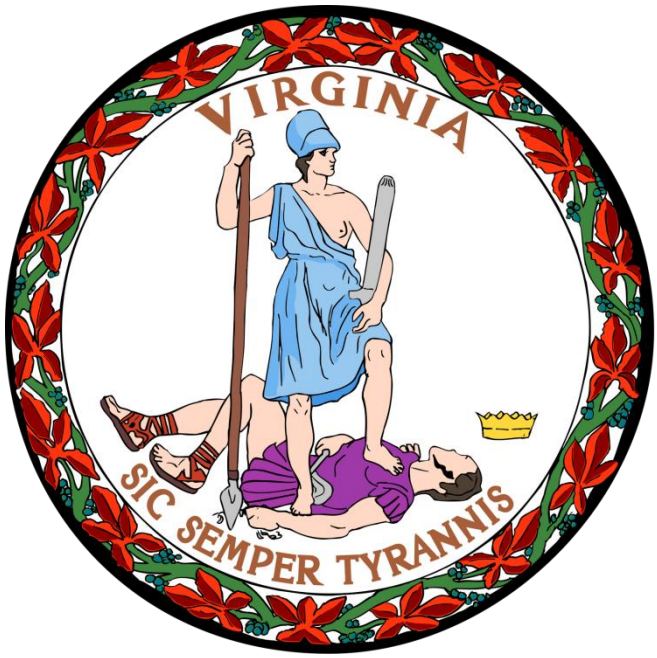
DBHDS Business Solution Development
Jefferson Building, 5rd Floor
mario.epps@dbhds.virginia.gov

.NET Framework Guidelines and Best Practices



Application Architect: Mario Epps

DBHDS Business Solution Development
Jefferson Building, 5rd Floor
mario.epps@dbhds.virginia.gov

Department of Behavioral Health and Developmental Services - Abstract

---

Abstract

This document describes the coding style guidelines for native .NET (C# and VB.NET) programming used by the Business Solutions Development team.

Application Architect: Mario Epps

Contents

Overview

This document defines the native .NET coding standard for the Business Solutions Development team project team. This standard derives from the experience of product development efforts and is continuously evolving. If you discover a new best practice or a topic that is not covered, please bring that to the attention of the Business Solutions Development team and have the conclusion added to this document.

No set of guidelines will satisfy everyone. The goal of a standard is to create efficiencies across a community of developers. Applying a set of well-defined coding standards will result in code with fewer bugs, and better maintainability.

1.1 Principles & Themes

High-quality samples exhibit the following characteristics because customers use them as examples of best practices:

1. **Understandable.** Samples must be clearly readable and straightforward. They must showcase the key things they're designed to demonstrate. The relevant parts of a sample should be easy to reuse. Samples should not contain unnecessary code. They must include appropriate documentation.
2. **Correct.** Samples must demonstrate properly how to perform the key things they are designed to teach. They must compile cleanly, run correctly as documented, and be tested.
3. **Consistent.** Samples should follow consistent coding style and layout to make the code easier to read. Likewise, samples should be consistent with each other to make them easier to use together. Consistency shows craftsmanship and attention to detail.
4. **Modern.** Samples should demonstrate current practices such as use of Unicode, error handling, defensive programming, and portability. They should use current recommendations for runtime library and API functions. They should use recommended project & build settings.
5. **Safe.** Samples must comply with legal, privacy, and policy standards. They must not demonstrate hacks or poor programming practices. They must not permanently alter machine state. All installation and execution steps must be reversible.
6. **Secure.** The samples should demonstrate how to use secure programming practices such as least privilege, secure versions of runtime library functions, and SDL-recommended project settings.

1.2 Terminology

Through-out this document there will be recommendations or suggestions for standards and practices. Some practices are very important and must be followed, others are guidelines that are beneficial in certain scenarios but are not applicable

everywhere. In order to clearly state the intent of the standards and practices that are discussed we will use the following terminology.

Wording	Intent	Justification
☑ **Do...**	This standard or practice should be followed in all cases. If you think that your specific application is exempt, it probably isn't.	These standards are present to mitigate bugs.
☒ **Do Not...**	This standard or practice should never be applied.	
☑ **You should...**	This standard or practice should be followed in most cases.	These standards are typically stylistic and attempt to promote a consistent and clear style.
☒ **You should not...**	This standard or practice should not be followed, unless there's reasonable justification.	
☑ **You can…**	This standard or practice can be followed if you want to; it's not necessarily good or bad. There are probably implications to following the practice (dependencies, or constraints) that should be considered before adopting it.	These standards are typically stylistic, but are not ubiquitously adopted.

General Coding Standards

These general coding standards can be applied to all languages - they provide high-level guidance to the style, formatting and structure of your source code.

2.1 Clarity and Consistency

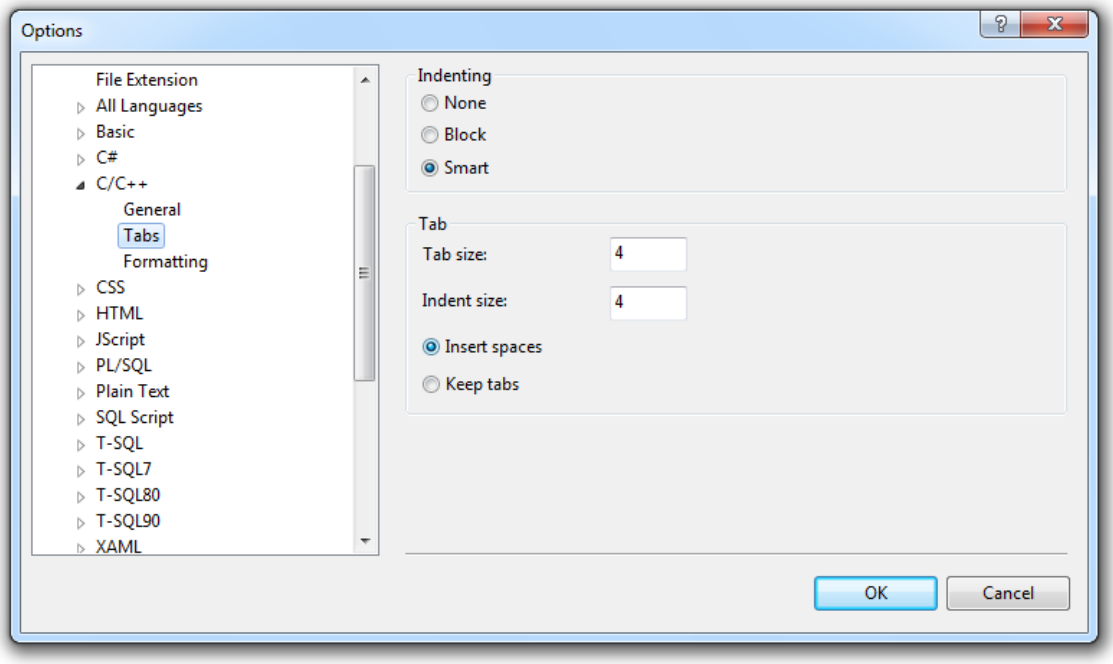
☑ **Do** ensure that clarity, readability and transparency are paramount. These coding standards strive to ensure that the resultant code is easy to understand and maintain, but nothing beats fundamentally clear, concise, self-documenting code.

☑ **Do** ensure that when applying these coding standards that they are applied consistently.

2.2 Formatting and Style

☒ **Do not** use tabs. It's generally accepted across Microsoft that tabs shouldn't be used in source files - different text editors use different spacing to render tabs, and this causes formatting confusion. All code should be written using four spaces for indentation.

The Visual Studio text editor can be configured to insert spaces for tabs.



