Standard Coding Guidelines

Variable naming and coding style guidelines for open source code development

James Ritchie Carroll

3/30/2011

Table of Contents

Introduction

Coding standards can significantly reduce the cost of software development for organizations with many developers. Standards make code easier to maintain and can improve the readability of code across all projects in an organization, both of which keep development costs lower. This document describes coding practices followed by the Grid Protection Alliance and the Tennessee Valley Authority open source code distributions which are applied to all public code for consistency, readability and maintainability.

The standards and naming conventions defined in this document assume C#.NET as the development language and use the .NET Framework Design Guidelines as a starting point.

Disclaimer: Some of the information in this document has been pulled together from various different sources, it should not be considered a completely original work.

Definitions

Camel case is a casing convention where the first letter is lower-case, words are not separated by any character but have their first letter capitalized. In this standard *camelCase* is used for any local variable and parameter name. Example: `thisIsCamelCased`.

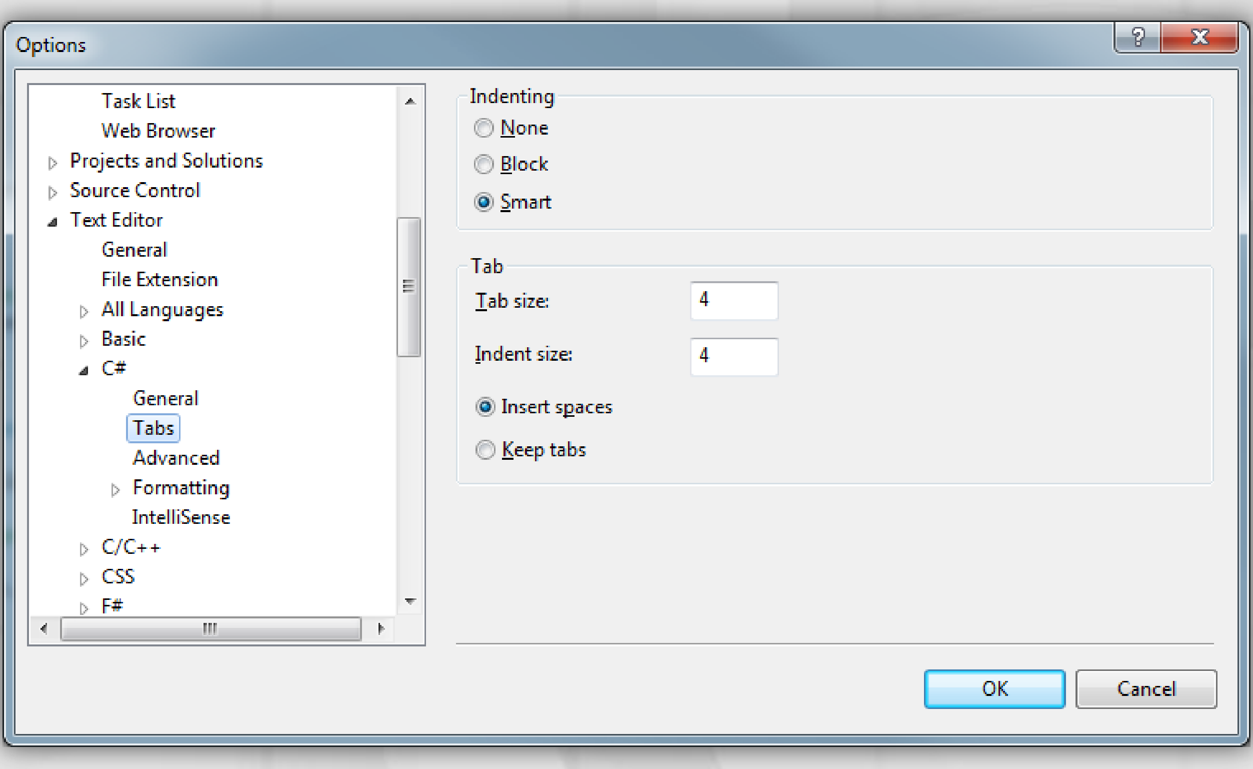
Pascal case, also known as upper camel case, is a casing convention where the first letter of each word is capitalized, and no separating character is included between words. In this standard *PascalCase* is used for any public or protected method or property name. Example: `ThisIsPascalCased`.

Hungarian notation is an identifier naming convention in computer programming, in which the name of a variable or function indicates its type or intended use. In this standard, Hungarian notation is *avoided*. Example: `strName`.

