VB.NET Coding Standards
iInterchange Coding Specifications

First Published: Feb, 2004
Revised Publication: Sep, 2004

Note: This is a live document and as such is subject to change.

Table of Contents

1 Document Overview

1.1 Document Introduction

This document is a tool to aid in the implementation of solid development practices to ensure the creation of consist, readable, and functional code. The development rules, guidelines, and specification outlined in this document will benefit developers in the maintenance, bug fixing, upgrades, and performance of existing and continued coding efforts. Although this document does outline many rules and guidelines for development, it still allows for developer independence when creating algorithm, code flow, and code implementation. But, as one developer touches another developer's code, the bases of the code should look as if one developer had created it. Many of these rules and guidelines are probably being implemented. There may be rules that are used but not seen in the list. If this is the case, please notify iInterchange so that they may be added.

2 Specifications

2.1 Variables

All variables should be declared with a combination of both the scope prefix and declaration type prefix. A minimum of one single uppercase letter should be used to separate words of a variable name. Uppercase letters should never be used for the scope or declaration type of a variable declaration. The following is a format for variable declaration:

<scope prefix>_<declaration type prefix><Name> AS <datatype>
Examples:
Private pvt_intRecordCount as Integer
Public pub_strUserName as String

2.2 Scope Prefix

All variable scopes are 3 letter combinations in lowercase letters followed by an underscore.

2.2.1 Module

Private pvt_
Public pub_
Friend fnd_

2.2.2 Classes and Forms

Private pvt_
Protected pro_
Friend fnd_
Public no prefix

2.2.3 Methods

Dim no prefix

2.3 Declaration Type Prefix

All variable declarations types are 3 letter combinations in lowercase letters.

int Integer
i16 Int16
i32 Int32
i64 Int64
dbl Double
sng Single
bln Boolean
byt Byte
vnt Variant
str String
dat Date
hlp Help Provider
tlt ToolTip
cmn Context Menu
tbr Toolbar
sbr Statusbar
ofd Open File Dialog box
sfd Save File Dialog box
fdl Font Dialog box
cdl Color Dialog box
pbr ProgressBar
rtb RichTextbox
tbr TrackBar
enm Enumerator
typ Type
ctl Control
frm Form
btn Button
lbl Label
txt Textbox
mnu Menu
chk Checkbox
rad Radio
pic Picturebox
pdl Print Dialog box
ppd Print Preview Dialog box
ppc Print Preview Control
obj Object
cls Class
itf Interface
frm Form class
mod Modules
col Collection
iml ImageList
hlp Help Provider
gbx Groupbox
pan Panel
dgd Datagrid
lst Listbox
cbx Combobox
lvw Listview
tvw Treeview
tab Tabcontrol
dtp Date Time Picker
cnd Month Calendar
hsb Horizontal Scroll Bar
rst Recordset
tbl Table
cnn Connection
drw Datarow
dst Dataset
cmd Command
vsb Vertical Scroll Bar
tmr Timer
spl Splitter
dud Domain UpDown
nud Numeric UpDown

If a variable is of an array, then the above declaration types should be prefixed with an a. Example:

Public pub_aintScores(4) As Integer

2.4 Name

All variable names should be no fewer than 3 characters in length. Variable names should be meaningful and intuitive in their use. The use of x or I is not allowed as a name.

2.5 Constants

Constants are declared in all uppercase letters with the exception of the scope and declaration type prefixes. Constants do adhere to the rules of variable definition followed by the Name.

Example:
Public const pub_strHOME_URL as string = “http://URL”

3 Classes

All classes should be well thought out, compact in design, reusable, and well documented. The following is the format for class definition;

<scope prefix>_<Name> (Without return value)
<scope prefix>_<declaration type prefix><Name> AS <datatype> (With return value)
When a type is defined in the name of a class, refer to the variable definition for the valid values.

3.1 Definition

Class name should be meaningful and intuitive to use and should be at least 8 characters in length.

3.2 Inheritance

Classes should be built as compact and streamlined as possible and thought of as building blocks to be reused. Although all classes will not be inherited or bases classes for other classes, building reusable objects should always be the forethought in designing classes.