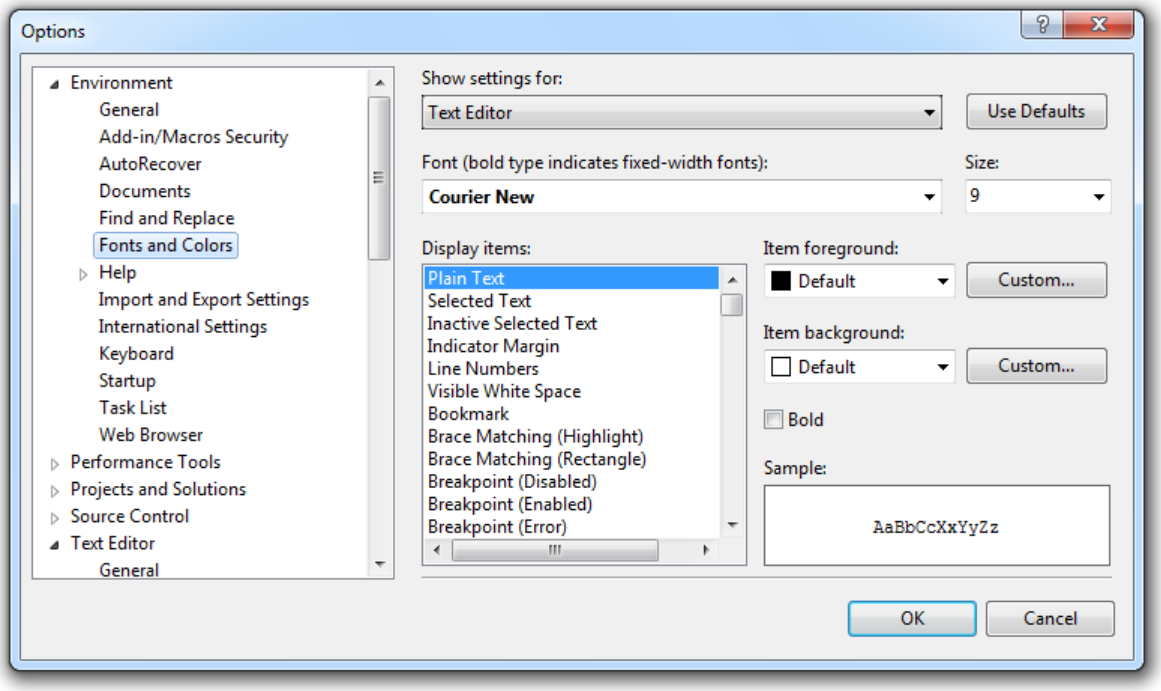
☑ **You should** limit the length of lines of code. Having overly long lines inhibits the readability of code. Break the code line when the line length is greater than column 78 for readability. If column 78 looks too narrow, use column 86 or 90.

Visual C# sample:

```
// Get and display whether the primary access token of the process belongs
// to user account that is a member of the local Administrators group even
// if it currently is not elevated (IsUserInAdminGroup).
try
{
    bool fInAdminGroup = IsUserInAdminGroup();
    this.lbInAdminGroup.Text = fInAdminGroup.ToString();
}
catch (Exception ex)
{
    this.lbInAdminGroup.Text = "N/A";
    MessageBox.Show(ex.Message, "An error occurred in IsUserInAdminGroup",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

Column 86

☑ **Do** use a fixed-width font, typically Courier New, in your code editor.



2.3 Using Libraries

☒ **Do not** reference unnecessary libraries, include unnecessary header files, or reference unnecessary assemblies. Paying attention to small things like this can improve build times, minimize chances for mistakes, and give readers a good impression.

2.4 Global Variables

☑ **Do** minimize global variables. To use global variables properly, always pass them to functions through parameter values. Never reference them inside of functions or classes directly because doing so creates a side effect that alters the state of the global without the caller knowing. The same goes for static variables. If you need to modify a global variable, you should do so either as an output parameter or return a copy of the global.

2.5 Variable Declarations and Initializations

☑ **Do** declare local variables in the minimum scope block that can contain them, typically just before use if the language allows; otherwise, at the top of that scope block.

☑ **Do** initialize variables when they are declared.

☑ **Do** declare and initialize/assign local variables on a single line where the language allows it. This reduces vertical space and makes sure that a variable does not exist in an un-initialized state or in a state that will immediately change.

```
// C++ sample:
        HANDLE hToken = NULL;
        PSID pIntegritySid = NULL;
        STARTUPINFO si = { sizeof(si) };
        PROCESS_INFORMATION pi = { 0 };
// C# sample:
        string name = myObject.Name;
        int val = time.Hours;
' VB.NET sample:
        Dim name As String = myObject.Name
```

```
Dim val As Integer = time.Hours
```

☒ **Do not** declare multiple variables in a single line. One declaration per line is recommended since it encourages commenting, and could avoid confusion. As a Visual C++ example,

Good:

```
CodeExample *pFirst = NULL;  // Pointer of the first element.

CodeExample *pSecond = NULL; // Pointer of the second element.
```

Bad:

```
CodeExample *pFirst, *pSecond;
```

The latter example is often mistakenly written as:

```
CodeExample *pFirst, pSecond;
```

Which is *actually* equivalent to:

```
CodeExample *pFirst;

CodeExample pSecond;
```

2.6 Function Declarations and Calls

The function/method name, return value and parameter list can take several forms. Ideally this can all fit on a single line. If there are many arguments that don't fit on a line those can be wrapped, many per line or one per line. Put the return type on the same line as the function/method name. For example,

Single Line Format:

```
// C++ function declaration sample:
HRESULT DoSomeFunctionCall(int param1, int param2, int *param3);
// C++ / C# function call sample:
hr = DoSomeFunctionCall(param1, param2, param3);
' VB.NET function call sample:
hr = DoSomeFunctionCall(param1, param2, param3)
```

Multiple Line Formats:

```
// C++ function declaration sample:
        HRESULT DoSomeFunctionCall(int param1, int param2, int *param3,
```

```
            int param4, int param5);

// C++ / C# function call sample:
        hr = DoSomeFunctionCall(param1, param2, param3,

            param4, param5);

' VB.NET function call sample:
        hr = DoSomeFunctionCall(param1, param2, param3, _param4, param5)
```

When breaking up the parameter list into multiple lines, each type/parameter pair should line up under the preceding one, the first one being on a new line, indented one tab. Parameter lists for function/method *calls* should be formatted in the same manner.

```
// C++ function declaration sample:
        HRESULT DoSomeFunctionCall(

          HWND hwnd,    // You can comment parameters, too

          T1 param1,    // Indicates something

          T2 param2,    // Indicates something else

          T3 param3,    // Indicates more

          T4 param4,    // Indicates even more

          T5 param5);   // You get the idea

// C++ / C# function call sample:
        hr = DoSomeFunctionCall(

          hwnd,

          param1,

          param2,

          param3,

          param4,

          param5);
```

```
' VB.NET function call sample:
        hr = DoSomeFunctionCall( _
          hwnd, _
          param1, _
          param2, _
          param3, _
          param4, _
          param5)
```

☑ **Do** order parameters, grouping the in parameters first, the out parameters last. Within the group, order the parameters based on what will help programmers supply the right values. For example, if a function takes arguments named “left” and “right”, put “left” before “right” so that their place match their names. When designing a series of functions which take the same arguments, use a consistent order across the functions. For example, if one function takes an input handle as the first parameter, all of the related functions should also take the same input handle as the first parameter.

2.7 Statements

☒ **Do not** put more than one statement on a single line because it makes stepping through the code in a debugger much more difficult.

Good:

```
// C++ / C# sample:
        a = 1;
        b = 2;
' VB.NET sample:
        If (IsAdministrator()) Then
          Console.WriteLine("YES")
        End If
```

Bad:

```
// C++ / C# sample:
        a = 1; b = 2;

' VB.NET sample:
        If (IsAdministrator()) Then Console.WriteLine("YES")
```

2.8 Enums

☑ **Do** use an enum to strongly type parameters, properties, and return values that represent sets of values.

☑ **Do** favor using an enum over static constants or “#define” values . An enum is a structure with a set of static constants. The reason to follow this guideline is because you will get some additional compiler and reflection support if you define an enum versus manually defining a structure with static constants.