Style Guidelines

Tabs & Indenting

Tab characters (`\0x09`) should not be used in code. All indentation should be done with 4 space characters. Converting the Tab key to spaces can be set automatically inside Visual Studio:



Indenting
None
Block
[✓ Smart]

Tab
Tab size: 4
Indent size: 4
[✓ Insert spaces]
Keep tabs

Bracing

Open braces should always be at the beginning of the line after the statement that begins the block. Contents of the brace should be indented by 4 spaces. For example:

```
if (type == typeof(bool))
{
    // Handle booleans as a special case to allow numeric entries as well as true/false
    return (T)((object)value.ParseBoolean());
}
else
{
    if (type == typeof(IConvertible))
    {
        // This is faster for native types than using type converter...
        return (T)Convert.ChangeType(value, type, culture);
    }
    else
    {
        // Handle objects that have type converters (e.g., Enum, Color, Point, etc.)
        TypeConverter converter = TypeDescriptor.GetConverter(type);
        return (T)converter.ConvertFromString(null, culture, value);
    }
}
```

"case" statements should be indented from the switch statement like this:

```
switch (protocol.ToLower())
{
    case "tcp":
        server = new TcpServer(settings.ToString());
        break;
    case "udp":
        server = new UdpServer(settings.ToString());
        break;
    default:
        throw new ArgumentException("Protocol \'" + protocol + "\' is not valid");
}
```

Braces can only be considered optional when there is only one line of following code. Even so, code following the statement should always be on the next line - not on the same line. If one side of an *if/else* statement needs braces, use braces on both sides. For example:

```
// This is OK:
if (ServerStarted != null)
    ServerStarted(this, EventArgs.Empty);

// This is *NOT* OK:
if (ServerStarted != null) ServerStarted(this, EventArgs.Empty);
```

```
// This is OK:
if (value && !Enabled)
    Start();
else if (!value && Enabled)
    Stop();

// This is *NOT* OK:
if (value && !Enabled) Start(); else if (!value && Enabled) Stop();
```

```
// This is OK:
if (processInterval == RealTimeProcessInterval)
{
    m_processingIsRealTime = true;
}
else
{
    m_processTimer = new System.Timers.Timer();
    m_processTimer.Elapsed += ProcessTimerThreadProc;
}
```

```
// This is *NOT* OK:
if (processInterval == RealTimeProcessInterval)
    m_processingIsRealTime = true;
else
{
    m_processTimer = new System.Timers.Timer();
    m_processTimer.Elapsed += ProcessTimerThreadProc;
}
```

Single Line Statements

Single line statements are code blocks that have braces that begin and end on the same line. Although they may look more compact, for the sake of readability and consistency with existing code, single line statements should be avoided:

```
// This is OK:
public virtual bool RequeueOnTimeout
{
    get
    {
        return m_requeueOnTimeout;
    }
    set
    {
        m_requeueOnTimeout = value;
    }
}

// This is *NOT* OK:
public virtual bool RequeueOnTimeout
{
    get { return m_requeueOnTimeout;  }
    set { m_requeueOnTimeout = value; }
}
```

Commenting

Comments should be used to describe intention, algorithmic overview, and/or logical flow with the goal that if from reading the comments alone, someone other than the author could understand a function's intended behavior and general operation. There is no such thing as too much commenting. All code needs some level of commenting to reflect the programmer's intent and approach.

File Headers

Each file should start with the standard header and copyright notice. For the Grid Protection Alliance, the header is as follows:

```
//******************************************************************************************************
//  Foo.cs - Gbtc
//
//  Copyright © 2010, Grid Protection Alliance.  All Rights Reserved.
//
//  Licensed to the Grid Protection Alliance (GPA) under one or more contributor license agreements. See
//  the NOTICE file distributed with this work for additional information regarding copyright ownership.
//  The GPA licenses this file to you under the Eclipse Public License -v 1.0 (the "License"); you may
//  not use this file except in compliance with the License. You may obtain a copy of the License at:
//
//      http://www.opensource.org/licenses/eclipse-1.0.php
//
//  Unless agreed to in writing, the subject software distributed under the License is distributed on an
//  "AS-IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. Refer to the
//  License for the specific language governing permissions and limitations.
//
//  Code Modification History:
//  ----------------------------------------------------------------------------------------------------
//  03/29/2011 - J. Ritchie Carroll
//       Generated original version of source code.
//
//******************************************************************************************************
```