3.3 Polymorphism

The use of polymorphism is encouraged. Polymorphism by nature breaks objects down into independent entities, streamlines code, makes object smaller, helps to isolate bugs, and limit the time necessary for maintenance. The base class for any interface that is surrounded by a namespace should use the name of clsBase.

2.4 Design Patterns

The understanding and use of Design Patterns is beneficial in the development and implementation of solid class structures. Wherever possible, Envision suggests that the Design Pattern practices be utilized.

2.5 Error Classes

- There should be separate error objects defined for the following categories;
- DataAccessError (This includes File and DB)
- DataValidationError
- BusinessRulesError
- CalculationError
- XMLError
- AnError (This is for all areas that don't have a specific Error Class or for the transference from an error from an Exception or Error Class object that was caught.)
- BaseError (This is the base class for all above error objects)

Although they are very similar in their implementation and functionality, there will be cases wherethe error messages thrown will be unique to the specific environment for which they belong.

The BaseError class is an abstract class that defines the interface for all other error objects. Each of the above classes should inherit the BaseError class as its base class. Each of the above classes should have the following exposed methods:

- Text (overloaded for both a Set and Get)
- Erase

4 Methods

Method names should be meaningful and intuitive to use and should be at least 6 characters in length. Avoid the use of the words Get and Set at the beginning of a method name to separate methods.

Use overloading instead. Example:

Public pub_GetUserName() as String
Public pub_SetUserName(byval bv_Name as String)
In this case just use
Public pub_UserName()
Public pub_UserName(byval bv_Name) as string

4.1 Passed Parameters

Parameters should be passed in a manner for which they are being used. If the value of a
parameter is to be changed within the method then it should be passed by reference otherwise passed by value. When declaring parameters, they should be explicitly defined as either passed by reference or by value. A parameter declaration is as follows:

<Usage> <prefix>_<declaration type><Name> AS <datatype>

Examples:
ByRef br_strName as String
ByVal bv_strName as String

4.2 Indentation

Tab size and Indent size should be set to 4 spaces. This must be set the same for all developers who will be working on or sharing code.

4.3 Namespaces

Namespaces should be used to group a collection of related classes, forms, or other objects together. Using namespaces will help in understanding the relationships of objects that support each other.

4.4 Error Trapping

All Functions, Subroutines, and Properties with more than one line of code should have a TRY CATCH Block. Where possible, there should be 3 catch statements as a general rule;

- The first Catch should contain an explicit class designed for trapping the type of error expected to be thrown.
- The second should be the Catch clsBaseError.

- The third is the Catch Exception and is always the last of the Catch statements. It is used to catch unexpected errors.

Example;

```
Try
        Catch clsClasstype
                Create a clsAnError object
                Assign constMethodName
                Collect and Assign Error Information
                Throw clsAnError
        Catch clsBaseError
                Assign constMethodName

Collect and Assign Error Information

                Throw clsAnError
                Catch Exception
                Create a clsAnError object
                Assign constMethodName
                Collect and Assign Error Information
                Throw clsAnError
        End Try
```

Once an error has been trapped, the follow is the sequence for collecting the error information;

1. The constMethodName should be assigned to the error object.
2. Any error information should be assigned to the error object.
3. The appropriate throw statement should be executed when necessary.

The throw is done to send the error information up the calling stack to the highest level where it will be logged. An error should never be logged unless the object logging it is known to be the highest level object in the call stack.

5 Code Documentation

Documenting code is just as important as the code itself. Code that is well documented saves money, time, aggravation, and helps the maintenance of future upgrades and bug fixes.

5.1 Statements

Although not every coded statement needs to be documented and good solid code can be self documented, it is still important to document individual coded statements. The important key elements to keep in mind are lines of code that play a key role, if blocks, activity within a loop, and those lines of code that are not self documenting.

5.2 Variable Declarations

When declaring variables each variable should have at least one line of comment describing the use of the variable. If the variable is relied upon by or reliant upon another variable this should be noted.

