C# Code Style Guide

Version 1.2

Scott Bellware

80% of the lifetime cost of a piece of software goes to maintenance.

Hardly any software is maintained for its whole life by the original author.

Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

Your source code is a product; you need to make sure it is as well-packaged and clean.

Introduction

Superior coding techniques and programming practices are hallmarks of a professional programmer. The bulk of programming consists of making a large number of small choices while attempting to solve a larger set of problems. How wisely those choices are made depends largely upon the programmer's skill and expertise.

This document addresses some fundamental coding techniques and provides a collection of coding practices.

The readability of source code has a direct impact on how well a developer comprehends a software system, which in turn directly affects project velocity. Code maintainability refers to how easily that software system can be changed to add new features, modify existing features, fix bugs, or improve performance. Although readability and maintainability are the result of many factors, one particular facet of software development upon which all developers have an influence is coding technique. The easiest method to ensure that a team of developers will yield quality code is to establish a coding standard, which is then enforced at routine code reviews. Although the primary purpose for conducting code reviews throughout the development life cycle is to identify defects in the code, the reviews can also be used to enforce coding standards in a uniform manner.

A comprehensive coding standard encompasses all aspects of code construction and, while developers should exercise prudence in its implementation, it should be closely followed. Completed source code should reflect a harmonized style, as if a single developer wrote the code in one session.

Style Guide

Source File Organization

One Class per File

Source files should contain one class definition per source file. Said differently, each class definition will exist within its own file. The stem of the file name must be the same name as the name used in the class declaration. For example, the class definition for a class named Loan will have a file name of Loan.cs.

Ordering

C# source files have the following ordering:

- `using` statements
- `namespace` statement
- Class and interface declarations

Namespace and Using Statements

The first non-comment lines of most C# source files is the `using` statements. After that, `namespace` statements can follow. For example:

```
using System.Data;

namespace Business.Framework;
```

Both the `using` statement and the `namespace` statement are aligned flush against the left margin.

The first letter of a component in a namespace is always capitalized. If the namespace name is an acronym, the first letter only of the namespace will be capitalized, as in `System.Data.Sql`. If the acronym only has two letters, both letters are capitalized, as in `System.IO`.

XML Documentation

Visual Studio provides for a type of documentation that the development environment is able to detect and extract to structured XML that is used to create code-level documentation that exists outside of the source code itself.

XML documentation is provided for class descriptions, methods, and properties. XML documentation should be used in all circumstances where it's available.

Refer to the detailed discussion on XML documentation in this document as well as in the documents provided with Visual Studio .NET.

Class and Interface Declaration

Sequence	Part of Class/Interface Declaration	Notes

1	Class/interface documentation	`/// <summary>`
		`/// The Person class provides ...`
		`/// </summary>`
		`public class Person`

2	`class` or `interface`	

	statement	
3	Fields	First `private`, then `protected`, then `internal`, and then `public`.
4	Properties	First `private`, then `protected`, then `internal`, and then `public`.
4	Constructors	First `private`, then `protected`, then `internal`, and then `public`. Default first, then order in increasing complexity.
5	Methods	Methods should be grouped by functionality rather than by scope or accessibility. For example a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

Indentation

Indentation is constructed with tabs, not spaces. Typically, tabs are set to be displayed as white space with a width of four characters.

Line Length

Optimizing for down level tools and editors such as Notepad should not impact code style. 80 character lines are a recommendation, not a hard and fast rule.

Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

- Break after an operator.
- Break after a comma.
- Prefer higher-level breaks to lower-level breaks.
- Indent once after a break.

Here is an example of breaking a method call:

```
SomeMethod1(longExpression1, someMethod2(longExpression2,
    longExpression3)); // Note: 1 indent start second line.
```

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 - longName5) +
    4 * longname6; // PREFER
```

```
longName1 = longName2 * (longName3 + longName4
    - longName5) + 4 * longname6; // AVOID
```

Following is an example of indenting method declarations:

```
SomeMethod( int anArg,
    Object anotherArg,
    String yetAnotherArg,
    Object andStillAnother)
{
    ...
}
```

Line wrapping for `if` statements should use the indent rule. For example:

```
// USE THIS INDENTATION
if ((condition1 && condition2) ||
    (condition3 && condition4) ||
    !(condition5 && condition6))
{
    DoSomethingAboutIt();
}

// OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4) ||
    !(condition5 && condition6))
{
    DoSomethingAboutIt();
}
```

Here are two acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression ? beta : gamma);

alpha = (aLongBooleanExpression ?
    beta :
    gamma);
```

Comments

C# programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by /*...*/, and //. Documentation comments are C# only, and are delimited by special XML tags that can be extracted to external files for use in system documentation.

