VB.NET コーディング標準

中西庸文

初版 2002 年 7 月 5 日
改訂 1 版 2002 年 7 月 15 日
改訂 2 版 2002 年 7 月 22 日
改訂 3 版 2002 年 8 月 21 日
改訂 4 版 2002 年 10 月 1 日
改訂 5 版 2002 年 10 月 8 日

オリジナル：http://ObjectClub.esm.co.jp/eXtremeProgramming/CodingStd.doc

このドキュメントは Java コーディング標準(オブジェクト倶楽部バージョン)を VB.NET 用に変更したもので、フリーかつ AS-IS ベースで提供しています。

コピー、修正、配布してかまいません。みなさんのプロジェクトでこれをカスタマイズして使用することを歓迎します。

フィードバックを歓迎します。ご意見などを以下のアドレスへ頂けると嬉しく思います。

フィードバック先：
extremeprogramming-jp@ObjectClub.esm.co.jp
t-nakanishi@taiyo-sys.co.jp
http://db-appli.com/xoops/modules/newbb/viewforum.php?forum=9&22

1. 方針

このコーディング標準は、ソフトウェア開発プロジェクトにおいて VB.NET でコーディングする際のルール、推奨、および迷った時の指針を提供するものである。

標準策定の方針は、読みやすくメンテナンスしやすいコードを書くことである。実際のコーディングにあたっては、プロジェクトメンバー全員がこのルールに合意していることが必要である。

実プロジェクトにおいては、このコーディング標準をカスタマイズして用いることを推奨する。

また、あわせて .NET Framework SDK ヘルプ「クラス ライブラリ開発者向けのデザイン ガイドライン」を参照することを推薦する。

※ 特に Extreme Programming プロジェクトでの利用を意識したものではなく、あらゆる VB.NET を使ったプロジェクトでの利用を想定している。

2. ファイル構成

(1) ファイル名

Public クラスはそのクラス名の 1 ファイルにする。

例：Public Class Customer は、Customer.vb に入れる。

パッケージ内の非パブリッククラスは、そのクラスが主に使われるパブリッククラスのファイルに含めて良い。

例外クラスは、1 ファイルに複数クラスをまとめてもよい。

(2) ファイルの位置

プロジェクトのルートディレクトリを決め、名前空間の“.”をディレクトリ階層に置き換えた位置に入れる。

例：
名前空間：CompanyName.TechnologyName.FeatureName

ソリューションの配置先：C:\CompanyName\TechnologyName
プロジェクトの配置先：C:\CompanyName\TechnologyName\FeatureName

(3) テストクラス名

クラス ClassName のユニットテストクラス名は、ClassNameTest とする。ソリューション毎テストは SolutionNameTests とする。

例：Customer クラスなら CustomerTest.vb を作成
例：VbSample ソリューションなら、VbSampleTests.vbproj とする

理由：一貫性のある名前づけ。テストコードは使い方のサンプル、デモともなる。

(4) テストクラスの位置

テストクラスは、被テストクラスと同じ階層のディレクトリ "Tests" に配置する。

例：
被テストクラスの位置：C:\CompanyName\TechnologyName\FeatureName
テストクラスの配置：C:\CompanyName\TechnologyName\Tests

理由:物理的に近い位置でないとメンテが忘れ去られる。製品コードとの分離については、別のツール(NAnt の build ファイルなど)で調整可能。

3．命名規則

(5)　名前空間

企業正式名(.組織名).テクノロジー名.機能名を使用する。

```
Imports CompanyName(.OrganizationName).TechnologyName.FeatureName
```

理由:名前空間のトップレベルに企業正式名を指定することにより、競合の発生の可能性をさらに下げることができる。

(6)　ファイル名

パブリックなクラス名は、ファイル名と同じでなくてはならない。(大文字小文字の区別を含めて)