```
        Good:
        // C++ sample:
        enum Color
        {
           Red,
           Green,
           Blue
        };
        // C# sample:
        public enum Color
        {
           Red,
           Green,
           Blue
        }
        ' VB.NET sample:
```

```
Public Enum Color
   Red
   Green
   Blue
End Enum
```

Bad:

```
// C++ sample:
const int RED   = 0;
const int GREEN = 1;
const int BLUE  = 2;
#define RED      0
#define GREEN    1
#define BLUE     2
// C# sample:
public static class Color
{
   public const int Red = 0;
   public const int Green = 1;
   public const int Blue = 2;
}
' VB.NET sample:
Public Class Color
   Public Const Red As Integer = 0
   Public Const Green As Integer = 1
   Public Const Blue As Integer = 2
```

```
End Class
```

☒ **Do not** use an enum for open sets (such as the operating system version, names of your friends, etc.).

☑ **Do** provide a value of zero on simple enums. Consider calling the value something like “None.” If such value is not appropriate for this particular enum, the most common default value for the enum should be assigned the underlying value of zero.

```
// C++ sample:
enum Compression
{
   None = 0,
   GZip,
   Deflate
};
// C# sample:
public enum Compression
{
   None = 0,
   GZip,
   Deflate
}
' VB.NET sample:
Public Enum Compression
   None = 0
   GZip
   Deflate
End Enum
```

☒ **Do not** use Enum.IsDefined for enum range checks in .NET. There are really two problems with Enum.IsDefined. First it loads reflection and a bunch of cold type metadata, making it a surprisingly expensive call. Second, there is a versioning issue here.

```
Good:

// C# sample:

if (c > Color.Black || c < Color.White)

{

   throw new ArgumentOutOfRangeException(...);

}

' VB.NET sample:

If (c > Color.Black Or c < Color.White) Then

   Throw New ArgumentOutOfRangeException(...);

End If

Bad:

// C# sample:

if (!Enum.IsDefined(typeof(Color), c))

{

   throw new InvalidEnumArgumentException(...);

}

' VB.NET sample:

If Not [Enum].IsDefined(GetType(Color), c) Then

   Throw New ArgumentOutOfRangeException(...);
```

2.8.1 Flag Enums

Flag enums are designed to support bitwise operations on the enum values. A common example of the flags enum is a list of options.

☑ **Do** apply the System.FlagsAttribute to flag enums in .NET. **Do not** apply this attribute to simple enums.

☑ **Do** use powers of two for the flags enum values so they can be freely combined using the bitwise OR operation. For example,

```
// C++ sample:
enum AttributeTargets
{
   Assembly = 0x0001,
   Class    = 0x0002,
   Struct   = 0x0004
   ...
};
// C# sample:
[Flags]
public enum AttributeTargets
{
   Assembly = 0x0001,
   Class    = 0x0002,
   Struct   = 0x0004,
   ...
}
' VB.NET sample:
<Flags()> _
Public Enum AttributeTargets
   Assembly = &H1
   Class = &H2
```

```
    Struct = &H4
    ...
End Enum
```

☑ **You should** provide special enum values for commonly used combinations of flags. Bitwise operations are an advanced concept and should not be required for simple tasks. FileAccess.ReadWrite is an example of such a special value. However, **you should not** create flag enums where certain combinations of values are invalid.

```
// C++ sample:
enum FileAccess
{
    Read = 0x1,
    Write = 0x2,
    ReadWrite = Read | Write
};
// C# sample:
[Flags]
public enum FileAccess
{
    Read = 0x1,
    Write = 0x2,
    ReadWrite = Read | Write
}
' VB.NET sample:
<Flags()> _
Public Enum FileAccess
    Read = &H1
```

```
    Write = &H2

    ReadWrite = Read Or Write

End Enum
```

☒ **You should not** use flag enum values of zero, unless the value represents “all flags are cleared” and is named appropriately as “None”. The following C# example shows a common implementation of a check that programmers use to determine if a flag is set (see the if-statement below). The check works as expected for all flag enum values except the value of zero, where the Boolean expression always evaluates to true.

.NET Coding Standards

3.1 Design Guidelines for Developing Class Libraries

These coding standards can be applied to C# and VB.NET. The Design Guidelines for Developing Class Libraries document on MSDN is a fairly thorough discussion of how to write managed code.

3.2 Files and Structure

☒ **Do not** have more than one public type in a source file, unless they differ only in the number of generic parameters or one is nested in the other. Multiple internal types in one file are allowed.

☑ **Do** name the source file with the name of the public type it contains. For example, MainForm class should be in MainForm.cs file and List<T> class should be in List.cs file.

3.3 Assembly Properties

The assembly should contain the appropriate property values describing its name, copyright, and so on.

Standard	Example
Set Copyright to Copyright © Microsoft Corporation 2010	[assembly: AssemblyCopyright("Copyright © Microsoft Corporation 2010")]
Set AssemblyCompany to Microsoft Corporation	[assembly: AssemblyCompany("Microsoft Corporation")]
Set both AssemblyTitle and AssemblyProduct to the current sample name	[assembly: AssemblyTitle("CSNamedPipeClient")] [assembly: AssemblyProduct("CSNamedPipeClient")]

3.4 Naming Conventions

3.4.1 General Naming Conventions

☑ **Do** use meaning names for various types, functions, variables, constructs and types.

☒ **You should not** use of shortenings or contractions as parts of identifier names. For example, use “GetWindow” rather than “GetWin”. For functions of common types, thread process, window procedures, dialog procedures use the common suffixes for these “ThreadProc”, “DialogProc”, “WndProc”.

☒ **Do not** use underscores, hyphens, or any other non-alphanumeric characters.

3.4.2 Capitalization Naming Rules for Identifiers

The following table describes the capitalization and naming rules for different types of identifiers.

Identifier	Casing	Naming Structure	Example
Class, Structure	PascalCasing	Noun	public class ComplexNumber {...} public struct ComplextStruct {...}
Namespace	PascalCasing	Noun ☒ **Do not** use the same name for a namespace and a type in that namespace.	namespace Microsoft.Sample.Windows7
Enumeration	PascalCasing	Noun ☑ **Do** name flag enums with plural nouns or noun phrases and simple enums with singular nouns or noun phrases.	[Flags] public enum ConsoleModifiers { Alt, Control }
Method	PascalCasing	Verb or Verb phrase	public void Print() {...} public void ProcessItem() {...}
Public Property	PascalCasing	Noun or Adjective ☑ **Do** name collection proprieties with a plural phrase describing the items in the collection, as opposed to a singular phrase followed by “List” or “Collection”.	public string CustomerName public ItemCollection Items public bool CanRead

		☑ **Do** name Boolean proprieties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also prefix Boolean properties with “Is,” “Can,” or “Has” but only where it adds value.	
Non-public Field	camelCasing or _camelCasing	Noun or Adjective. ☑ **Do** be consistent in a code sample when you use the '_' prefix.	private string name; private string _name;

Event	PascalCasing	Verb or Verb phrase
		☑ **Do** give events names with a concept of before and after, using the present and past tense.
		☒ **Do not** use “Before” or “After” prefixes or postfixes to indicate pre and post events.
			// A close event that is raised after the window is closed.
			public event WindowClosed
			
			// A close event that is raised before a window is closed.
			public event WindowClosing

Delegate	PascalCasing	☑ **Do** add the suffix ‘EventHandler’ to names of delegates that are used in events. ☑ **Do** add the suffix ‘Callback’ to names of delegates other than those used as event handlers. ☒ **Do not** add the suffix “Delegate” to a delegate.	public delegate WindowClosedEventHandler
Interface	PascalCasing ‘I’ prefix	Noun	public interface IDictionary

Constant	PascalCasing for publicly visible; camelCasing for internally visible; All capital only for abbreviation of one or two chars long.	Noun	public const string MessageText = "A"; private const string messageText = "B"; public const double PI = 3.14159...;
Parameter, Variable	camelCasing	Noun	int customerID;
Generic Type Parameter	PascalCasing 'T' prefix	Noun ☑ **Do** name generic type parameters with descriptive names, unless a single-letter name is completely self-explanatory and a descriptive name would not add value. ☑ **Do** prefix descriptive type parameter names with T. ☑ **You should** using T as the type parameter name for types with one single-letter type parameter.	T, TItem, TPolicy
Resource	PascalCasing	Noun ☑ **Do** provide descriptive rather than short identifiers. Keep them concise where possible, but do not sacrifice	ArgumentExceptionInvalidName

		readability for space. ☑ **Do** use only alphanumeric characters and underscores in naming resources.	

3.4.3 Hungarian Notation

☒ **Do not** use Hungarian notation (i.e., do not encode the type of a variable in its name) in .NET.

3.4.4 UI Control Naming Conventions

UI controls would use the following prefixes. The primary purpose was to make code more readable.

Control Type	Prefix
Button	btn
CheckBox	chk
CheckedListBox	lst
ComboBox	cmb
ContextMenu	mnu
DataGrid	dg
DateTimePicker	dtp
Form	suffix: XXXForm
GroupBox	grp
ImageList	iml
Label	lb

ListBox	lst
ListView	lvw
Menu	mnu
MenuItem	mnu
NotificationIcon	nfy
Panel	pnl
PictureBox	pct
ProgressBar	prg
RadioButton	rad
Splitter	spl
StatusBar	sts
TabControl	tab
TabPage	tab
TextBox	tb
Timer	tmr
TreeView	tvw

For example, for the “File | Save” menu option, the “Save” MenuItem would be called “mnuFileSave”.

3.5 Constants

☑ **Do** use constant fields for constants that will never change. The compiler burns the values of const fields directly into calling code. Therefore const values can never be changed without the risk of breaking compatibility.