Implementation comments are meant for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective, to be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding component is built or in what directory it resides should not be included as a comment. Discussion of nontrivial or obscure design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Following are recommended commenting techniques:

- When modifying code, always keep the commenting around it up to date.
- Comments should consist of complete sentences and follow active language naming responsibilities (*Adds the element* instead of *The element is added*).
- At the beginning of every routine, XML documentation is used to indicate the routine's purpose, assumptions, and limitations. A boilerplate comment should be a brief introduction to understand why the routine exists and what it can do. Refer to the detailed discussion on XML documentation in this document as well as in the document provided with Visual Studio .NET.

- Avoid adding comments at the end of a line of code; end-line comments make code more difficult to read. However, end-line comments are appropriate when annotating variable declarations. In this case, align all end-line comments at a common tab stop.
- Avoid using clutter comments, such as an entire line of asterisks. Instead, use white space to separate comments from code. XML documentation serves the purpose of delineating methods.
- Avoid surrounding a block comment with a typographical frame. It may look attractive, but it is difficult to maintain.
- Prior to deployment, remove all temporary or extraneous comments to avoid confusion during future maintenance work.
- If you need comments to explain a complex section of code, examine the code to determine if you should rewrite it. If at all possible, do not document bad code – rewrite it. Although performance should not typically be sacrificed to make the code simpler for human consumption, a balance must be maintained between performance and maintainability.
- Use complete sentences when writing comments. Comments should clarify the code, not add ambiguity.
- Comment as you code, because most likely there won't be time to do it later. Also, should you get a chance to revisit code you've written, that which is obvious today probably won't be obvious six weeks from now.
- Avoid the use of superfluous or inappropriate comments, such as humorous sidebar remarks.
- Use comments to explain the intent of the code. They should not serve as inline translations of the code.
- Comment anything that is not readily obvious in the code. This point leads to allot of subjective interpretations. Use your best judgment to determine an appropriate level of what it means for code to be not really obvious.
- To prevent recurring problems, always use comments on bug fixes and work-around code, especially in a team environment.
- Use comments on code that consists of loops and logic branches. These are key areas that will assist the reader when reading source code.
- Separate comments from comment delimiters with white space. Doing so will make comments stand out and easier to locate when viewed without color clues.
- Throughout the application, construct comments using a uniform style, with consistent punctuation and structure.
- Comments should never include special characters such as form-feed and backspace.

Implementation Comment Formats

C# syntax provides for many styles of code comments. For simplicity and based on the heuristic use of comments in C#, we will use comments traditionally reserved for end of line comments and code disabling for all cases of code comments.

Block Comments

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

A blank line to set it apart from the rest of the code should precede a block comment.

```
// Here is a block comment
```

```
// that breaks across multiple
// lines.
```

Single-Line Comments

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

```
if (condition)
{
    // Handle the condition.
    ...
}
```

Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Here's an example of a trailing comment in C# code:

```
if (a == 2)
{
    return true; // Special case
}
else
{
    return isPrime(a); // Works only for odd a
}
```

Code-Disabling Comments

The `//` comment delimiter can comment out a complete line or only a partial line. Code-disabling comment delimiters are found in the first position of a line of code flush with the left margin. Visual Studio .NET provides for bulk commenting by selecting the lines of code to disable and pressing CTRL+K, CTRL+C. To uncomment, use the CTRL+K, CTRL+U chord.

The following is an example of code-disabling comments:

```
if (foo > 1)
{
    // Do a double-flip.
    ...
}
else
{
    return false; // Explain why here.
}

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

Documentation Comments

C# provides a mechanism for developers to document their code using XML. In source code files, lines that begin with /// and that precede a user-defined type such as a class, delegate, or interface; a member such as a field, event, property, or method; or a namespace declaration can be processed as comments and placed in a file.

XML documentation is required for classes, delegates, interfaces, events, methods, and properties. Include XML documentation for fields that are not immediately obvious.

The following sample provides a basic overview of a type that has been documented.