For the Tennessee Valley Authority, the header looks like this:

```
//******************************************************************************************************
//  Foo.cs - Gbtc
//
//  Tennessee Valley Authority, 2009
//  No copyright is claimed pursuant to 17 USC § 105.  All Other Rights Reserved.
//
//  This software is made freely available under the TVA Open Source Agreement (see below).
//  Code in this file licensed to TVA under one or more contributor license agreements listed below.
//
//  Code Modification History:
//  ----------------------------------------------------------------------------------------------------
//  03/29/2011 - Pinal C. Patel
//       Original version of source code generated.
//
//******************************************************************************************************
```

The TVA open source agreement and any contributor license agreements often follow for code being maintained and distributed by TVA.

Either of the headers can be automatically inserted in the document using a standard macro.

Documentation Comments

All public and protected elements (methods, properties, events, delegates, enumerations, constants, etc.) should use fully populated XML documentation comments. Private elements must have a comment also but can use a non-structured simple // comment. As a matter of style, periods are not normally used within short intra-method code comments. Full standard punctuation is used for any structured comments since these comments are used to produce automated documentation:

```
/// <summary>
/// Represents a proper commenting example.
/// </summary>
public class Foo
{
    /// <summary>
    /// Releases all the resources used by the <see cref="Foo"/> object.
    /// </summary>
    public void Dispose()
    {
        // Call the protected dispose method to release resources
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

In cases where you need to move the XML documentation to an external file, use of the `<include>` tag is allowed. However enough local non-structured simple `//` comments still need to exist to guide the reader about the author's intent without having to go track down the external documentation.

```
    /// <include file='doc\Foo.uex' path='docs/doc[@for="Foo.MyMethod"]/*' />
    // Releases the unmanaged resources used by the Foo object and
    // optionally releases the managed resources.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
```

Comment Style

The `//` (two slashes) style of comment tags should be used in most situations (that is, instead of the `/* start/stop comment */` style) . Where ever possible, place comments above the code instead of beside it. Here are some examples:

```
    // Load settings from config file
    LoadSettings();

    // See if this computer is part of a domain
    if (UserInfo.MachineIsJoinedToDomain)
    {
```

Comments can also be placed at the end of a line when space allows:

```
// Fields
private FrameQueue m_frameQueue;   // Queue of frames to be published
private int m_framesPerSecond;     // Frames per second
private double m_lagTime;          // Allowed past time deviation tolerance, in seconds
private double m_leadTime;         // Allowed future time deviation tolerance, in seconds
```

Spacing

Spaces improve readability by decreasing code density. Here are some guidelines for the use of space characters within code:

- Use a single space after a comma between function arguments:

Right:	`Console.In.Read(myChar, 0, 1);`
Wrong:	`Console.In.Read(myChar,0,1);`

- Use a single space before flow control statements:

Right:	`while (x == y)`
Wrong:	`while(x==y)`

- Use a single space before and after comparison operators:
  Right: `if (x == y)`
  Wrong: `if (x==y)`
- ***Do not*** use a space after the parenthesis and function arguments:
  Right: `CreateFoo(myChar, 0, 1)`
  Wrong: `CreateFoo( myChar, 0, 1 )`
- ***Do not*** use spaces between a function name and parenthesis:
  Right: `CreateFoo()`
  Wrong: `CreateFoo ()`
- ***Do not*** use spaces inside brackets:
  Right: `x = dataArray[index];`
  Wrong: `x = dataArray[ index ];`

Regions

All code should use the following code regions for any non-static class:

```
public class Foo
{
    #region [ Members ]

    // Nested Types

    // Constants

    // Delegates

    // Events

    // Fields

    #endregion

    #region [ Constructors ]

    #endregion

    #region [ Properties ]

    #endregion

    #region [ Methods ]

    #endregion

    #region [ Operators ]

    #endregion

    #region [ Static ]

    // Static Fields
```

```
        // Static Constructor

        // Static Properties

        // Static Methods

        #endregion
    }
```

Unused regions should be removed. The region code can automatically be applied using the "regions snippet".

Additionally, an "`[ Enumerations ]`" region is used to encapsulate enumeration definitions and is located above the class that initially or primarily uses the enumeration:

```
    #region [ Enumerations ]

