sect

> 0 || ((Address.BuildNo != null && Address.BuildNo !=

"")?Address.BuildNo:DEFAULT\_BUILDING\_NO) > 0) &&

Address.IsNotPrepared)

Member.Address.Normalize();

Following example is correct:

if (Address.Room != null && Address.Room != "" &&

(Address.Sect > 0 ||

((Address.BuildNo != null && Address.BuildNo != "")?

Address.BuildNo:DEFAULT\_BUILDING\_NO) > 0) &&

Address.IsNotPrepared)

Member.Address.Normalize();

## References

1. Coding techniques

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconcodingtechniques.asp>

2. C# XML Documentation

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vcoriXMLDocumentation.asp>

1. In Pascal case each concatenated word begins with capital letter: CustromAttribute. [↑](#footnote-ref-1)
2. In camel case each concatenated word begins with capital letter except the first: custromAttribute. [↑](#footnote-ref-2)