(7)　クラス名

先頭大文字。あとは区切りを大文字。

```
Class PascalCasing
```

(8)　例外クラス名

最後を Exception としたクラス名。

```
Class ClassNameEndsWithException
```

(9) 属性クラス名

カスタムの属性クラスには、最後に必ず **Attribute** を付ける。

```
Class ClassNameEndsWithAttribute
```

(10) インターフェイス名

クラス名に同じ。また常に最初に **I** を付ける。

```
Interface INameOfInterface
```

また、クラスにある能力を加える様な利用の場合、その能力を示す形容詞とし、**-able** を接尾にする。

例: **IEnumerable, ICloneable, IXmlSerializable, …**

(11) 実装クラス名

特に **Interface** と区別の必要があれば、最後に **Impl** を付ける。

```
Class ClassNameEndsWithImpl
```

(12) 抽象クラス名

抽象クラス名に適当な名前が無いとき、 **Abstract** から始まりサブクラス名を連想させる名前を付ける。

```
MustInherit Class AbstractBeforeSubClassName
```

(13) 定数(Const and ReadOnly)

先頭大文字。あとは区切りを大文字。

```
Const PascalCascading As Integer = 0
ReadOnly PascalCascading As Integer = 0
Shared ReadOnly PascalCascading As Integer = 0
```

定数と読み取り専用変数は異なる意味を持つ事に注意すること。(定数はコンパイル時に取得されるが、読取専用変数の値は実行時まで取得されない。)

(14) 列挙型(Enum)

先頭大文字。あとは区切りを大文字。

```
Enum PascalCasing
```

列挙型がビットフィールドを表すときは複数形にし、**FlagsAttribute** を付加する。

```
<Flags()> Enum PascalCasings
```

(15) 列挙値

先頭大文字。あとは区切りを大文字。

```
PascalCasing
```

(16) イベント名

先頭大文字。あとは区切りを大文字。

```
Event PascalCasing()
```

イベントの名前には動詞を使用すること。

イベントハンドラ名には、**EventHandler** サフィックスを付けること。

sender および **e** という名前の **2** つのパラメータを指定すること。

イベント引数クラスの名前には、**EventArgs** サフィックスを付けること。

イベント発生前の概念を表すイベント名には動名詞を使用し、イベント発生後の概念を表すイベント名には動詞の過去形を使用すること。

例:

```
Event EventName As EventNameEventHandler
Delegate Sub EventNameEventHandler(Byval sender As Object, _
                                   Byval e As EventNameEventArgs)

Event Closing As CancelEventHandler
Event Closed As EventHandler
```

(17) メソッド名

先頭大文字。あとは区切りを大文字。

```
Sub PascalCasing()
Function PascalCasing() As Object
```

(18) ファクトリメソッド(オブジェクトを new するもの)

```
Function NewX() As X
Function CreateX() As X
```

(19) コンバータメソッド(オブジェクトを別のオブジェクトに変換するもの)

```
Function ToX() As X
```

(20) プロパティ名

先頭大文字。あとは区切りを大文字。

```
Property PascalCasing() As Object
```

(21) Boolean 変数を返すメソッド

Is + 形容詞、Can + 動詞、Has + 過去分詞、三単元動詞、三単元動詞 + 名詞。

良い例：

```
Function IsEmpty() As Boolean
Function CanGet() As Boolean
Function HasChanged() As Boolean
Function Contains(ByVal x As Object) As Boolean
Function ContainsKey(ByVal key As String) As Boolean
```

悪い例：

```
Function Empty() As Boolean ‘「空にする」という動詞的な意味に取れる。
Function CheckXXX()As Boolean ‘ True がどちらの意味か分かりづらい。
```

理由： If, While 文等の条件が読みやすくなる。また True がどちらの意味か分かりやすい。

(22) Boolean 変数

形容詞、is + 形容詞、can + 動詞、has + 過去分詞、三単元動詞、三単元動詞 + 名詞。

```
Dim isEmpty As Boolean
Dim dirty As Boolean
Dim containsMoreElements As Boolean
```