```
// XmlSample.cs
using System;

/// <summary>
/// Class level summary documentation goes here.
/// </summary>
/// <remarks>
/// Longer comments can be associated with a type or member
/// through the remarks tag.
/// </remarks>
public class SomeClass
{
    /// <summary>
    /// Store for the name property.
    /// </summary>
    private string name;


    /// <summary>
    /// Name property.
    /// </summary>
    /// <value>
    /// A value tag is used to describe the property value.
    /// </value>
    public string Name
    {
        get
        {
            if (this.name == null)
            {
                throw new Exception("Name is null");
            }

            return myName;
        }
    }


    /// <summary>
    /// The class constructor.
    /// </summary>
    public SomeClass()
    {
        // TODO: Add Constructor Logic here
    }


    /// <summary>
    /// Description for SomeMethod.
    /// </summary>
    /// <param name="s">Parameter description for s goes here.</param>
    /// <seealso cref="String">
    /// You can use the cref attribute on any tag to reference a type
    /// or member
    /// and the compiler will check that the reference exists.
    /// </seealso>
    public void SomeMethod(string s) {}
```

```
    /// <summary>
    /// Some other method.
    /// </summary>
    /// <returns>
    /// Return results are described through the returns tag.
    /// </returns>
    /// <seealso cref="SomeMethod(string)">
    /// Notice the use of the cref attribute to reference a specific
    /// method.
    /// </seealso>
    public int SomeOtherMethod()
    {
        return 0;
    }


    /// <summary>
    /// The entry point for the application.
    /// </summary>
    /// <param name="args">A list of command line arguments.</param>
    public static int Main(String[] args)
    {
        // TODO: Add code to start application here
        return 0;
    }
}
```

XML documentation starts with `///`. When you create a new project, the wizards put some starter `///` lines in for you. The processing of these comments has some restrictions:

- The documentation must be well-formed XML. If the XML is not well-formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.
- Developers are not free to create their own set of tags.
- There is a recommended set of tags.
- Some of the recommended tags have special meanings:
    - The <param> tag is used to describe parameters. If used, the compiler will verify that the parameter exists and that all parameters are described in the documentation. If the verification failed, the compiler issues a warning.
    - The cref attribute can be attached to any tag to provide a reference to a code element. The compiler will verify that this code element exists. If the verification failed, the compiler issues a warning. The compiler also respects any using statements when looking for a type described in the cref attribute.
    - The <summary> tag is used by IntelliSense inside Visual Studio to display additional information about a type or member.

If you need to give information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment or single-line comment immediately *after* the declaration.

Document comments must not be positioned inside a method or constructor definition block, because C# associates documentation comments with the first declaration *after* the comment.

Here are the XML documentation tags available:

Tag	Notes
<c>	The <c> tag gives you a way to indicate that text within a description should be marked as code. Use <code> to indicate multiple lines as code.
<code>	The <code> tag gives you a way to indicate multiple lines as code. Use <c> to

	indicate that text within a description should be marked as code.
<example>	The <example> tag lets you specify an example of how to use a method or other library member. Commonly, this would involve use of the <code> tag.
<exception>	The <exception> tag lets you document an exception class. Compiler verifies syntax.
<include>	The <include> tag lets you refer to comments in another file that describe the types and members in your source code. This is an alternative to placing documentation comments directly in your source code file. The <include> tag uses the XML XPath syntax. Refer to XPath documentation for ways to customize your <include> use. Compiler verifies syntax.
<list>	The <listheader> block is used to define the heading row of either a table or definition list. When defining a table, you only need to supply an entry for term in the heading. Each item in the list is specified with an <item> block. When creating a definition list, you will need to specify both term and text. However, for a table, bulleted list, or numbered list, you only need to supply an entry for text. A list or table can have as many <item> blocks as needed.
<para>	The <para> tag is for use inside a tag, such as <remarks> or <returns>, and lets you add structure to the text.
<param>	The <param> tag should be used in the comment for a method declaration to describe one of the parameters for the method. Compiler verifies syntax.
<paramref>	The <paramref> tag gives you a way to indicate that a word is a parameter. The XML file can be processed to format this parameter in some distinct way. Compiler verifies syntax.
<permission>	The <permission> tag lets you document the access of a member. The System.Security.PermissionSet lets you specify access to a member.
<remarks>	The <remarks> tag is where you can specify overview information about a class or other type. <summary> is where you can describe the members of the type.
<returns>	The <returns> tag should be used in the comment for a method declaration to describe the return value.
<see>	The <see> tag lets you specify a link from within text. Use <seealso> to indicate text that you might want to appear in a See Also section. Compiler verifies syntax.
<seealso>	The <seealso> tag lets you specify the text that you might want to appear in a See Also section. Use <see> to specify a link from within text.
<summary>	The <summary> tag should be used to describe a member for a type. Use <remarks> to supply information about the type itself.
<value>	The <value> tag lets you describe a property.

Comment Tokens - TODO, HACK, UNDONE

When you add comments with comment tokens to your code, you automatically add shortcuts to the Task List window. Double-click any comment displayed in the Task List to move the insertion point directly to the line of code where the comment begins.

Note: Comments in HTML, .CSS, and .XML markup are not displayed in the Task List.

To add a comment hyperlink to the Task List window, enter the comment marker. Enter TODO, HACK, or UNDONE. Add the Comment text.

