China Construction Bank
Niederlassung Frankfurt

VB.NET Coding Standards

This document contains coding conventions and style guidelines that will ensure that VB.NET code will be of consistent and superior quality. It includes general coding guidelines as well as discussions on naming, formatting, and language usage patterns.

Contents

1. Introduction

1.1 Purpose

The purpose of this document is to provide coding style standards for the development source code written in VB.NET. Adhering to a coding style standard is an industry proven best-practice for making team development more efficient and application maintenance more cost-effective. While not comprehensive, these guidelines represent the minimum level of standardization expected in the source code of projects written in VB.NET.

1.2 Scope

This document provides guidance on the formatting, commenting, naming, and programming style of VB.NET source code and is applicable to component libraries, web services, web sites, and rich client applications.

1.3 Document Conventions

Example code is shown using the `Code` font and shows syntax as it would be color coded in Visual Studio's code editor.

2. VB.NET Golden Rules

The following guidelines are applicable to all aspects VB.NET development:

- Follow the style of existing code. Strive to maintain consistency within the code base of an application. If further guidance is needed, look to these guidelines and the .NET framework for clarification and examples.
- Make code as simple and readable as possible. Assume that someone else will be reading your code.
- Prefer small cohesive classes and methods to large monolithic ones.
- Use a separate file for each class, struct, interface, enumeration, and delegate with the exception of those nested within another class.
- Turn `Option Explicit` and `Option Strict` on for every project under Project | Properties | Common Properties | Build. These can be made the default options for all new projects by going to Tools | Options | Projects | VB Defaults.
- Don't use the `On Error Goto` or `On Error Next` statements. Use structured exception handling via `Try/Catch` blocks instead. They are the preferred method of performing error handling in .NET.
- Write the comments first. When writing a new method, write the comments for each step the method will perform before coding a single statement. These comments will become the headings for each block of code that gets implemented.
- Use liberal, meaningful comments within each class, method, and block of code to document the **purpose** of the code.
- Mark incomplete code with `' TODO:` comments. When working with many classes at once, it can be very easy to lose a train of thought.
- Never hard code "magic" values into code (strings or numbers). Instead, define constants, static read-only variables, and enumerations or read the values from configuration or resource files.
- Use the StringBuilder class and it's Append(), AppendFormat(), and ToString() methods instead of the string concatenation operator (+=) for large strings. It is much more memory efficient.
- Never present debug information to yourself or the end user via the UI (e.g. MessageBox). Use tracing and logging facilities to output debug information.
- Gaps and exceptions to these guidelines should be discussed and resolved with your application architect.

3. Formatting

3.1 Class Layout

Classes should be organized into regions within an application using a layout determined by your application architect. Consult your architect for the layout convention used in your application.

Example:

```
‘ Class layout based on functionality
Class RuleChecker

	#Region persistence

	#Region cache

	#Region mode

	#Region eval
End Class
```

Guidelines:

- Use the same convention consistently in all classes in an application.
- Omit regions if their associated class elements are not needed.
- The `Designer Generated Code` regions created by Visual Studio’s Visual Designer should contain only code generated by the designer and should not be manually modified. If you modify a method in this region (such as New or Dispose), move it out of the region.

3.2 Indicating Scope

Fully-qualify the names of types and type members only in order to resolve ambiguity or to emphasize that the referent is found in an unfamiliar place.

Example:

```
Public Sub alloc(host As mscoree.CorRuntimeHostClass)
	If (host <> Nothing) Then
		Me.host = new mscoree.CorRuntimeHostClass();
	End If
End Sub
Private Sub cleanup()
	If (host <> Nothing) Then
		System.Runtime.InteropServices.Marshal.ReleaseComObject(host);
	End If
End Sub
```

Guidelines:

- Don’t include the `Me` keyword before member accesses except to resolve ambiguity due to shadowing.
- Use Imports to avoid repetition of familiar namespaces

3.3 Indentation & Braces

Statements should be indented (using tabs) into blocks that show relative scope of execution. A consistent tab size should be used for all indentation in an application. This can be configured in Visual Studio using the settings in Tools | Options | Text Editor | Basic | Tabs.