```
public class Int32
{
   public const int MaxValue = 0x7fffffff;
   public const int MinValue = unchecked((int)0x80000000);
}
```

```
Public Class Int32
   Public Const MaxValue As Integer = &H7FFFFFFF
   Public Const MinValue As Integer = &H80000000
End Class
```

☑ **Do** use public static (shared) readonly fields for predefined object instances. If there are predefined instances of the type, declare them as public readonly static fields of the type itself. For example,

```
public class ShellFolder
{
   public static readonly ShellFolder ProgramData = new ShellFolder("ProgramData");
   public static readonly ShellFolder ProgramFiles = new ShellFolder("ProgramData");
   ...
}
```

```
Public Class ShellFolder
   Public Shared ReadOnly ProgramData As New ShellFolder("ProgramData")
   Public Shared ReadOnly ProgramFiles As New ShellFolder("ProgramFiles")
   ...
End Class
```

3.6 Strings

☒ **Do not** use the '+' operator (or '&' in VB.NET) to concatenate many strings. Instead, you should use StringBuilder for concatenation. However, **do** use the '+' operator (or '&' in VB.NET) to concatenate small numbers of strings.

Good:

```
StringBuilder sbXML = new StringBuilder();
sbXML.Append("<parent>");
sbXML.Append("<child>");
sbXML.Append("Data");
sbXML.Append("</child>");
sbXML.Append("</parent>");
```

Bad:

```
String sXML = "<parent>";
sXML += "<child>";
sXML += "Data";
sXML += "</child>";
sXML += "</parent>";
```

☑ **Do** use overloads that explicitly specify the string comparison rules for string operations. Typically, this involves calling a method overload that has a parameter of type StringComparison.

☑ **Do** use StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase for comparisons as your safe default for culture-agnostic string matching, and for better performance.

☑ **Do** use string operations that are based on StringComparison.CurrentCulture when you display output to the user.

☑ **Do** use the non-linguistic StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase values instead of string operations based on CultureInfo.InvariantCulture when the comparison is linguistically irrelevant (symbolic, for example). Do not use string operations based on StringComparison.InvariantCulture in most cases. One of the few exceptions is when you are persisting linguistically meaningful but culturally agnostic data.

☑ **Do** use an overload of the String.Equals method to test whether two strings are equal. For example, to test if two strings are equal ignoring the case,

```
if (str1.Equals(str2, StringComparison.OrdinalIgnoreCase))
If (str1.Equals(str2, StringComparison.OrdinalIgnoreCase)) Then
```

☒ **Do not** use an overload of the String.Compare or CompareTo method and test for a return value of zero to determine whether two strings are equal. They are used to sort strings, not to check for equality.

☑ **Do** use the String.ToUpperInvariant method instead of the String.ToLowerInvariant method when you normalize strings for comparison.

3.7 Arrays and Collections

☑ **You should** use arrays in low-level functions to minimize memory consumption and maximize performance. In public interfaces, do prefer collections over arrays.

Collections provide more control over contents, can evolve over time, and are more usable. In addition, using arrays for read-only scenarios is discouraged as the cost of cloning the array is prohibitive.

However, if you are targeting more skilled developers and usability is less of a concern, it might be better to use arrays for read-write scenarios. Arrays have a smaller memory footprint, which helps reduce the working set, and access to elements in an array is faster as it is optimized by the runtime.

☒ **Do not** use read-only array fields. The field itself is read-only and can't be changed, but elements in the array can be changed. This example demonstrates the pitfalls of using read-only array fields:

Bad:

```
public static readonly char[] InvalidPathChars = { '\"', '<', '>', '|'};
```

This allows callers to change the values in the array as follows:

```
InvalidPathChars[0] = 'A';
```

Instead, you can use either a read-only collection (only if the items are immutable) or clone the array before returning it. However, the cost of cloning the array may be prohibitive.

```
public static ReadOnlyCollection<char> GetInvalidPathChars()
{
   return Array.AsReadOnly(badChars);
}
public static char[] GetInvalidPathChars()
{
```

```
    return (char[])badChars.Clone();
}
```

☑ **You should** use jagged arrays instead of multidimensional arrays. A jagged array is an array with elements that are also arrays. The arrays that make up the elements can be of different sizes, leading to less wasted space for some sets of data (e.g., sparse matrix), as compared to multidimensional arrays. Furthermore, the CLR optimizes index operations on jagged arrays, so they might exhibit better runtime performance in some scenarios.

```
// Jagged arrays
int[][] jaggedArray =
{
    new int[] {1, 2, 3, 4},
    new int[] {5, 6, 7},
    new int[] {8},
    new int[] {9}
};
Dim jaggedArray As Integer()() = New Integer()() _
{ _
    New Integer() {1, 2, 3, 4}, _
    New Integer() {5, 6, 7}, _
    New Integer() {8}, _
    New Integer() {9} _
}
// Multidimensional arrays
int [,] multiDimArray =
{
    {1, 2, 3, 4},
```

```
    {5, 6, 7, 0},
    {8, 0, 0, 0},
    {9, 0, 0, 0}
};

Dim multiDimArray(,) As Integer = _
{ _
    {1, 2, 3, 4}, _
    {5, 6, 7, 0}, _
    {8, 0, 0, 0}, _
    {9, 0, 0, 0} _
}
```

☑ **Do** use Collection<T> or a subclass of Collection<T> for properties or return values representing read/write collections, and use ReadOnlyCollection<T> or a subclass of ReadOnlyCollection<T> for properties or return values representing read-only collections.

☑ **You should** reconsider the use of ArrayList because any objects added into the ArrayList are added as System.Object and when retrieving values back from the arraylist, these objects are to be unboxed to return the actual value type. So it is recommended to use the custom typed collections instead of ArrayList. For example, .NET provides a strongly typed collection class for String in System.Collection.Specialized, namely StringCollection.

☑ **You should** reconsider the use of Hashtable. Instead, try other dictionary such as StringDictionary, NameValueCollection, HybridCollection. Hashtable can be used if less number of values is stored.

☑ When you are creating a collection type, **you should** implement IEnumerable so that the collection can be used with LINQ to Objects.

☒ **Do not** implement both IEnumerator<T> and IEnumerable<T> on the same type. The same applies to the nongeneric interfaces IEnumerator and IEnumerable. In other words, a type should be either a collection or an enumerator, but not both.

☒ **Do not** return a null reference for Array or Collection. Null can be difficult to understand in this context. For example, a user might assume that the following code will work. Return an empty array or collection instead of a null reference.

```
int[] arr = SomeOtherFunc();

foreach (int v in arr)

{

   ...