5.3 Classes, Forms, and Modules

The following documentation comments should go at the top of every class, form, or module that is created.

```
'**********************************************************************
' ' Name:
' NAME OF THE CLASS, FORM, or MODULE
' ' Purpose:
' CREATE A GOOD SOLID DESCRIPTION OF WHAT PURPOSE THE
' OBJECT SERVERS.
' ' Changes and Change Date
' NOTE ANY CHANGES TO THE OBJECT LIKE ADDED METHODS OR
' VARIABLES OR BUG FIXES.
'
'**********************************************************************
```

5.4 Methods

The following documentation comments should go at the beginning of every method.

```
'**********************************************************************
' ' Name:
' NAME OF THE FUNCTION, SUBROUTINE, OR PROPERTY
' ' Purpose:
' CREATE A GOOD SOLID DESCRIPTION OF WHAT PURPOSE THE
' METHOD SERVERS AND ANY RULES THAT GOVERNS THE USE OF THE
' METHOD.
' ' Parameters:
' IN
' NAME OF THE PARAMETER
' PURPOSE OF THE PARAMETER AND ANY VALUE RESTRICTIONS
' OUT
```

```
' NAME OF THE PARAMETER
' PURPOSE OF THE PARAMETER AND ANY VALUE RESTRICTIONS
'
' Return Value
' WHAT THE RETURN TYPE IS AND ANY VALUE RESTRICTIONS
' ' Changes and Change Date
' NOTE ANY CHANGES TO THE METHOD LIKE ADDED VARIABLES OR BUG
‘ FIXES.
‘
‘**********************************************************************
```

6 Rules

The following rule set has been collected over the years through various sources and developers experiences. It should be considered the norm with very few if any exceptions.

6.1 Single line "if…then" statements must not be used

In the interest of making source code more readable, single line "if…then" statements are to be avoided i.e. the 'End If' keyword must be used for all 'if' statements.

6.2 The : should not be used to separate statements

Each statement should be on its own line.

6.3 Variable declaration should be ordered in the following manner

Constants, Private, Protected, Public. When declared in a method the order should be Constants, Dim.

6.4 Constant declarations should be ordered in the following manner:

Private, Protected, Public.

6.5 Method bodies, whether functions, procedures must not exceed 50 lines

For readability and possible reuse, if a method becomes more than 50 lines of code, break the method up into smaller methods.

6.6 Property procedures must not exceed 5 lines

Property procedures should be short and quick executing code bits and generally limited to accessing private member variables.

6.7 Constants declaration type must be explicitly specified using "As" keyword

Although a constant can't be changed, it's still a good idea to define the constant as the data type it is using.

6.8 Constants must be correctly formed

<scope prefix><declaration type prefix><body> The body of a constant's name must always be uppercase and underscore characters are used to separate individual words.

6.9 Shared constants and enumerators should be placed in grouped module files

Creating large module files of constants and enumerators is not permitted. Break up the

constants and enumerators into smaller module files. It is better to have to include many module files than to include a module file with many constants or enumerators that will not be used within a program.

6.10 All function, subroutines, and property methods must use "Try…Catch" blocks

for error trapping

To ensure good error trapping, a "Try…Catch" block must exist in all methods. Multiple catch statements are preferred if multiple type of error can be generated.

6.11 Using "Catch Exception" should be the last catch statement in a "Try…Catch" block

Envision prefers that specific classes are created for catching errors. These classes will inherit the Exception class as a base class. The specific classes should be listed first in the catch statements with the Catch Exception last.

6.12 Re-throw errors from called methods to the calling method

If an error is caught, it should be re-thrown to propagate up through the calling stack.

6.13 Allocate a unique DLL base address

To minimize memory usage and achieve optimal loading of DLLs, allocate a unique base address when building the component.

6.14 Always use "Option Explicit"

Use Option Explicit to force the declaration of variables.

6.15 Never use "Option Base"

The "Option Base" statement determines the default base index for an array. By default, the base index is 0 and should not be altered.

6.16 Early binding is preferred over late binding

By default , use early binding on components unless it's absolutely necessary to use late binding.

6.17 Explicitly declare variables using the object type. Do not use the object variable type

When creating variables of other object types like class, form, etc… use the explicit object name instead of the object data type and the CType() function.

6.18 Avoid "Lbound()," "Ubound," and "Count property" in "For…Next" loops

When iterating an array variable using a For…Next loop, assign a variable the value returned from a call to Lbound(), Ubound, or Count prior to commencement of the loop.