Example:

```
Private Function CalculateDiscountedAmount( _
    ByVal amount As Decimal,_
    ByVal purchaseMethod As PurchaseMethod) As Decimal

    ‘ Calculate the discount based on the purchase method
    Dim discount As Decimal = 0.0
    Select Case purchaseMethod

        Case PurchaseMethod.Cash
            discount = Me.CalculateCashDiscount(amount)
            Trace.Writeline(“Cash discount of {0} applied.”, discount)

        Case PurchaseMethod.CreditCard
            discount = Me.CalculateCreditCardDiscount(amount)
            Trace.WriteLine(“Credit card discount of {0} applied.”, discount)

        Case Else
            Trace.WriteLine(“No discount applied.”)

    End Select

    ‘ Compute the discounted amount, making sure not to give money away
    Dim discountedAmount As Decimal = amount – discount
    If discountedAmount < 0.0 Then
        discountedAmount = 0.0
    End If
    LogManager.Publish(discountedAmount.ToString())

    Return discountedAmount

End Function
```

3.4 White space

Liberal use of white space is highly encouraged. This provides enhanced readability and is extremely helpful during debugging and code reviews. The indentation example above shows an example of the appropriate level of white space.

Guidelines:

- Blank lines should be used to separate logical blocks of code in much the way a writer separates prose using headings and paragraphs. Note the clean separation between logical sections in the previous code example via the leading comments and the blank lines immediately following.

3.5 Long lines of code

Comments and statements that extend beyond 80 columns in a single line can be broken up and indented for readability. Care should be taken to ensure readability and proper representation of the scope of the information in the broken lines. When passing large numbers of parameters, it is acceptable to group related parameters on the same line.

Example:

```
Private Function Win32FunctionWrapper( _
    ByVal arg1 As Integer, _
    ByVal arg2 As String, _
    ByVal arg3 As Boolean) As String

    ' Perform a PInvoke call to a win32 function,
    ' providing default values for obscure parameters,
    ' to hide the complexity from the caller
    If Win32.InternalSystemCall(vbNull, _
        arg1, arg2, _
        Win32.GlobalExceptionHandler, _
        0, _
        arg3, _
        vbNull) Then

        Win32FunctionWrapper = "Win32 system call succeeded."
    Else
        Win32FunctionWrapper = "Win32 system call failed."
    End If

End Function
```

Guidelines:

- When breaking parameter lists into multiple lines, indent each additional line one tab further than the starting line that is being continued.
- Group similar parameters on the same line when appropriate.
- When breaking comments into multiple lines, match the indentation level of the code that is being commented upon.
- When the additional indented lines of a multi-line statement are part of the beginning of an indented code block, use a blank line after the multi-line statement to clearly visually separate the multi-line statement from the first statement of the indented code block.
- Consider embedding large string constants in resources and retrieving them dynamically using the .NET `ResourceManager` class.

4. Commenting

4.1 End-Of-Line Comments

Use End-Of-Line comments only with variable and member field declarations. Use them to document the purpose of the variable being declared.

Example:

```
Private name As String = String.Empty ‘ User-visible label for control
Private htmlName As String = String.Empty ‘ HTML name attribute value
```

4.2 Single Line Comments

Use single line comments above each block of code relating to a particular task within a method that performs a significant operation or when a significant condition is reached.

Example:

```
‘ Compute total price including all taxes
Dim stateSalesTax As Decimal = Me.CalculateStateSalesTax(amount, Customer.State)
Dim citySalesTax  As Decimal = Me.CalculateCitySalesTax(amount, Customer.City)
Dim localSalesTax As Decimal = Me.CalculateLocalSalesTax(amount, Customer.Zipcode)
Dim totalPrice  As Decimal  = amount + stateSalesTax + citySalesTax + localSalesTax
Console.WriteLine(“Total Price: {0}”, totalPrice)
```

Guidelines:

- Comments should always begin with a single quote, followed by a space.
- Comments should document intent, not merely repeat the statements made by the code.
- Use an imperative voice so that comments match the tone of the commands being given in code.