    /// <summary>
    /// Indicates the current state of the client.
    /// </summary>
    public enum ClientState
    {
        /// <summary>
        /// Client is establishing connection.
        /// </summary>
        Connecting,
        /// <summary>
        /// Client has established connection.
        /// </summary>
        Connected,
        /// <summary>
        /// Client connection is terminated.
        /// </summary>
        Disconnected
    }

    #endregion
```

Implementation of IDisposable

Classes using any member variable that implements IDisposable should themselves implement IDisposable. Failing to properly implement the recommended dispose pattern for classes can lead to memory leaks and abnormal termination exceptions. The proper implementation of IDisposable can be setup easily using the "idisposable snippet". If you are overriding a base class or component that already implements IDisposable, you can use the "disposec snippet" instead. After using the snippet, make sure code segments end up in their proper regions: the `m_disposed` private member variable should be in "`[ Members ]`" region under "`// Fields`", the destructor (*~class()*) should be located in the "`[ Constructors ]`" region, and finally both "`Dispose()`" methods should be the first methods under "`[ Methods ]`" region. Here is a fully implemented example:

```
/// <summary>
/// Represents a proper implementation example of <see cref="IDisposable"/>.
/// </summary>
public class Foo : IDisposable
{
    #region [ Members ]

    // Fields
    private FileStream m_cache;     // Stream object for file cache
    private bool m_disposed;        // Disposed flag

    #endregion

    #region [ Constructors ]

    /// <summary>
    /// Creates a new instance of <see cref="Foo"/>.
    /// </summary>
    public Foo()
    {
        m_cache = new FileStream("local.cache", FileMode.OpenOrCreate);
    }

    /// <summary>
    /// Releases <see cref="Foo"/> object resources.
    /// </summary>
    ~Foo()
    {
        Dispose(false);
    }

    #endregion

    #region [ Methods ]

    /// <summary>
    /// Releases all the resources used by the <see cref="Foo"/> object.
    /// </summary>
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    /// <summary>
    /// Releases the unmanaged resources used by the <see cref="Foo"/> object.
    /// </summary>
    /// <param name="disposing">true to release both managed and unmanaged resources;
false to release only unmanaged resources.</param>
    protected virtual void Dispose(bool disposing)
    {
        if (!m_disposed)
        {
            try
            {
                if (disposing)
                {
                    if (m_cache != null)
                        m_cache.Dispose();
                    m_cache = null;
                }
            }
            finally
            {
                m_disposed = true;  // Prevent duplicate dispose.
            }
        }
    }