}
```

3.8 Structures

☑ **Do** ensure that a state where all instance data is set to zero, false, or null (as appropriate) is valid. This prevents accidental creation of invalid instances when an array of the structs is created.

☑ **Do** implement IEquatable<T> on value types. The Object.Equals method on value types causes boxing and its default implementation is not very efficient, as it uses reflection. IEquatable<T>.Equals can have much better performance and can be implemented such that it will not cause boxing.

3.8.1 Structures vs. Classes

☒ **Do not** define a struct unless the type has all of the following characteristics:

- It logically represents a single value, similar to primitive types (int, double, etc.).
- It has an instance size fewer than 16 bytes.
- It is immutable.
- It will not have to be boxed frequently.

In all other cases, you should define your types as classes instead of structs.

3.9 Classes

☑ **Do** use inheritance to express “is a” relationships such as “cat is an animal”.

☑ **Do** use interfaces such as IDisposable to express “can do” relationships such as using “objects of this class can be disposed”.

3.9.1 Fields

☒ **Do not** provide instance fields that are public or protected. Public and protected fields do not version well and are not protected by code access security demands. Instead of using publicly visible fields, use private fields and expose them through properties.

☑ **Do** use public static read-only fields for predefined object instances.

☑ **Do** use constant fields for constants that will never change.

☒ **Do not** assign instances of mutable types to read-only fields.

3.9.2 Properties

☑ **Do** create read-only properties if the caller should not be able to change the value of the property.

☒ **Do not** provide set-only properties. If the property getter cannot be provided, use a method to implement the functionality instead. The method name should begin with Set followed by what would have been the property name.

☑ **Do** provide sensible default values for all properties, ensuring that the defaults do not result in a security hole or an extremely inefficient design.

☒ **You should not** throw exceptions from property getters. Property getters should be simple operations without any preconditions. If a getter might throw an exception, consider redesigning the property to be a method. This recommendation does not apply to indexers. Indexers can throw exceptions because of invalid arguments. It is valid and acceptable to throw exceptions from a property setter.

3.9.3 Constructors

☑ **Do** minimal work in the constructor. Constructors should not do much work other than to capture the constructor parameters and set main properties. The cost of any other processing should be delayed until required.

☑ **Do** throw exceptions from instance constructors if appropriate.

☑ **Do** explicitly declare the public default constructor in classes, if such a constructor is required. Even though some compilers automatically add a default constructor to your class, adding it explicitly makes code maintenance easier. It also ensures the default constructor remains defined even if the compiler stops emitting it because you add a constructor that takes parameters.

☒ **Do not** call virtual members on an object inside its constructors. Calling a virtual member causes the most-derived override to be called regardless of whether the constructor for the type that defines the most-derived override has been called.

3.9.4 Methods

☑ **Do** place all out parameters after all of the pass-by-value and ref parameters (excluding parameter arrays), even if this results in an inconsistency in parameter ordering between overloads.

☑ **Do** validate arguments passed to public, protected, or explicitly implemented members. Throw System.ArgumentException, or one of its subclasses, if the validation fails: If a null argument is passed and the member does not support null arguments, throw ArgumentNullException. If the value of an argument is outside the allowable range of values as defined by the invoked method, throw ArgumentOutOfRangeException.

3.9.5 Events

☑ **Do** be prepared for arbitrary code executing in the event-handling method. Consider placing the code where the event is raised in a try-catch block to prevent program termination due to unhandled exceptions thrown from the event handlers.

☒ **Do not** use events in performance sensitive APIs. While events are easier for many developers to understand and use, they are less desirable than Virtual Members from a performance and memory consumption perspective.

3.9.6 Member Overloading

☑ **Do** use member overloading rather than defining members with default arguments. Default arguments are not CLS-compliant and cannot be used from some languages. There is also a versioning issue in members with default arguments. Imagine version 1 of a method that sets an optional parameter to 123. When compiling code that calls this method without specifying the optional parameter, the compiler will embed the default value (123) into the code at the call site. Now, if version 2 of the method changes the optional parameter to 863, then, if the calling code is not recompiled, it will call version 2 of the method passing in 123 (version 1's default, not version 2's default).

Good:

```
Public Overloads Sub Rotate(ByVal data As Matrix)
   Rotate(data, 180)
End Sub
```

```
Public Overloads Sub Rotate(ByVal data As Matrix, ByVal degrees As Integer)
   ' Do rotation here
End Sub
Bad:
Public Sub Rotate(ByVal data As Matrix, Optional ByVal degrees As Integer = 180)
   ' Do rotation here
End Sub
```

☒ **Do not** arbitrarily vary parameter names in overloads. If a parameter in one overload represents the same input as a parameter in another overload, the parameters should have the same name. Parameters with the same name should appear in the same position in all overloads.

☑ **Do** make only the longest overload virtual (if extensibility is required). Shorter overloads should simply call through to a longer overload.

3.9.7 Interface Members

☒ **You should not** implement interface members explicitly without having a strong reason to do so. Explicitly implemented members can be confusing to developers because they don't appear in the list of public members and they can also cause unnecessary boxing of value types.

☑ **You should** implement interface members explicitly, if the members are intended to be called only through the interface.

3.9.8 Virtual Members

Virtual members perform better than callbacks and events, but do not perform better than non-virtual methods.

☒ **Do not** make members virtual unless you have a good reason to do so and you are aware of all the costs related to designing, testing, and maintaining virtual members.

☑ **You should** prefer protected accessibility over public accessibility for virtual members. Public members should provide extensibility (if required) by calling into a protected virtual member.

3.9.9 Static Classes

☑ **Do** use static classes sparingly. Static classes should be used only as supporting classes for the object-oriented core of the framework.

3.9.10 Abstract Classes

☒ **Do not** define public or protected-internal constructors in abstract types.

☑ **Do** define a protected or an internal constructor on abstract classes.

A protected constructor is more common and simply allows the base class to do its own initialization when subtypes are created.

```
public abstract class Claim
{
   protected Claim()
   {
      ...
   }
}
```

An internal constructor can be used to limit concrete implementations of the abstract class to the assembly defining the class.

```
public abstract class Claim
{
   internal Claim()
   {
      ...
   }
}
```

3.10 Namespaces

☑ **Do** use the default namespaces of projects created by Visual Studio in All-In-One Code Framework code samples. It is not necessary to rename the namespace to the form of Microsoft.Sample.TechnologyName.

3.11 Errors and Exceptions

3.11.1 Exception Throwing

☑ **Do** report execution failures by throwing exceptions. Exceptions are the primary means of reporting errors in frameworks. If a member cannot successfully do what it is designed to do, it should be considered an execution failure and an exception should be thrown. **Do not** return error codes.

☑ **Do** throw the most specific (the most derived) exception that makes sense. For example, throw ArgumentNullException and not its base type ArgumentException if a null argument is passed. Throwing System.Exception as well as catching System.Exception are nearly always the wrong thing to do.

☒ **Do not** use exceptions for the normal flow of control, if possible. Except for system failures and operations with potential race conditions, you should write code that does not throw exceptions. For example, you can check preconditions before calling a method that may fail and throw exceptions. For example,

```
// C# sample:
if (collection != null && !collection.IsReadOnly)
{
   collection.Add(additionalNumber);
}
' VB.NET sample:
If ((Not collection Is Nothing) And (Not collection.IsReadOnly)) Then
   collection.Add(additionalNumber)
End If
```

☒ **Do not** throw exceptions from exception filter blocks. When an exception filter raises an exception, the exception is caught by the CLR, and the filter returns false. This behavior is indistinguishable from the filter executing and returning false explicitly and is therefore very difficult to debug.

```
' VB.NET sample
' This is bad design. The exception filter (When clause)
' may throw an exception when the InnerException property
' returns null
Try
  ...
Catch e As ArgumentException _
When e.InnerException.Message.StartsWith("File")
  ...
End Try
```

☒ **Do not** explicitly throw exceptions from finally blocks. Implicitly thrown exceptions resulting from calling methods that throw are acceptable.

3.11.2 Exception Handling

☒ **You should not** swallow errors by catching nonspecific exceptions, such as System.Exception, System.SystemException, and so on in .NET code. Do catch only specific errors that the code knows how to handle. You should catch a more specific exception, or re-throw the general exception as the last statement in the catch block. There are cases when swallowing errors in applications is acceptable, but such cases are rare.