(23) 英語と日本語

すべての識別子の名前は英語を基本とし、別に、日英対応用語辞書を作成してプロジェクトの全ライフサイクルでメンテナンスすること。

(24) 名前の対称性

クラス名、メソッド名を付ける際は、以下の英語の対称性に気を付ける。

```
Add / Remove
Insert / Delete
Get / Set
Start / Stop
Begin / End
Send / Receive
First / Last
Get / Release
Put / Get
Up / Down
Show / Hide
Source / Target
Open / Close
Source / Destination
Increment / Decrement
Lock / Unlock
Old / New
Next / Previous
```

(25) ループカウンタ

スコープ（通用範囲）が狭いループカウンタに **i, j, k** という名前をこの順に使う。

(26) スコープが狭い名前

スコープが狭い変数名は，型名を略したものを使って良い。

例： `Dim ds As DataSet = New DataSet()`

(27) 意味がとれる名前

変数名から役割が読み取れる名前を好め。

悪い例：`Sub Copy(ByVal s1 As String, ByVal s2 As String)`
良い例：`Sub Copy(ByVal source As String, ByVal destination As String)`

(28) 無意味な名前

`Info`, `Data`, `Temp`, `Str`, `Buf` という名前は再考を要する。

悪い例：`Dim temp As Double = Math.sqrt(b*b - 4*a*c)`
良い例：`Dim determinant As Double = Math.sqrt(b*b - 4*a*c)`

(29) Private / Protected / Friend / Protected Friend スコープのインスタンス変数

最初小文字で、あとは区切りを大文字。プリフィクス / サフィクスを適用する場合は、変数の読みやすさを考慮すること。ハンガリアン表記法は使用しない。

```
Private camelCasing As Object
Protected camelCasing As Object
Friend camelCasing As Object
Protected Friend camelCasing As Object
```

(30) Public スコープのインスタンス変数

先頭大文字。あとは区切りを大文字。極力使用しないようにすること。

```
Public PascalCasing As Object
```

(31) Private / Protected / Friend / Protected Friend スコープの共有変数

最初小文字で、あとは区切りを大文字。プリフィクス / サフィクスを適用する場合は、変数の読みやすさを考慮すること。

```
Private Shared camelCasing As Object
Protected Shared camelCasing As Object
Friend Shared camelCasing As Object
Protected Friend Shared camelCasing As Object
```

(32) Public スコープの共有変数

先頭大文字。あとは区切りを大文字。極力使用しないようにすること。

```
Public Shared PascalCasing As Object
```

(33) ローカル変数

最初小文字で、あとは区切りを大文字。ハンガリアン表記法は使用しない。

```
Dim camelCasing As Object
```

(34) 大文字小文字

大文字と小文字は同一として扱われるので注意が必要である。

例：インスタンス変数とプロパティが重なる場合。

```
ReadOnly Property Name() As Integer ' 不可
    Get
        Return name
    End Get
End Property
```

4. ガイドライン

(35) Option Strict

原則としてコンパイラの規定値として **Option Strict On** に設定し、型変換を明示的に行う。**Option Strict Off** にする必要性のある場合はファイルレベルで宣言する。

(36) Option Explicit

原則としてコンパイラの規定値として **Option Explicit On** に設定し、ファイル内のすべての変数を明示的に宣言する。

(37) #Region / #End Region ディレクティブ

コード領域は#**Region** / **#End Region** ディレクティブで宣言し、その領域についての説明を含める。

例：

```
#Region "インスタンス変数"

    Private name As String

#End Region

#Region "コンストラクタ"
```

```
        Public Sub New(ByVal name As String)
            Me.name = name
        End Sub

    #End Region
```

(38) メソッド/プロパティの宣言

メソッド/プロパティの宣言ではスコープを明示的に指定する。

(39) 長い行

一行は最大 100 桁とし、それを超える場合は行を分割する。分割の指針は、(1) ローカル変数を利用、(2)算術演算子/連結演算子で改行、(3)カンマで改行、(4)優先度の低い演算子の前で改行　とする。