```
// TODO Fix this method.

// HACK This method works but needs to be redesigned.
```

A hyperlink to your comment will appear in the Task List in the Visual Studio development environment.

Declarations

Number Per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
private int level = 2; // indentation level
private int size = 8;  // size of table
```

is preferred over

```
private int level, size; // AVOID!!!
```

Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first. For instance, if you declare an int without initializing it and expect a public method of the owning class to be invoked that will act on the in, you will have no way of knowing if the int was properly initialized for it was used. In this case declare the int and initialize it with an appropriate value.

Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
public void SomeMethod()
{
    int int1 = 0;         // Beginning of method block.

    if (condition)
    {
        int int2 = 0;     // Beginning of "if" block.
        ...
    }
}
```

The one exception to the rule is indexes of `for` loops, which in C# can be declared in the `for` statement:

```
for (int i = 0; i < maxLoops; i++)
{
    // Do something
}
```

Class and Interface Declarations

When coding C# classes and interfaces, the following formatting rules should be followed:

- No space between a method name and the parenthesis "(" starting its parameter list
- Open brace "{" appears at the beginning of the line following declaration statement and is indented to the beginning of the declaration.
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement. For null statements, the "}" should appear immediately after the "{" and both braces should appear on the same line as the declaration with 1 blank space separating the parentheses from the braces:

```
public class Sample : Object
{
    private int ivar1;
    private int ivar2;

    public Sample(int i, int j)
    {
        this.ivar1 = i;
        this.ivar2 = j;
    }

    protected void EmptyMethod() {}
}
```

- Methods are separated by two blank lines.

Properties

If the body of the `get` or `set` method of a property consists of a single statement, the statement is written on the same line as the method signature. White space is inserted between the property method (get, set) and the opening brace. This will create visually more compact class definitions..

```
public int Foo
{
    get { return this.foo; }
    set { this.foo = value; }
}
```

instead of

```
public int Foo
{
    get
    {
        return this.foo;
    }
    set
    {
        this.foo = value;
    }
}
```

Statements

Simple Statements

Each line should contain at most one statement. Example:

```
argv++;        // Correct
argc--;        // Correct
argv++; argc--;        // AVOID!
```

Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "`{ statements }`". See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the beginning of the line following the line that begins the compound statement and be indented to the beginning of the compound statement. The closing brace should begin a line and be indented to the beginning of the compound statement.

- Braces are used around all statements, even single statements, when they are part of a control structure, such as a `if-else` or `for` statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

return Statements

A `return` statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;
```

```
return myDisk.size();
```

```
return (size ? size : defaultSize);
```

if, if-else, if else-if else Statements

The `if-else` class of statements should have the following form:

```
if (condition)
{
    statements;
}

if (condition)
{
    statements;
}
else
{
    statements;
}

if (condition)
{
    statements;
}
else if (condition)
{
    statements;
}
else
{
    statements;
}
```

Note: `if` statements always use braces `{}`. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
    statement;
```

for Statements

A `for` statement should have the following form:

```
for (initialization; condition; update)
{
    statements;
}
```

while Statements

A `while` statement should have the following form:

```
while (condition)
{
    statements;
}
```

An empty `while` statement should have the following form:

```
while (condition);
```

do-while Statements

A `do-while` statement should have the following form:

```
do
{
    statements;
} while (condition);
```

switch Statements

A `switch` statement should have the following form:

```
switch (condition)
{
    case 1:
        // Falls through.
    case 2:
        statements;
        break;
    case 3:
        statements;
        goto case 4;
    case 4:
        statements;
        break;
    default:
        statements;
        break;
}
```

When there is no code between two cases and there is no `break` statement, the code falls though. If case 1 is satisfied, the code for case 2 will execute. If code is present between two cases, and a fall through is desired, a `goto case` statement is required, as in case 3. This is done so that errors aren't introduced by inadvertently omitting a `break`.

Every time a case falls through (doesn't include a `break` statement), add a comment where the `break` statement would normally be.

Every `switch` statement should include a default case. The `break` in the default case is redundant, but it prevents a fall-through error if later another `case` is added.

try-catch Statements

A `try-catch` statement should have the following format:

```
try
{
    statements;
}
catch (ExceptionClass e)
{
    statements;
}
```

A `try-catch` statement may also be followed by `finally`, which executes regardless of whether or not the `try` block has completed successfully.

```
try
{
    statements;
}
catch (ExceptionClass e)
{
    statements;
}
finally
{
    statements;
}
```

White Space

Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.
One blank line should always be used in the following circumstances:

- Between the local variables in a method and its first statement
- Between logical sections inside a method to improve readability
- After the closing brace of a code block that is not followed by another closing brace.

Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example:

```
while (true)
{
    ...
}
```

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except “.” should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

```
a += c + d;
a = (a + b) / (c * d);

while (d < n)
{
    n++;
}

this.PrintSize("size is " + foo + "\n");
```

- The expressions in a `for` statement should be separated by blank spaces. Example:

```
for (expr1; expr2; expr3)
```

Naming Rules

Naming rules make programs more understandable by making them easier to read. They can also give information about the function of the identifier, for example, whether it's a constant, package, or class, which can be helpful in understanding the code.

Perhaps one of the most influential aids to understanding the logical flow of an application is how the various elements of the application are named. A name should tell "what" rather than "how." By avoiding names that expose the underlying implementation, which can change, you preserve a layer of abstraction that simplifies the complexity. For example, you could use `GetNextOrder()` instead of `GetNextArrayElement()`.

A tenet of naming is that difficulty in selecting a proper name may indicate that you need to further analyze or define the purpose of an item. Make names long enough to be meaningful but short enough to avoid being wordy. Programmatically, a unique name serves only to differentiate one item from another. Expressive names function as an aid to the human reader; therefore, it makes sense to provide a name that the human reader can comprehend. However, be certain that the names chosen are in compliance with the applicable rules and standards.

Following are recommended naming techniques:

Methods

- Names of methods should contain active verb forms and imperatives (`DeleteOrder`, `OpenSocket`). It is not necessary to include the noun name when the active verb refers directly to the containing class. Example:

```
Socket.OpenSocket(); // AVOID! No need to mention "Socket" in name

Socket.Open(); // PREFER
```

- Avoid elusive names that are open to subjective interpretation, such as `Analyze()` for a routine, or `xxK8` for a variable. Such names contribute to ambiguity more than abstraction.
- Use the verb-noun method for naming routines that perform some operation on a given object, such as `CalculateInvoiceTotal()`.
- In method overloading, all overloads should perform a similar function.

Variables

- Do not use any special prefix characters to indicate that the variable is scoped to the class, as in `_name` or `m_name`. Always use the "this" keyword when referring to members at a class's root scope from within a lower level scope, as in `this.name`.
- Do not use Hungarian notation for field names. Good names describe semantics, not type.
- Prepend computation qualifiers (avg, sum, min, max, index) to the beginning of a variable name where appropriate.
- Use customary opposite pairs in variable names, such as min/max, begin/end, and open/close.
- In object-oriented languages, it is redundant to include class names in the name of class properties, such as `Book.BookTitle`. Instead, use `Book.Title`.
- Collections should be named as the plural form of the singular objects that the collection contains. A collection of `Book` objects is named `Books`.
- Boolean variable names should contain "Is" or "is" which implies Yes/No or True/False values, such as `isFound`, or `isSuccess`.

- Avoid using terms such as `Flag` when naming status variables, which differ from Boolean variables in that they may have more than two possible values. Instead of `orderFlag`, use a more descriptive name such as `orderStatus`.
- Even for a short-lived variable that may appear in only a few lines of code, still use a meaningful name. Use single-letter variable names, such as `i` or `j` for short-loop indexes only.
- Constants should *not* be all uppercase with underscores between words, such as `NUM_DAYS_IN_WEEK`. Constants follow the same naming rules as properties. The aforementioned constant would be named `NumDaysInWeek`.
- Temporary variables should always be used for one purpose only; otherwise, several variables should be declared.

Parameters

- Do not prefix method parameters with any special character to indicate that they are parameters.
- Parameter names should follow the naming rules for variables (above).

Tables

- When naming tables, express the name in the singular form. For example, use `Employee` instead of `Employees`.
- When naming columns of tables, do not repeat the table name; for example, avoid having a field called `EmployeeLastName` in a table called `Employee`.
- Do not incorporate the data type in the name of a column. This will reduce the amount of work needed should it become necessary to change the data type later.

Microsoft SQL Server

- Do not prefix stored procedures with `sp_`, because this prefix is reserved for identifying system-stored procedures.
- In Transact-SQL, do not prefix variables with `@@`, which should be reserved for truly global variables such as `@@IDENTITY`.

General

- Implementation details, in particular type specifications, should not be mention in the name of a descriptor. This is a common trait in procedural languages like Visual Basic where lowercase prefixes are used to encode the data type in the name of the identifier, such as `oInvoice`. This approach is not applicable to contemporary languages where the aforementioned identifier is written simply as `invoice`.
- Names with semantic content are preferred to names with type specifications (`sizeOfArray` instead of `anInteger`).
- Names of descriptors should be chosen in such a way that they can be read like a sentence within instructions.
- Minimize the use of abbreviations. If abbreviations are used, be consistent in their use. An abbreviation should have only one meaning and likewise, each abbreviated word should have only one abbreviation. For example, if using *min* to abbreviate *minimum*, do so everywhere and do not later use it to abbreviate *minute*.
- When naming methods, include a description of the value being returned, such as `GetCurrentCustomerName()`.

- File and folder names, like method names, should accurately describe what purpose they serve.
- Avoid homonyms when naming elements to prevent confusion during code reviews, such as *write* and *right*.
- When naming elements, be aware of commonly misspelled words. Also, be aware of differences that exist between American and British English, such as color/colour and check/cheque.

Abbreviations

To avoid confusion and guarantee cross-language interoperation, follow these rules regarding the use of abbreviations:

- When using acronyms, use camel case for acronyms more than two characters long. For example, use HtmlButton. However, you should capitalize acronyms that consist of only two characters, such as `System.IO` instead of `System.Io`.
- Do not use abbreviations in identifiers or parameter names.
- Do not use abbreviations or contractions as parts of identifier names. For example, use `GetWindow` instead of `GetWin`.
- Do not use acronyms that are not generally accepted in the computing field.

Where appropriate, use well-known acronyms to replace lengthy phrase names. For example, use UI for User Interface and OLAP for On-line Analytical Processing.

Capitalization

Use the following three conventions for capitalizing identifiers:

Pascal case

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: **B**ack**C**olor.

Camel case

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example: **b**ack**C**olor.

Uppercase

All letters in the identifier are capitalized. Use this convention only for identifiers that consist of two or fewer letters. For example: `System.IO`, `System.Web.UI`.

You might also have to capitalize identifiers to maintain compatibility with existing, unmanaged symbol schemes, where all uppercase characters are often used for enumerations and constant values. In general, these symbols should not be visible outside of the assembly that uses them.

The following table provides rules and example for common identifiers:

Identifier Type	Rules for Naming	Examples
Namespaces	Namespace names should be nouns, in Pascal case. Avoid the use of	`MyCompany.Framework.Data;` `MyCompany.Factories;`

	underscores ("_") in namespace names. Try to keep names simple and descriptive. Use whole words and avoid acronyms and abbreviations unless the abbreviation is much more widely used than the long form, such as Url or Html. All custom namespace names are to begin with the company name if applicable, followed by the project, product, or technology name, and the purpose name, followed by the purpose of the package as an organizational unit.	