```
Good:
// C# sample:
try
{
  ...
}
catch(System.NullReferenceException exc)
```

```
{
   ...
}
catch(System.ArgumentOutOfRangeException exc)
{
   ...
}
catch(System.InvalidCastException exc)
{
   ...
}


' VB.NET sample:
Try
   ...
Catch exc As System.NullReferenceException
   ...
Catch exc As System.ArgumentOutOfRangeException
   ...
Catch exc As System.InvalidCastException
   ...
End Try
Bad:
// C# sample:
try
```

```
{
   ...
}
catch (Exception ex)
{
   ...
}
```

```
' VB.NET sample:
Try
   ...
Catch ex As Exception
   ...
End Try
```

☑ **Do** prefer using an empty throw when catching and re-throwing an exception. This is the best way to preserve the exception call stack.

```
Good:
// C# sample:
try
{
   ... // Do some reading with the file
}
catch
{
```

```
    file.Position = position; // Unwind on failure
    throw; // Rethrow
}
' VB.NET sample:
Try
    ... ' Do some reading with the file
Catch ex As Exception
    file.Position = position ' Unwind on failure
    Throw ' Rethrow
End Try
Bad:
// C# sample:
try
{
    ... // Do some reading with the file
}
catch (Exception ex)
{
    file.Position = position; // Unwind on failure
    throw ex; // Rethrow
}

' VB.NET sample:
Try
    ... ' Do some reading with the file
```

```
Catch ex As Exception
   file.Position = position ' Unwind on failure
   Throw ex ' Rethrow
End Try
```

3.12 Resource Cleanup

☒ **Do not** force garbage collections with GC.Collect.

3.12.1 Try-finally Block

☑ **Do** use try-finally blocks for cleanup code and try-catch blocks for error recovery code. **Do not** use catch blocks for cleanup code. Usually, the cleanup logic rolls back resource (particularly, native resource) allocations. For example,

```
// C# sample:
FileStream stream = null;
try
{
   stream = new FileStream(...);
   ...
}
finally
{
   if (stream != null)
      stream.Close();
}
' VB.NET sample:
```

```
Dim stream As FileStream = Nothing
Try
   stream = New FileStream(...)
   ...
Catch ex As Exception
   If (stream IsNot Nothing) Then
      stream.Close()
   End If
End Try
```

C# and VB.NET provide the using statement that can be used instead of plain try-finally to clean up objects implementing the IDisposable interface.

```
// C# sample:
using (FileStream stream = new FileStream(...))
{
   ...
}
' VB.NET sample:
Using stream As New FileStream(...)
   ...
End Using
```

Many language constructs emit try-finally blocks automatically for you. Examples are C#/VB's using statement, C#'s lock statement, VB's SyncLock statement, C#'s foreach statement, and VB's For Each statement.

3.12.2 Basic Dispose Pattern

The basic implementation of the pattern involves implementing the System.IDisposable interface and declaring the Dispose(bool) method that implements all resource cleanup logic to be shared between the Dispose method and the

optional finalizer. Please note that this section does not discuss providing a finalizer. Finalizable types are extensions to this basic pattern and are discussed in the next section. The following example shows a simple implementation of the basic pattern:

```
// C# sample:
public class DisposableResourceHolder : IDisposable
{
   private bool disposed = false;
   private SafeHandle resource; // Handle to a resource
   public DisposableResourceHolder()
   {
      this.resource = ... // Allocates the native resource
   }
   public void DoSomething()
   {
      if (disposed)
         throw new ObjectDisposedException(...);
      // Now call some native methods using the resource
      ...
   }
   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }
```

```
    protected virtual void Dispose(bool disposing)
    {
        // Protect from being called multiple times.
        if (disposed) return;
        if (disposing)
        {
            // Clean up all managed resources.
            if (resource != null)
                resource.Dispose();
        }
        disposed = true;
    }
}
' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable
    Private disposed As Boolean = False
    Private resource As SafeHandle ' Handle to a resource
    Public Sub New()
        resource = ... ' Allocates the native resource
    End Sub
    Public Sub DoSomething()
        If (disposed) Then
            Throw New ObjectDisposedException(...)
        End If
```

```
        ' Now call some native methods using the resource
        ...
    End Sub
    Public Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub
    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        ' Protect from being called multiple times.
        If disposed Then Return
        If disposing Then
            ' Clean up all managed resources.
            If (resource IsNot Nothing) Then
                resource.Dispose()
            End If
        End If
        disposed = True
    End Sub
End Class
```

☑ **Do** implement the Basic Dispose Pattern on types containing instances of disposable types.

☑ **Do** extend the Basic Dispose Pattern to provide a finalizer on types holding resources that need to be freed explicitly and that do not have finalizers. For example, the pattern should be implemented on types storing unmanaged memory buffers.

☑ **You should** implement the Basic Dispose Pattern on classes that themselves don't hold unmanaged resources or disposable objects but are likely to have subtypes that do. A great example of this is the System.IO.Stream class. Although it is an abstract base class that doesn't hold any resources, most of its subclasses do and because of this, it implements this pattern.

☑ **Do** declare a protected virtual void Dispose(bool disposing) method to centralize all logic related to releasing unmanaged resources. All resource cleanup should occur in this method. The method is called from both the finalizer and the IDisposable.Dispose method. The parameter will be false if being invoked from inside a finalizer. It should be used to ensure any code running during finalization is not accessing other finalizable objects. Details of implementing finalizers are described in the next section.

```
// C# sample:
protected virtual void Dispose(bool disposing)
{
   if (disposing)
   {
      // Clean up all managed resources.
      if (resource != null)
         resource.Dispose();
      }
   }
}
' VB.NET sample:
Protected Overridable Sub Dispose(ByVal disposing As Boolean)
   ' Protect from being called multiple times.
   If disposed Then Return
   If disposing Then
      ' Clean up all managed resources.
      If (resource IsNot Nothing) Then
         resource.Dispose()
      End If
   End If
```

```
    disposed = True
End Sub
```

☑ **Do** implement the IDisposable interface by simply calling Dispose(true) followed by GC.SuppressFinalize(this). The call to SuppressFinalize should only occur if Dispose(true) executes successfully.

```
// C# sample:
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}
' VB.NET sample:
Public Sub Dispose() Implements IDisposable.Dispose
    Dispose(True)
    GC.SuppressFinalize(Me)
End Sub
```

☒ **Do not** make the parameterless Dispose method virtual. The Dispose(bool) method is the one that should be overridden by subclasses.

☒ Y**ou should not** throw an exception from within Dispose(bool) except under critical situations where the containing process has been corrupted (leaks, inconsistent shared state, etc.). Users expect that a call to Dispose would not raise an exception. For example, consider the manual try-finally in this C# snippet:

```
TextReader tr = new StreamReader(File.OpenRead("foo.txt"));
try
{
    // Do some stuff
}
finally
```

```
{
   tr.Dispose();
   // More stuff
}
```

If Dispose could raise an exception, further finally block cleanup logic will not execute. To work around this, the user would need to wrap every call to Dispose (within their finally block!) in a try block, which leads to very complex cleanup handlers. If executing a Dispose(bool disposing) method, never throw an exception if disposing is false. Doing so will terminate the process if executing inside a finalizer context.

☑ **Do** throw an ObjectDisposedException from any member that cannot be used after the object has been disposed.