4.3 ‘ TODO: Comments

Use the `‘ TODO:` comment to mark a section of code that needs further work before release. Source code should be searched for these comments before each release build.

5. Capitalization & Naming

5.1 Capitalization

Follow the standard Naming Guidelines established by the .NET framework team by using only three capitalization styles: **Pascal**, **Camel**, and **Upper** casing.

Examples:

Identifier Type	Capitalization Style	Example(s)
Abbreviations	Upper	ID, REF
Namespaces	Pascal	AppDomain, System.IO
Classes & Structs	Pascal	AppView
Constants & Enums	Pascal	TextStyles
Interfaces	Pascal	IEditableObject
Enum values	Pascal	TextStyles.BoldText
Property	Pascal	BackColor
Variables, and Attributes	Pascal (public) Camel (private, protected, local)	WindowSize windowWidth, windowHeight
Methods	Pascal (public, private, protected) Camel (parameters)	ToString() SetFilter(filterValue As String)
Local Variables	Camel	recordCount
Parameters	Camel	fileName

Guidelines:

- In **Pascal** casing, the first letter of an identifier is capitalized as well as the first letter of each concatenated word. This style is used for all public identifiers within a class library, including namespaces, classes and structures, properties, and methods.
- In **Camel** casing, the first letter of an identifier is lowercase but the first letter of each concatenated word is capitalized. This style is used for private and protected identifiers within the class library, parameters passed to methods, and local variables within a method.
- **Upper** casing is used only for abbreviated identifiers and acronyms of four letters or less.

5.2 Naming

Follow the standard set by the .NET framework team when it comes to naming. The 6. Programming section of this document provides naming templates for each construct within the VB.NET language. These templates can be used in conjunction with the tables provided in Appendix A. Naming Parts & Pairs to yield meaningful names in most scenarios.

6. Programming

6.1 Namespaces

Namespaces represent the logical packaging of component layers and subsystems. The declaration template for namespaces is: CompanyName.ProjectOrDomainName.PackageName.*SubsystemName*.

Examples:

```
Microsoft.Data.DataAccess
Microsoft.Logging.Listeners
```

Guidelines:

- Project and package level namespaces will normally be predetermined by an application architect for each project.
- Use Pascal casing when naming Subsystem namespaces.

6.2 Classes & Structures

Classes and structures represent the 'Nouns' of a system. As such, they should be declared using the following template: Noun + *Qualifier(s).* Classes and structures should declared with qualifiers that reflect their derivation from a base class whenever possible.

Examples:

```
CustomerForm
      Inherits Form
CustomerCollection
      Inherits CollectionBase
```

Guidelines:

- Use Pascal casing when naming classes and structures.
- Classes and structures should be broken up into distinct `#Regions` as previously described in the class layout guidelines.
- All public classes and their methods should be documented using single quoted comments above them. Use this comment style to document the purpose of the class and its methods.
- Default values for fields should be assigned on the line where the field is declared. These values are assigned at runtime just before the constructor is called. This keeps code for default values in one place, especially when a class contains multiple constructors.

6.3 Interfaces

Interfaces express behavior contracts that derived classes must implement. Interface names should use Nouns, Noun Phrases, or Adjectives that clearly express the behavior that they declare.

Examples:

```
IComponent
IFormattable
ITaxableProduct
```

Guidelines:

- Prefix interface names with the letter '`I`'.
- Use Pascal casing when naming interfaces.