Classes	Class names should be nouns, in Pascal case. As with namespaces, keep class names simple and descriptive. Use whole words and avoid acronyms and abbreviations unless the abbreviation is much more widely used than the long form, such as Url or Html.	`class SalesOrder;`
		`class LineItem;`
		`class HtmlWigets;`

Interfaces	Interface names use Pascal case and begin with the letter "I".	`interface IBusinessRule;`
Methods	Methods should be active verb/nouns forms, in Pascal case.	`GetDataRow();` `UpdateOrder();`

Instance Fields	Instance fields are in camel case. Variable names should not start with underscore, even though it is allowed.
	
	Variable names should be meaningful. The choice of a variable name should be mnemonic – that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided.
		`protected string name;`
		
		`private int orderId;`
		`private string lastName;`
		`private float width;`
		
		`// AVOID!`
		`private _total;`

Enum Types and Enum Values	Enum types and values are in Pascal case. Use abbreviations sparingly. Do not use an `Enum` suffix on Enum type names. Use a singular name for most Enum types, but use a plural name for Enum types that are bit fields.	`enum Status {ReadyToGo,`
		`WaitingForNow};`
		
		`enum Day {Monday, Tuesday};`

Events	Events are in Pascal case. Use the suffix "EventHandler" on event handler 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	`public delegate void`
		`MouseEventHandler(object sender,`
		`MouseEventArgs e);`

	class for the *e* parameter type. Name an event argument class with the EventArgs suffix. Consider naming events with a verb. Use a gerund (the "ing" form of a verb) to create an event name that expresses the concept of pre-event, and a past-tense verb to represent post-event. For example, a Close event that can be canceled should have a `Closing` event and a `Closed` event. Do not use the `BeforeXxx/AfterXxx` naming pattern. Do not use a prefix or suffix on the event declaration on the type. For example, use `Close` instead of `OnClose`. In general, you should provide a protected method called `OnXxx` on types with events that can be overridden in a derived class. This method should only have the event parameter *e*, because the sender is always the instance of the type.	
Exception Classes	Exception classes are in Pascal case and always have the suffix "Exception".	`InvalidCastException` `DomainValueException`
Custom Attributes	Custom attributes are in Pascal case and always have the suffix "Attribute".	`PersistentEntityAttribute`
Properties	Properties are in Pascal case with an. Property names should directly reflect the underlying attribute.	`public int OrderId` `public string LastName`

Object References	Objects references are camel case when non-public and pascal case when public (although references should never be public without good reason). Except where exceptionally warranted, objects are named after their class. An exception to this rule would be when two or more objects are needed of the same class within the same scope. Avoid naming object references as abbreviations or acronyms of the class name.	`public string Name;`
		
		`Order order = new Order;`
		`LineItem lineItem =`
		`new LineItem();`

Constants	Constants are in Pascal case. They should not be all uppercase with words separated by underscores ("_").	`const int NumDaysInWeek = 4;`
		
		`// AVOID!`
		`const int NUM_DAYS_IN_WEEK = 4;`

Practices

Design Rules and Heuristics

- Design coherent operations, that is, operations that fulfill only one task.
- Do without side effects: do not work with global variables and the like in your operations. Pass this kind of information as arguments, instead.
- A subclass should support all attributes, operations, and relations of its superclass; suppressions of these properties should be avoided.
- A subclass should not define constraints on inherited properties of its superclass.
- If inherited operations need to be overwritten, they should be compatible with the behavior of the overwritten ones.
- Aim for even distribution of knowledge about the application domain across all classes
- Design your concepts to be as general as possible. That is, design with a view to interfaces instead of implementation.
- Design client-server relationships between classes (cooperation principal).
- Minimize dependencies between classes.
- The superclass of an abstract class is itself an abstract class too.
- Maximize the internal binding of classes. Responsibilities which belong together should be concentrated in one class.
- Minimize the external dependencies of a class. Keep the number of contracts (interfaces) with other classes to a minimum.
- Instead of functional modes, different operations should be provided. Where functional modes are used, use enum types as mode discriminators.
- Avoid indirect navigation. Limit the knowledge of classes about their neighboring classes.
- The code for one operation should not exceed one page. Lengthy operations are the hallmark of procedural programming. If an operation is lengthy, it is likely that it is a candidate for redesign.
- Mind uniform and descriptive names, data types, and parameter orders.
- If in an operation you find `switch`/`case` instructions or several consecutive instructions, this is a symptom of procedural thinking (polymorphism phobia).
- Take extreme values into account (minimum, maximum, nil, nonsense) and plan a robust behavior in all situations.
- Try to do without artificial or arbitrary limits (for example, a list with 14 entries) and try to implement dynamic behavior.
- It is never too early to start thinking about undo functions, user-specific configurations, user access right concepts, error handling, etc.
- Take company-specific and general standards into account.
- Avoid data-heavy and data-driven design. Behavior driven design has advantages over purely data-driven design. In data driven design, few central control classes emerge, but a high overall coupling of classes. In behavior-driven design the tasks are more equally divided across the classes, significantly fewer messages are generated, and the classes are coupled more loosely.

Providing Access to Instance and Class Variables

Don't make any instance or class variables public or protected without good reason. Often, instance variables don't need to be explicitly set or gotten. Use a public or protected property instead.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a `struct` instead of a class, then it's appropriate to make the class's instance variables public.

Use the `this` keyword when referencing instance fields in methods. The reader will be able to immediately differentiate between variables scoped to the method and those scoped to the object.

Literals

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler. Example:

```
d = (a = b + c) + r;    // AVOID!
```

should be written as:

```
a = b + c;
d = a + r;
```

Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others-you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d)     // AVOID!
if ((a == b) && (c == d)) // RIGHT
```

Parameters

Check for valid parameter arguments. Perform argument validation for every public or protected method and property set accessor. Throw meaningful exceptions to the developer for invalid parameter arguments. Use the System.ArgumentException Class, or a class derived from System.Exception.

Constructor parameters used to initialize instance fields should have the same name as the instance field. Example:

```
public class Foo
{
    private string fooId;


    public foo(string fooId)
    {
        this.fooId = fooId;
    }
}
```

Returning Values

Try to make the structure of your program match the intent. Example:

```
if (booleanExpression)
```

```
{
    return true;
}
else
{
    return false;
}
```

should instead be written as:

```
return booleanExpression;
```

Similarly,

```
if (condition)
{
    return x;
}

return y;
```

should be written as:

```
return (condition ? x : y);
```

Avoid excessive nesting using *guard clause*

Just as indentation increasing the readability of code, it also contributes to ambiguity. Nesting happens when one control structure exists within another control structure, and possibly even another control structure. When reading code that resides within many nested blocks, the programmer must maintain an awareness of the pre-conditions that lead to the code being executed. Although a compiler is especially gifted at maintaining a stack of unresolved control structures, programmers are less so. Nesting becomes increasingly more ambiguous near the end of the control structures where for example code can be executed at the conclusion of an `if` block and prior to the conclusion of another `if` block.

```
public SomeMethod()
{
    for (int i = 1, i < 100, i++)
    {
        if (i > 10)
        {
            ...    // Do something
            if (arg == someNumber)
            {
                ...  // Do more
            }
        }
    }
}
```

Nesting becomes even more unreadable when code inside the structure stretches on for many lines or, in extreme situations, up to a page. Using the complement of the conditional expression leads to an early resolution of control structures and a flattening of the nesting.

```
if (i > 10)
```

becomes:

```
if (i <= 10)
```

A `continue` would be executed in order to start back at the top of the `for` block. This achieves a flattening of the nests, and a conditional block is resolved as quickly as possible.

```
public SomeMethod()
{
    for (int i = 1, i < 100, i++)
    {
        // Guard
        if (i <= 10)
        {
            continue;
        }

        ...    // Do something

        if (arg == someNumber)
        {
            ...  // do more
        }
    }
}
```

Note: The Extract Method refactoring combined with the Consolidate Conditional Expression work well to support the flattening of indentation in code blocks as well.

Debug Code

Debug code should not be stripped from the source base. If debug code significantly contributes to the understanding and the maintenance of the code, then leave the debug code inside the class definition. The compiler will strip the debug code from the DLL's when a class is compiled for production.

Refactoring

Refactoring is a technique to restructure code in a disciplined way. Refactoring follows a set of rules. These rules are named and published in a catalog in a similar fashion to Design Patterns.

Refactoring is not only a way to repair old code, or to make existing code more flexible, but it is also a way to write new code based on a system of best practices. Not all refactorings are useful at the outset, but many are, and knowledge of the techniques is invaluable.

The online version of the refactoring catalog can be found at http://www.refactoring.com/catalog/index.html.

Conclusion

Using solid coding techniques and good programming practices to create high quality code plays an important role in software quality. In addition, by consistently applying a well-defined coding standard and proper coding techniques, and holding routine code reviews, a team of programmers working on a software project is more likely to yield a software system that is easier to comprehend and maintain.