```
// C# sample:
public class DisposableResourceHolder : IDisposable
{
   private bool disposed = false;
   private SafeHandle resource; // Handle to a resource
   public void DoSomething()
   {
      if (disposed)
         throw new ObjectDisposedException(...);
      // Now call some native methods using the resource
      ...
   }
   protected virtual void Dispose(bool disposing)
   {
      if (disposed) return;
      // Cleanup
```

```
        ...
        disposed = true;
    }
}
' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable
    Private disposed As Boolean = False
    Private resource As SafeHandle ' Handle to a resource
    Public Sub DoSomething()
        If (disposed) Then
            Throw New ObjectDisposedException(...)
        End If
        ' Now call some native methods using the resource
        ...
    End Sub

    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        ' Protect from being called multiple times.
        If disposed Then Return
        ' Cleanup
        ...
        disposed = True
    End Sub
```

```
End Class
```

3.12.3 Finalizable Types

Finalizable types are types that extend the Basic Dispose Pattern by overriding the finalizer and providing finalization code path in the Dispose(bool) method. The following code shows an example of a finalizable type:

```
// C# sample:
public class ComplexResourceHolder : IDisposable
{
   bool disposed = false;
   private IntPtr buffer; // Unmanaged memory buffer
   private SafeHandle resource; // Disposable handle to a resource
   public ComplexResourceHolder()
   {
      this.buffer = ... // Allocates memory
      this.resource = ... // Allocates the resource
   }
   public void DoSomething()
   {
      if (disposed)
         throw new ObjectDisposedException(...);

      // Now call some native methods using the resource
      ...
   }
   ~ComplexResourceHolder()
```

```
    {
        Dispose(false);
    }
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    protected virtual void Dispose(bool disposing)
    {
        // Protect from being called multiple times.
        if (disposed) return;
            if (disposing)
        {
            // Clean up all managed resources.
            if (resource != null)
                resource.Dispose();
        }
        // Clean up all native resources.
        ReleaseBuffer(buffer);
        disposed = true;
    }
}
' VB.NET sample:
Public Class DisposableResourceHolder
```

```
Implements IDisposable

Private disposed As Boolean = False

Private buffer As IntPtr ' Unmanaged memory buffer

Private resource As SafeHandle ' Handle to a resource

Public Sub New()

   buffer = ... ' Allocates memory

   resource = ... ' Allocates the native resource

End Sub

Public Sub DoSomething()

   If (disposed) Then

      Throw New ObjectDisposedException(...)

   End If

   ' Now call some native methods using the resource

   ...

End Sub

Protected Overrides Sub Finalize()

   Dispose(False)

   MyBase.Finalize()

End Sub

Public Sub Dispose() Implements IDisposable.Dispose

   Dispose(True)

   GC.SuppressFinalize(Me)

End Sub

Protected Overridable Sub Dispose(ByVal disposing As Boolean)

   ' Protect from being called multiple times.
```

```
        If disposed Then Return
        If disposing Then
          ' Clean up all managed resources.
          If (resource IsNot Nothing) Then
            resource.Dispose()
          End If
        End If
        ' Clean up all native resources.
        ReleaseBuffer(Buffer)
        disposed = True
      End Sub
    End Class
```

☑ **Do** make a type finalizable, if the type is responsible for releasing an unmanaged resource that does not have its own finalizer. When implementing the finalizer, simply call Dispose(false) and place all resource cleanup logic inside the Dispose(bool disposing) method.

```
    // C# sample:
    public class ComplexResourceHolder : IDisposable
    {
      ...
      ~ComplexResourceHolder()
      {
        Dispose(false);
      }
      protected virtual void Dispose(bool disposing)
      {
```

```
        ...
    }
}
' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable
    ...
    Protected Overrides Sub Finalize()
        Dispose(False)
        MyBase.Finalize()
    End Sub
    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        ...
    End Sub
End Class
```

☑ **Do** be very careful to make type finalizable. Carefully consider any case in which you think a finalizer is needed. There is a real cost associated with instances with finalizers, from both a performance and code complexity standpoint.

☑ **Do** implement the Basic Dispose Pattern on every finalizable type. See the previous section for details on the basic pattern. This gives users of the type a means to explicitly perform deterministic cleanup of those same resources for which the finalizer is responsible.

☑ **You should** create and use a critical finalizable object (a type with a type hierarchy that contains CriticalFinalizerObject) for situations in which a finalizer absolutely must execute even in the face of forced application domain unloads and thread aborts.

☑ **Do** prefer resource wrappers based on SafeHandle or SafeHandleZeroOrMinusOneIsInvalid (for Win32 resource handle whose value of either 0 or -1 indicates an invalid handle) to writing finalizer by yourself to encapsulate unmanaged resources where possible, in which case a finalizer becomes unnecessary because the wrapper is responsible for its own

resource cleanup. Safe handles implement the IDisposable interface, and inherit from CriticalFinalizerObject so the finalizer logic will absolutely execute even in the face of forced application domain unloads and thread aborts.

```
/// <summary>
/// Represents a wrapper class for a pipe handle.
/// </summary>
[SecurityCritical(SecurityCriticalScope.Everything),
HostProtection(SecurityAction.LinkDemand, MayLeakOnAbort = true),
SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]
internal sealed class SafePipeHandle : SafeHandleZeroOrMinusOneIsInvalid
{
   private SafePipeHandle()
      : base(true)
   {
   }
   public SafePipeHandle(IntPtr preexistingHandle, bool ownsHandle)
      : base(ownsHandle)
   {
      base.SetHandle(preexistingHandle);
   }
   [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success),
   DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
   [return: MarshalAs(UnmanagedType.Bool)]
   private static extern bool CloseHandle(IntPtr handle);

   protected override bool ReleaseHandle()
```

```
    {
        return CloseHandle(base.handle);
    }
}
/// <summary>
/// Represents a wrapper class for a local memory pointer.
/// </summary>
[SuppressUnmanagedCodeSecurity,
HostProtection(SecurityAction.LinkDemand, MayLeakOnAbort = true)]
internal sealed class SafeLocalMemHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    public SafeLocalMemHandle()
        : base(true)
    {
    }
    public SafeLocalMemHandle(IntPtr preexistingHandle, bool ownsHandle)
        : base(ownsHandle)
    {
        base.SetHandle(preexistingHandle);
    }
    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success),
    DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    private static extern IntPtr LocalFree(IntPtr hMem);