6.19 Avoid null string comparisons

When testing to see if a string is empty, use the Len() function instead of comparing a variable to “”

6.20 Use string character functions instead of variant character functions

Avoid the use of the variant “chr()”, “left()”, “mid()”, etc… functions; use “chr$()”, “left$()”, “mid$()”, etc… function instead.

6.21 Avoid the use of “+” operator for string concatenation

The “+” operator must not be used for string concatenation. Always use the “&.” All parameters to a method must be explicitly defined. When creating a parameter for a method, it must be explicitly defined with byval or byref, and use the “As” keyword to define the declaration type.

6.22 Return data types for function must be specified

Each function that returns a value should be explicitly defined even if a variant is going to be returned.

6.23 All functions, subroutines, and properties must have a single exit point

Having a single exit point simplifies debugging and can help to ensure that clean-up code is executed prior to exiting the method.

6.24 Optional parameters must always be assigned a default value

When creating optional parameters, be sure to assign them a default value to insure that they are not null or empty.

6.25 “IsMissing()” function should never be used

Since all optional parameters are assigned a default value, the “IsMissing()” function is not necessary.

6.26 The “Public” keyword for variables in classes should be used carefully

A private variable with a property for access is the preferred method of exposed variables in classes.

6.27 Put it on paper first

Throughout the design of a projects objects, data model, and algorithms should be thought out and designed, even if hand drawn, on paper first. Talk about issues with a peer to gain another perspective.

6.28 Avoid including “using” statements or references that are not necessary

Only include Using statements for items that are actually being used all others should be removed.

6.29 All classes must have a “New” and “Finalize” subroutine defined even if they are empty

Since the complier will put these subroutines in the code automatically it’s a good idea to go ahead and explicitly implement them in the editor.

6.30 All variable declarations are to be explicitly defined

When defining a variable there should always be a data type defined. Explicitly define the data type for which the variable will use even if it’s a variant, which is the default data type.

6.31 Never use “Redim” to change the declaration type of a variable

Redim should only be used to change the size of an array; but never change the data type of a variable.

6.32 Use “Redim” and “Preserver” sparingly

Redim/Preserves are very expensive due to the shifting of memory is required. If using a Redim/Preserve is necessary, do it in large blocks and use a counter to keep track of the actual size and shrink the size as the last step.

6.33 All property declarations must use the “As” keyword to define the type expectedto be used

Properties are the interface to private variables. As such, properties should use the AS keyword in the SET block and declaration and be defined with the same data type as the variable for which they are governing.

6.34 All functions, subroutines, and properties should have a “Try Catch” block

Error trapping is one of the most important parts of any successful development effort. Every function, subroutine, and property that has more than one of code, should have a “Try…Catch” block. Refer to the Error Trapping Section of this document for details.

6.35 All functions, subroutines, and properties need a constant with the method and class name

In order to create a proper audit trail for fixing bugs and identifying errors, the first line of every method will be as follows CONST [[[DLL Name], [EXE Name]], [COMPONENT Name], [METHOD Name]]
Example:
constMETHODNAME = “[Logging Controller.clsLogger.Write]”

The preceding tells us that something happened in the Logging Controller DLL in a class called clsLogger with a method called Write. Refer to the Error Trapping section (3.4) for usage.

6.36 Minimize the use of “.” separators to enhance readability and performance

When referencing an objects methods, properties, or operations the “.” should not be used more than twice. If the “.” is being used more than twice, Envision prefers that an assignment to a variable of the referenced item be used. This will allow for more

readable code and the code will execute faster as segment/offset references are expensive in terms of performance.

Example:
Here a XML stream has several nodes that have a firstchild reference.
xmlDoc.firstchild.firstchild.firstchild.selectSingleNode(“SOMENODE”)
The preferred syntax would be
xmlNode = xmlDoc.firstchild.firstchild
xmlNode = xmlNode.firstchild.selectSingleNode(“SOMENODE”)

6.37 Use FOREACH when iterating through collections

When iterating collections use the FOREACH interface instead of a standard FOR loop with counter variable control. FOREACH is more intuitive and executes faster than a standard FOR loop.

--End of Document--