6.4 Constants

Constants and static read-only variables should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Example:

```
Public Const DefaultValue As Integer = 25
Public Shared Readonly DefaultDatabaseName As String = "Membership"
```

Guidelines:

- Use Pascal casing when naming constants and static read only variables.
- Prefer the use of `Shared ReadOnly` over `Const` for public constants whenever possible. Constants declared using `Const` are substituted into the code accessing them at compile time. Using `Shared ReadOnly` variables ensures that constant values are accessed at runtime. This is safer and less prone to breakage, especially when accessing a constant value from a different assembly.

6.5 Enumerations

Enumerations should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Example:

```
' Enumerates the ways a customer may purchase goods.
<Flags()>Public Enum PurchaseMethod
   All = Not 0
   None = 0
   Cash = 1
   Check = 2
   CreditCard = 4
   DebitCard = 8
   Voucher = 16
End Enum
```

Guidelines:

- Use Pascal casing when naming enumerations.
- Use the `<Flags()>` attribute only to indicate that the enumeration can be treated as a bit field; that is, a set of flags.

6.6 Variables, Fields & Parameters

Variables, fields, and parameters should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Examples:

```
Dim lowestCommonDenominator As Integer = 10
Dim firstRedBallPrice As Decimal = 25.0
```

Guidelines:

- Use Camel casing when naming variables, fields, and parameters.

- Define variables as close as possible to the first line of code where they are used.
- Declare each variable and field on a separate line. This allows the use of End-Of-Line comments for documenting their purpose.
- Assign initial values whenever possible. The .NET runtime defaults all unassigned variables to 0 or null automatically, but assigning them proper values will alleviate unnecessary checks for proper assignment elsewhere in code.
- Avoid meaningless names like `i, j, k`, and `temp`. Take the time to describe what the object really is (e.g. use `index` instead of `i`; use `swapInt` instead of `tempInt`).
- Use a positive connotation for boolean variable names (e.g. `isOpen` as opposed to `notOpen`).

Atomic Types

- For atomic types, the first letter of the name defines the type, as follows:

Prefix	Type	Example
i	Integer or Long	iWordCount
f	Float (Single or Double)	fRadius
s	String	sFirstName
b	Boolean	bQuiet
c	Currency	cCurrentBal
d	Date	dStartTime

Consistency with standard Visual Basic and Windows API naming conventions. Allows easy identification of the type and role of a variable, without adding a lot of naming overhead for frequently used types.

Object and Complex Types

- For objects and complex types, and three-letter prefix defines the type, as follows:

Prefix	Type	Example
frm	Form	`frmCustInput`
txt	TextBox	`txtLastName`
lst	ListBox	`lstServiceTypes`
cmd	Command Button	`cmdCancel`
cbo	ComboBox	`cboCategory`
pic	PictureBox	`picLogo`
chk	CheckBox	`chkPrimaryAddress`
opt	Option Button	`optGenderFemale`
tmr	Timer	`tmrElapsed`
lbl	Label	`lblCopyright`
tbr	Toolbar	`tbrEditing`
ctl	Control (if type is not known)	`ctlSortableList`
col	Collection	`colFormFields`

obj	Object (if type is not known)	`objParent`
var	Variant (if type is not known)	`varNextField`

6.7 Properties

Properties should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Examples:

```
Public Property TotalPrice()
    Get
        TotalPrice = Me.totalPrice
    End Get
    Set(ByVal Value)
        ' Set value and fire changed event if new value is different
        If Not Me.totalPrice.Equals(Value) Then
            Me.totalPrice = Value
            Me.OnTotalPriceChanged()
        End If
    End Set
End Property
```

Guidelines:

- Use the common prefixes for inspection properties (properties that return query information about an object). See Appendix A. Naming Parts & Pairs for common prefixes.
- When there is a property setter that sets another property:
  - If the code in the other property sets a private member field in the same class, the field should be set directly, without calling the property setter for that field.
  - If a property setter sets a private field that would normally be set via another property setter, the originating setter is responsible for firing any events the other setter would normally fire (e.g. Changed events).
  - If a value that needs to be set that does NOT correspond to a private field, then an appropriate property setter or method should be called to set the value.