    protected override bool ReleaseHandle()
```

```
        {
            return (LocalFree(base.handle) == IntPtr.Zero);
        }
    }
```

☒ **Do not** access any finalizable objects in the finalizer code path, as there is significant risk that they will have already been finalized. For example, a finalizable object A that has a reference to another finalizable object B cannot reliably use B in A's finalizer, or vice versa. Finalizers are called in a random order (short of a weak ordering guarantee for critical finalization).

It is OK to touch unboxed value type fields.

Also, be aware that objects stored in static variables will get collected at certain points during an application domain unload or while exiting the process. Accessing a static variable that refers to a finalizable object (or calling a static method that might use values stored in static variables) might not be safe if Environment.HasShutdownStarted returns true.

☒ **Do not** let exceptions escape from the finalizer logic, except for system-critical failures. If an exception is thrown from a finalizer, the CLR may shut down the entire process preventing other finalizers from executing and resources from being released in a controlled manner.

Appendix A - Software Design Checklist – Form

Software Design Checklist:	
Agency Name	*Department of Behavioral Health and Developmental Services*
Project Name	
Phase/Release	
Date	

Contents of Checklist:

Criteria - "Only one selection per line item will be accepted".	Yes / No / NA
1. Documented system requirements are used as the basis for selecting a design methodology.	☐ ☐ ☐
2. Resources necessary to perform software design activities on the project (i.e., estimated staff, development tools) are identified.	☐ ☐ ☐
3. Using a documented design methodology identifies a software structure.	☐ ☐ ☐
4. System design entities, inputs, and outputs are derived from the software structure.	☐ ☐ ☐
5. Customer interfaces are designed in consultation with the system owner.	☐ ☐ ☐
6. A logical data model that describes the system's data control flow is constructed.	☐ ☐ ☐
7. A Functional Design Document is created and distributed to the project team members and the system owner.	☐ ☐ ☐
8. A Functional Design Review is performed.	☐ ☐ ☐
9. At least one In-Phase Assessment is performed before the Functional Design Phase Exit.	☐ ☐ ☐
10. A system architecture including hardware, software, database, and data communications structures is specified.	☐ ☐ ☐
11. An Analysis of Benefits and Costs (ABC) are conducted on several system architecture alternatives and are used as the basis for an architecture recommendation.	☐ ☐ ☐
12. Functional Design entities are used as the basis for creating system modules, procedures, and objects.	☐ ☐ ☐
13. A physical data model, based on the logical data model, is developed.	☐ ☐ ☐
14. A system design is approved and baselined.	☐ ☐ ☐
15. Changes to the system design baseline are managed and controlled.	☐ ☐ ☐

16. A System Design Document is created.	☐ ☐ ☐
17. A Critical Design Review is conducted.	☐ ☐ ☐
18. System design activities are reviewed with the project manager/leader both periodically and as needed.	☐ ☐ ☐
19. Software Quality Assurance/Improvement periodically reviews and/or audits software design activities and deliverables and reports the results.	☐ ☐ ☐

Appendix B - Deployment Assessment Checklist – Form

For Deployment of:	
Agency Name	*Department of Behavioral Health and Developmental Services*
Project Name	
Phase/Release	
Date	

Criteria - **"Only one selection per line item will be accepted".**	**Yes / No / NA**
a. Are system requirements documented?	☐ ☐ ☐
b. Have system requirements been reviewed and approved by the designated approvers?	☐ ☐ ☐
c. Has the system design been reviewed and approved by the designated approvers?	☐ ☐ ☐
d. Are software requirements documented?	☐ ☐ ☐
e. Have software requirements been reviewed and approved by the designated approvers?	☐ ☐ ☐
f. Has the software design been reviewed and approved by the designated approvers?	☐ ☐ ☐
g. Is there a Requirements Traceability Matrix indicating traceability between requirements, design, and testing?	☐ ☐ ☐
h. Do test planning documents that describe the overall planning efforts and test approach exist?	☐ ☐ ☐
i. Is testing, as specified in the test planning documents, complete?	☐ ☐ ☐
j. Are test results documented?	☐ ☐ ☐
k. Is product defect-free?	☐ ☐ ☐
l. Have all remaining defects been documented?	☐ ☐ ☐
m. Is product acceptance sign-off (e.g., Final Acceptance) complete?	☐ ☐ ☐
n. Is the product in compliance with documented security standards?	☐ ☐ ☐

Criteria - **"Only one selection per line item will be accepted".**	**Yes / No / NA**
o. Has the risk assessment been executed?	☐ ☐ ☐
p. Has the security plan been documented?	☐ ☐ ☐
q. Has there been a security review/test?	☐ ☐ ☐
r. Have planned configuration audits been executed?	☐ ☐ ☐
s. Have configuration audit results been documented?	☐ ☐ ☐
t. Have planned data creation/conversion activities been executed, or are they on schedule to be completed as planned?	☐ ☐ ☐
u. Have planned training activities been executed, or are they on schedule to be completed as planned?	☐ ☐ ☐
v. Are documents to be produced for the purpose of aiding in installation, support, or use of the product complete, published, and distributed, or are they on schedule to be completed, published, and distributed prior to deployment?	☐ ☐ ☐
w. Are transition to support activities complete, or are they on schedule to be completed as planned?	☐ ☐ ☐
x. Are activities for notifying stakeholders of the release on schedule to be completed as planned?	☐ ☐ ☐
y. Are activities to enable the operation and maintenance of the product on schedule to be completed as planned?	☐ ☐ ☐
z. Have site preparation activities been completed?	☐ ☐ ☐
aa. Have environment preparation activities (e.g., correct OS, memory, etc.) been completed?	☐ ☐ ☐
bb. Is the selected software technology for the project listed on the enterprise's technology catalog, or has the appropriate authority approved the exception?	☐ ☐ ☐
cc. If the project requires purchased application software products, are all license agreements complete?	☐ ☐ ☐

Criteria - **"Only one selection per line item will be accepted".**	**Yes / No / NA**
dd. If the project requires purchased application software products, are all maintenance agreements in place and documented?	☐ ☐ ☐
ee. If the project requires purchased software products, have those items been installed in the production environment and tested?	☐ ☐ ☐
ff. If the project requires purchased hardware products, have those items been installed and tested?	☐ ☐ ☐
gg. If the project requires purchased hardware products, has all base application software been installed and tested?	☐ ☐ ☐
hh. If the project requires purchased hardware products, are all maintenance agreements in place and documented?	☐ ☐ ☐
ii. Is the production environment staged and prepared for release of the product for operational use?	☐ ☐ ☐

Signature: _____ Date: _____

Appendix C - DBHDS (Central Office) Software Development Platform Inventory

Application Platform	Description	Sustainable	Upgradable	Upgrade Imminent
ASP.NET	ASP.NET is a unified Web development model that includes the services necessary to build enterprise-class Web applications.	✓		
VB.NET	Visual Basic .NET is the result of a significant rebuild of Visual Basic for the Microsoft .NET Framework.	✓		
Classic Visual Basic	Visual Basic is a third-generation event-driven programming language and integrated development environment (IDE) from Microsoft for its .COM programming model first released in 1991.		✓	
Classic ASP	Active Server Pages (ASP), also known as Classic ASP or ASP Classic, was Microsoft's first server-side script engine for dynamically generated web pages.		✓	
Access	Microsoft Access, also known as Microsoft Office Access, is a database management system from Microsoft that combines the relational Microsoft Jet Database Engine with a graphical user interface and software-development tools.			✓
C#	C# is a programming language encompassing imperative, declarative, functional, and component-oriented programming disciplines.	✓		
CRM 4.0	Microsoft Dynamics CRM is a customer relationship management software package developed by Microsoft.			✓
CRM 11.0	Microsoft Dynamics CRM is a customer relationship management software package developed by Microsoft.	✓		

- Sustainable: Applications built on these platforms meet current software engineering standards.

- Upgradable: Applications built on these platforms will need to be upgraded in the near future. No new applications will be built on these platforms.
- Upgrade Imminent: Applications built on these platforms will need upgrading immediately. No new applications will be built on these platforms.

.NET Framework

Microsoft developed the .NET Framework in the late 1990s, originally under the name of Next Generation Windows Services (NGWS). The .NET Framework is included with Windows Server 2008 and Windows Vista. Version 3.5-4.5 is included with Windows 7 and Windows Server 2008 R2, and can also be installed on Windows XP and Windows Server 2003. On 12 April 2010, .NET Framework 4 was released alongside Visual Studio 2010.

.NET Framework Architecture

Common Language Infrastructure (CLI)

The purpose of the Common Language Infrastructure (CLI) is to provide a language-neutral platform for application development and execution, including functions for exception handling, garbage collection, security, and interoperability. By implementing the core aspects of the .NET Framework within the scope of the CL, this functionality will not be tied to a single language but will be available across the many languages supported by the framework. Microsoft's implementation of the CLI is called the Common Language Runtime, or CLR.

JavaScript

JavaScript is a prototype-based scripting language that is dynamic. Its syntax was influenced by the language C. The key design principles within JavaScript are taken from the self and scheme programming languages. It is a multi-paradigm language supporting object-oriented and functional programming styles. JavaScript is use in applications outside of web pages; for example, PDF documents, site-specific browsers, and desktop widgets.

ASP.NET

ASP.NET Web pages, known officially as Web Forms, are the main building blocks for application development. Web forms are contained in files with ".aspx" extensions; these files typically contain static (X) HTML markups. ASP.NET aims for performance benefits over other script-based technologies (including classic ASP) by compiling the server-side code to one or more DLL files on a Web server. This compilation happens automatically when a web-page is requested. This feature provides the ease of development offered by scripting languages with the performance benefits of a compiled binary code.

Appendix D - References

The following references were used to develop the guidelines described in this document:

- *.Net Framework General Reference: Design Guidelines for Class Library Developers* – MSDN Online Reference
- *Code Complete* - McConnell
- *Writing Solid Code* - Macguire
- *Practical Standards for Microsoft Visual Basic* – Foxall
- *Practical Guidelines and Best Practices for Visual Basic and Visual C# Developers* – Balena & Dimauro

Revision History

Date	Rev	Description	Author
12/03/13	1.0	Initial Release	Mario Epps
05/12/14	2.0	Updated	Mario Epps