例：

```
Dim length As Double = Math.Sqrt(Math.Pow(New Random().NextDouble, 2.0) +
Math.Pow(New Random().NextDouble, 2.0))

' 方針(1)
Dim xSquared As Double = Math.Pow(New Random().NextDouble, 2.0)
Dim ySquared As Double = Math.Pow(New Random().NextDouble, 2.0)
Dim length As Double = Math.Sqrt(xSquared + ySquared)

' 方針(2)
Dim length As Double = Math.Sqrt(Math.Pow(New Random().NextDouble, 2.0) + _
                                 Math.Pow(New Random().NextDouble, 2.0))

' 方針(3)
LongMethodSignature(value(0), value(1), value(2), _
                    value(3), value(4), value(5))
```

```
' 方針(4)
Return Me Is obj _
    Or (TypeOf obj Is Class1 _
        And Me.Field = obj.Field)
```

(40) 長いクラス宣言行

クラスの宣言が長い場合は Inherits / Implements 節で改行とする。

例:

```
Public Class LongNameClassImplemenation : Inherits AbstractImplementation :
Implements IXmlSerializable, Icloneable
```

ならば、

```
Public Class LongNameClassImplemenation
    Inherits AbstractImplementation
    Implements IXmlSerializable, Icloneable
```

属性によりクラスの宣言が長くなる場合は、カンマ、属性で改行する。

例：

```
<Serializable(), _
 System.Diagnostics.DebuggerStepThrough()> _
Public Class LongNameClassAttribute
```

(41) 長いメソッド宣言行

メソッドの宣言が長い場合はカンマで改行とする。

例：

```
Sub LongMethodSignature(ByVal a As Integer, ByVal b As Integer, _
                        ByVal c As Integer, ByVal d As Integer, _
                        ByVal e As Integer, ByVal f As Integer)
```

属性によりメソッドの宣言が長くなる場合は、カンマ、属性で改行する。

例：

```
<MethodImpl(MethodImplOptions.Synchronized), _
 System.Diagnostics.DebuggerStepThrough()> _
Sub LongNameMethodAttribute(ByVal a As Integer, ByVal b As Integer)
```

(42) MustInherit Class vs. Interface

抽象クラス(MustInherit Class)はなるべく使わず、Interface を多用せよ。MustInherit Class は、一部実装があり、一部抽象メソッドであるような場合にのみ使う。

理由：Interface は幾つでも継承できるが、Class は 1 つだけ。1 つから継承してしまうと、もう継承できずもったいない。

(43) Public Variable

インスタンス変数は、極力 Public にせず、妥当なプロパティを設ける。

理由：オブジェクト指向の標準。クラスの内部状態に勝手にアクセスさせるのはよくない。ただし、以下の条件をすべて満たす場合、インスタンス変数を Public にし、直接アクセスさせてもよい。

- そのインスタンス変数が他のインスタンス変数と独立であり、単独で変更されても内部の整合性をくずさない。
- どちらにしても，Get / Set プロパティを書く。
- インスタンス変数の実装が将来に渡って変更されないことが根拠付けられる。

また、上記に当てはまらない場合でも、極度に速度を気にする場合は、この限りではない。(ただし、慎重にコメントすること)

(44) 初期化

初期化をあてにしない(参照が Nothing に初期化されているとか)。また、２度初期化しない。

悪い例：

```
Public Class PoorInitialization

    Private name As String = "initial_name"

    Public Sub New()
        name = "initial_name"
    End Sub

End Class
```

理由：初期化に関するバグを最小化する。

(45) Shared 変数を避ける

Shared 変数(クラス変数)は極力避ける。

理由：Shared 変数は、セミグローバルと言って良い。より文脈依存なコードを招き、副作用を覆いかくしてしまう。

(46) Private vs. Protected

Private よりは、Protected を使用する。

理由：Private は確実にそのクラス外からの使用をシャットアウトできるが、クライアントが、より細かいチューニングを SubClass 化によって行うことを出来なくしてしまう。

別法: Private をより好んで使え。Protected にしてしまうと以降、変更が起ったときにそれを継承している全クラスに影響が出てしまう。

(47) Get / Set プロパティ

無闇にインスタンス変数へのプロパティ(Get / Set プロパティ)を作成して Public にすることは避ける。その必要性を検討し、もっと意味のあるプロパティ / メソッドにする。