6.8 Methods

Methods should be named using the following format: Verb + *Adjective(s)* + Noun + *Qualifier(s)*

Example:

```
Private Function FindRedCansByPrice( _
    ByVal price As Decimal, _
    ByRef canListToPopulate As Integer, _
    ByRef numberOfCansFound As Integer) As Boolean
```

Guidelines:

- Related methods that have the same (or similar) parameter lists, should have the parameters in the same order.
- Avoid large methods. As a method's body approaches 20 to 30 lines of code, look for blocks that could be split into their own methods and possibly shared by other methods.

- If you find yourself using the same block of code more than once, it's a good candidate for a separate method.
- Group related methods within a class together into a region and order them by frequency of use (i.e. more frequently called methods should be near the top of their regions).
- Functions should use the `Return` keyword.

6.9 Event Handlers

Event handlers should be declared using the following format: ObjectName_EventName

Example:

```
Private Sub HelpButton_Click ( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles HelpButton.Click
```

6.10 Error Handling

Use exceptions only for exceptional cases, not for routine program flow. Thrown exceptions have significant performance overhead.

Guidelines:

- Pass a descriptive string into the constructor when throwing an exception.
- Use grammatically correct error messages, including ending punctuation. Each sentence in the description string of an exception should end in a period.
- If a property or method throws an exception in some cases, document this in the comments for the method. Include which exception is thrown and what causes it to be thrown.
    - Example: Comment for Order.TotalCost property might read "If the TotalCost property is set when the cost should be calculated, an `InvalidOperationException` is thrown."
- Use the following exceptions if appropriate:
    - `ArgumentException` (and `ArgumentNull`, `ArgumentOutOfRange`, `IndexOutOfRange`): Used when checking for valid input parameters to method.
    - `InvalidOperationException`: Used when a method call is invalid for the current state of an object.

      Example: TotalCost, a read/write property, cannot be set if the cost should be calculated. If it's set and it fails this rule, an `InvalidOperationException` is thrown.
    - `NotSupportedException`: Used when a method call is invalid for the class.

      Example: Quantity, a virtual read/write property, is overridden by a derived class. In the derived class, the property is read-only. If the property is set, a `NotSupportedException` is thrown.
    - `NotImplementedException`: Used when a method is not implemented for the current class.

Example: A interface method is stubbed in and not yet implemented. This method should throw a `NotImplementedException`.

- Derive your own exception classes for a programmatic scenarios. For web applications, new exceptions should be based derived from the core `Exception` class. For windows applications, new exceptions should be derived from `System.ApplicationException`.

  Example: `DeletedByAnotherUserException Inherits Exception`. Thrown to indicate a record being modified has been deleted by another user.

Appendix A. Naming Parts & Pairs

A.1 Common Adjective Pairs

Old…/New…
Source…/Destination…
Source…/Target…
First…/Next…/Current…/Previous…/Last…
Min…/Max…

A.2 Common Property Prefixes

Allow… (Allows…)
Can…
Contains…
Has…
Is…
Use… (Uses…)

A.3 Common Verb Pairs

Add…/Remove…	Open…/Close…
Insert…/Delete…	Create…/Destroy…
Increment/…Decrement…	Acquire…/Release…
Lock…/Unlock…	Up…/Down…
Begin…/End…	Show…/Hide…
Fetch…/Store…	Start…/Stop…
To…/From… (*Convert* implied)	

A.4 Common Qualifiers Suffixes

…Avg	…Limit
…Count	…Ref
…Entry	…Sum
…Index	…Total

Note: Avoid using **Num** because of semantics; use **Index** and **Count** instead. Also, avoid using **Temp**; take the time to describe what the object really is (e.g. use **SwapValue** instead of **TempValue**).

Appendix B. 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
- *The Elements of Java Style* – Vermeulen, et. al.
- *The Elements of C++ Style* – Misfeldt, et. al.

Revision History

Date	Rev	Description	Author
25.11.13	1.0	Initial Release	Paschalis Papas