    #endregion
}
```

Naming

Make names long enough to be meaningful but short enough to avoid excessive verbosity, but do not abbreviate. Programmatically, a unique name serves only to differentiate one item from another but expressive names function as an aid to a human reader; therefore, it makes sense to provide a detailed name that a human reader can comprehend. A summary of the naming conventions in this standard include:

- Use a "`m_`" prefix for **private** member variables
- Use a "`s_`" prefix for **private** static variables
- Use *camelCase* for **private** member variables
- Use *PascalCase* for **internal**, **protected** or **public** member variables
- Use *camelCase* for parameters
- Use *camelCase* for local variables
- Use *PascalCase* for function, property, event, and class names
- Prefix interfaces names with `I`
- ***Do not*** use *Hungarian* notation
- ***Do not*** abbreviate
- ***Do not*** prefix *enums*, *classes*, or *delegates* with any letter

The goal is to produce *consistent* source code appearance (especially with existing code) and clean readable source. Code legibility should be the primary objective.

General

- Use *PascalCase* for class, method, property and constant names.
- 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`.
- In cases where a name is composed of multiple sections separated by periods (i.e., assembly names), sections that are only one or two letters in length should have all letters capitalized (e.g., `IO` or `UI`).

- Except for industry acceptable abbreviations and acronyms (e.g., `TVA`, `Xml`) or common technical abbreviations and acronyms (e.g., `Usb`, `Dvd`), avoid using abbreviations. Is OK to use common, well-known acronyms to replace lengthy phrase names (such as, `UI`).
- Other abbreviations should be avoided - such as, avoid using `Min` or `Max` - instead spell these words out (i.e., `Minimum` or `Maximum`) for clarity and consistency with existing code. Use the following motto to help you remember: "***if in doubt, spell it out***".
- When using an accepted abbreviation, use it consistently. An abbreviation should have only one meaning and likewise, each abbreviated word should have only one abbreviation. For example, if you use `UI` to abbreviate `UserInterface`, do so everywhere and do not use `UI` to also abbreviate `UniversalInterface`.
- Avoid using abbreviations in identifiers or parameter names.
- When naming functions, include a description of the value being returned, such as `GetCurrentWindowName`().
- Use the verb-noun method for naming routines that perform some operation on a given object, such as `CalculateInvoiceTotal`().
- File and folder names, like procedure names, should accurately describe their purpose.
- All overloads should perform a similar function. For those languages that do not permit function overloading, establish a naming standard that relates similar functions.
- Avoid elusive names that are open to subjective interpretation, such as `AnalyzeThis`() for a routine, or `xxK8` for a variable. Such names contribute to ambiguity more than abstraction. It is better to be as descriptive as possible
- Avoid reusing the same name for different elements, such as a routine called `ProcessSales`() and a variable called `processSales`.
- When naming elements, avoid commonly misspelled words. Also, be aware of differences that exist between regional spellings, such as *color/colour* and *check/cheque*. Use U.S. English as the default for spelling.
- Avoid reserved keywords for class or method names.
- Avoid typographical marks to identify data types, such as **$** for strings or **%** for integers.

PascalCase vs. camelCase

Since most names are constructed by concatenating several words, use mixed-case formatting to simplify reading them. In addition, to help distinguish between variables and routines, use Pascal casing (e.g., `CalculateInvoiceTotal`) for routine names where the first letter of each word is capitalized. For variable and parameter names, use camel casing (e.g., `documentFormatType`) where the first letter of each word except the first is capitalized.

Everything that is publicly exposed (i.e., **public**, **protected**, or **internal**) should be in Pascal case with no underscore characters (_).

The only underscore characters that should ever be used are only on the **private** "`m_`" or "`s_`" prefix for member variables or static variables respectively.

Namespace Names

- Use Pascal case for namespaces, and separate logical components with periods.
- Do not create two namespaces with names that differ only by case.
- Do not use the same name for a namespace and a class.
- Do not use the underscore character (_) unless it would used as a substitute for a period in the namespace (e.g., the namespace for IEEE C37.118 would be `IeeeC37_118`).

Class Names

- Use a noun or noun phrase to name a class.
- Avoid using Class names that duplicate commonly used .NET Framework namespaces, such as: `System`, `Collections`, `Forms`, or `UI`.
- Do not use a type prefix, such as C, on a class name.
- Do not use the underscore character (_).
- Names for exception classes should end with the suffix "`Exception`".
- Names for custom attribute classes should end with the suffix "`Attribute`".
- Name an event argument class with the "`EventArgs`" suffix.
- Abstract classes should end with the suffix "`Base`".

Variable Names

- Do not use Hungarian notation prefixes on variable names (e.g., `intCount`)
- Avoid using acronyms when naming variables, except for variables with two or fewer letters or industry acceptable acronyms (e.g., `Xml`).
- Start the name of **private** member variables with "`m_`".
- Start the name of **private** static variables with "`s_`".
- Append computation qualifiers (`Average`, `Summary`, `Minimum`, `Maximum`, `Index`) to the end of a variable name where appropriate.
- Use complementary pairs in variable names, such as `Minimum`/`Maximum`, `Begin`/`End` and `Open`/`Close`.
- Boolean variable names should contain `Is` which implies *Yes*/*No* or *True*/*False* values, such as `fileIsFound`.
- Use nouns, noun phrases or abbreviations of nouns to name static fields.
- 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 `documentFlag`, use a more descriptive name such as `documentFormatType`.
- Variables should not contain the type name as part of the variable name (e.g., use `name` instead of `nameString`.)

- Even for a short-lived variable that may appear in only a few lines of code, still use a meaningful name. The ***only*** exception to this should be the use of single-letter variable names (e.g., `i`, `j`, `k`, or `x`, `y`, `z`) for short-loop indexes.
- When applicable, do not use literal numbers or strings when named constants or enumerations are available (e.g., `NumberOfDaysInWeek` instead of 7) for ease of maintenance and understanding.

Note: General practice is to avoid using protected or public member variables. Using a private member variable with an associated property is the preferred implementation (see example below). Exceptions can be made for extremely simple classes, private classes or in cases of optimization.