理由：インスタンス変数は、他のインスタンス変数に依存していることが多い。クラス内部の整合性を崩してはならない。

(48) 変数隠し

基本クラスの変数名と、同じ変数名を使う事は避けよ。

理由：一般的にはこれはバグである。もし意図があるならコメントせよ。

(49) 配列宣言

配列の宣言は、arrayName() As Type とせよ。

例：

```
Public Shared Sub Main(ByVal args() As String)  --- ○
Public Shared Sub Main(ByVal args As String())  --- ×
```

(50) Public メソッド

クラスの Public メソッドは、「自動販売機のインターフェイス」を目標に。分かりやすく、使いかたを間違っても内部の整合性はこわれないように設計する。また、可能ならば契約による設計(Design by Contract)を行い、クラスの不変条件と共にメソッドの事前・事後条件をコードで表現せよ。

(51) 状態取得と状態変更の分離

メソッドは,「1 つの事」を行うように設計せよ。特に、状態変更と状態取得の 2 つのサー

ビスを１つのメソッドで行わない。状態を変更するメソッドは Sub にせよ.

理由 1：１つの事を行うメソッドの方が分かりやすい。

理由 2：並行性の制御、例外の安全保証がしやすい。

理由 3：サブクラス化による拡張がしやすい。

(52) Me の Return

クライアントの便宜を考えたつもりでも、Me を Return することはなるべく避ける。

理由：A.Meth1().Meth2().Meth3() というような連鎖は、一般的に Synchronization 上の問題の元になる。

(53) メソッドの多重定義

引数のタイプによるメソッドのオーバーロードはなるべく避ける(引数の数が違うものは OK である)。特に、継承と絡むと厄介である。

例：

×：
```
Overloads Sub Draw(ByVal rectangle As Rectangle)
Overloads Sub Draw(ByVal point As Point)
```

○：
```
Sub DrawRectangle(ByVal rectangle As Rectangle)
Sub DrawPoint(ByVal point As Point)
```

(54) Equals()と GetHashCode()

Object.Equals()メソッドをオーバーライドするなら、同時に GetHashCode()メソッドもオーバーライドせよ。逆も同じ。

理由：コンテナクラス(Hashtable)などに対応するため。

(55) Clone()

もし、**Clone()** メソッドが使われるようなら、**ICloneable** を実装し明示的にそれを書く。

例：

```
Imports System

Public Class Foo
    Implements ICloneable

    Public Function Clone() As Object Implements ICloneable.Clone
        Dim myFoo As Foo = CType(Me.MemberwiseClone, Foo)

        ' Foo クラスの属性のクローン
        '...
    End Function

End Class
```

理由：簡易コピーではよくないケースがほとんどである。

(56) デフォルトコンストラクタ

可能ならいつでもデフォルトのコンストラクタ(引数がないもの)を用意せよ。

理由：リフレクションを使用して、**Assembly.CreateInstance(TypeName)**で型名文字列からダイナミックにそのクラスを作成可能。

(57) MustOverride Method in MustInherit Classes

MustInherit クラスでは、**no-op** のメソッドを書くより、明示的に **MustOverride** メソッドと宣言せよ。また、共有可能なデフォルトの実装を用意できるなら、それを **Protected** とし、サブクラスが **1** 行で処理を書けるようにせよ。

理由：.Net の IDE は、実装されていない MustOverride メソッドを検出できるため、単に実装を忘れていただけ、というバグを回避できる。

(58) オブジェクトの同値比較

オブジェクトの比較では Equals() メソッドを使い、= を使うな。特に、String の比較では = を使用してはならない。

理由：もし実装者が Equals()を用意しているなら、それを使ってほしくて実装したはず。また、String の比較では Option Compare Text に設定されていると大文字/小文字が区別されない。