```
/// <summary>
/// Represents a proper implementation of a member variable.
/// </summary>
public class Foo
{
    #region [ Members ]

    // Fields
    private bool m_enabled;     // Enabled flag

    #endregion

    #region [ Properties ]

    /// <summary>
    /// Gets or sets enabled state for <see cref="Foo"/>.
    /// </summary>
    public bool Enabled
    {
        get
        {
            return m_enabled;
        }
        set
        {
            m_enabled = value;
        }
    }

    #endregion
}
```

Parameter Names

- Use camel case for parameter names.
- Use descriptive parameter names. Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios.
- Use names that describe a parameter's meaning rather than names that describe a parameter's type.

Event Names

- 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 (e.g., `Closing` and `Closed`).
- Do not use a prefix or suffix "`On`" for an event declaration on the type. For example, do use `Close` instead of `OnClose`. The "`On`" prefix is used for protected methods that raise the event for the benefit of inherited classes, for example:

```
/// <summary>
/// Raises the <see cref="ConnectionAttempt"/> event.
/// </summary>
protected virtual void OnConnectionAttempt()
{
    m_currentState = ClientState.Connecting;

    if (ConnectionAttempt != null)
        ConnectionAttempt(this, EventArgs.Empty);
}
```

Note: Event implementations should not use reserved parameters. Event implementations should use the standard `EventHandler` syntax, for example:

```
/// <summary>
/// Occurs when the client has successfully sent data to the server.
/// </summary>
[Category("Data"),
Description("Occurs when the client has successfully sent data to the server.")]
public event EventHandler SendDataComplete;

/// <summary>
/// Occurs when an <see cref="Exception"/> is encountered sending data to the server.
/// </summary>
/// <remarks>
/// <see cref="EventArgs{T}.Argument"/> is the <see cref="Exception"/> encountered.
/// </remarks>
[Category("Data"),
Description("Occurs when an Exception is encountered when sending data to the server.")]
public event EventHandler<EventArgs<Exception>> SendDataException;
```

Enumeration Names

- Name enumerations with nouns or noun phrases, or adjectives that describe behavior.
- For enumerations representing distinct items, do not make the enumeration name plural.
- For enumerations representing related items (such as bit flags that can be OR'd together), make the enumeration name plural.
- Do not prefix parameter names with Hungarian type notation.
- Do not use an enum suffix on enum type names.
- Do not use the underscore character (_).

Note: the FlagsAttribute should always be added to enumerations representing bit flags.

Method Names

- Use verbs or verb phrases to name methods.
- Do not prefix parameter names with Hungarian type notation.
- Do not use the underscore character (_).

Property Names

- Use nouns or noun phrases to name properties.
- Consider naming a property with the same name as its underlying type.
- Do not use Hungarian notation prefixes on property names.

Interface Names

- Name interfaces with nouns or noun phrases, or adjectives that describe behavior.
- Interface names should begin with the prefix `I`.
- Use similar names when defining a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter `I` prefix on the interface name and/or `Base` suffix for abstract classes.
- Do not use the underscore character (_).

Object Names

- Starting with the default name for controls, replace the control sequence number with a more descriptive name (e.g. replace `Button1` with `ButtonSaveFile`).

Solution Names

Create the SourceSafe Solution folder for the entire application with all relevant project folders underneath. For example, one would expect to find the public class "`System.Windows.Forms.Control`" in the file system as **System\Windows\Forms\Control.cs**.

When creating or changing an application's folder, use complete names whenever possible, including spaces (no underscores), for the folder names.

Avoid using acronyms and abbreviations when naming folders, unless the application's name is too long. For example, the application "Writing New Code" would be stored in the folder "Writing New Code", not in "WNC".

Interop Methods

Methods used as *interop wrappers* (e.g., `DllImport` statements) are currently contained to the `TVA.Interop.WindowsApi` static class. An exception has been made for naming conventions for elements (e.g., *structures*, *enumerations* and *constants*) within this class to allow named consistency with their Win32 counterparts. It is suggested that if new API calls are needed that they be added to this existing class for containment and isolation.

File Organization

- Source files should contain only one public class type.
- Care should be taken so that the public class name and the file name match.
- Enumerations and delegates should normally appear above the class with primary association.
- All source code files should include a standard header.
- Using statements should come after the header and before the namespace declaration.
- Using statements should be sorted and relevant (i.e. right click anywhere in the source code and select Organize Usings / Remove and Sort)
- Standard regions should be used in all non-static classes.
- Non-standard regions should be avoided unless code is large and the regions are helping with extra organization.
- Static classes typically define no regions.
- Directory names should follow the namespace hierarchy for the class.