理由：ユニットテストでは、AssertEquals が Equals() を利用しているため、簡単に同値テストが書ける。

(59) 宣言と初期化

ローカル変数は、初期値と共に宣言せよ。

理由：変数の値に関する仮定を最小化する。

悪い例：

```
Sub F(ByVal start As Integer)
    Dim i, j As Integer ' 初期値なしの宣言
    ' 多くのコード
    ' ...
    i = start + 1
    j = i + 1
    ' i, jを使う
    ' ...
End Sub
```

良い例：

```
Sub F(ByVal start As Integer)
```

```
    ' 多くのコード
    ' ...
    Dim i As Integer = start + 1
    Dim j As Integer = i + 1
    ' i, jを使う
    ' ...
End Sub
```

(60) ローカル変数の再利用は悪

ローカル変数を使い回しするより、新しいものを宣言して初期化せよ。

理由：変数の値に関する仮定を最小化する。
理由：コンパイラの最適化を助ける。

悪い例：

```
Sub F(ByVal n As Integer, ByVal delta As Integer)
    Dim i As Integer ' 初期値なしの宣言
    For i = 0 To n - 1
        ' iを使う
    Next

    For i = 0 To n - 1  ' またiを使う
        If (...) Then
            Exit For
        End If
    Next

    If (i <> n - 1) Then    ' 最後まで回ったかの判定にiを使っている
        ' ...
    End If

    i = n - delta * 2   ' またまた再利用
    '...
End Sub
```

良い例：

```
Sub F(ByVal n As Integer, ByVal delta As Integer)
    Dim i As Integer
    For i = 0 To n - 1
        ' iを使う
    Next

    Dim found As Boolean = False

    Dim j As Integer
    For j = 0 To n - 1
        ' jを使う
        If (...) Then
            found = True
            Exit For
        End If
    Next

    If (found) Then
        ' ...
    End If

    Dim total As Integer = n - delta * 2    ' 別の意味ある変数
    '...
End Sub
```

(61) 大小比較演算子

“<”、 “<=” を好んで使い、”>”、 “>=” はなるべく避けよ。

理由：大小の方向を統一し、右側を大きい方にすることで、混乱を避ける。

(62) キャスト

キャストは、できる限り TypeOf の条件文で囲め。

例：

```
Dim cx As C = Nothing

If (TypeOf x Is C) Then
    cx = CType(x, C)
Else
    EvasiveAction()
End If
```

理由：これで、「オブジェクトがそのインスタンスじゃなかったら?」とういことを常に考える癖がつく。ただし、キャスト出来ない場合がバグである、と判断できる場合は、この限りではない。

(63) 例外クラス

例外クラスは大域的な性格をもち、多用するとプログラムの流れを読みにくくしてしまうことを認識する。

例外クラスは、新たに作成するよりも、.NET クラスライブラリに含まれているものを利用できれば利用する。

例: IOException, FileNotFoundException, ArgumentException などは利用しやすい標準例外。

新たな例外の作成は、そのパッケージ全体のインターフェイスとして検討すること。

(64) メソッド引数の変更は悪

原則としてメソッドの引数は入力であり、出力としては使わないこと。すなわちメソッド内部で引数の状態を変更するメソッドを呼ばないこと。出力引数に新たなオブジェクトを

代入しないこと。

悪い例:

```
Sub MoveX(ByVal p As Point, ByVal dx As Integer)
    p.X = p.X() + dx ' 引数を変更している(なるべく避ける)
End Sub

Sub MoveX(ByVal p As Point, ByVal dx As Integer)
    Dim p As New Point(p.X + dx, p.Y)
    ' これは呼び出し側に伝わらない
End Sub
```

例外：パフォーマンスを気にする場合。

(65) メソッド引数の名前

メソッドの引数は、読みやすいものにすること。特に、インスタンス変数と重なった場合、Me を活用し、引数の読みやすさを犠牲にしないこと。

悪い例：

```
Sub Reset(ByVal x_ As Integer, ByVal y_ As Integer)
    x = x_
    y = y_
End Sub
```

良い例：

```
' 引数名を x_, y_ などとしない
Sub Reset(ByVal x As Integer, ByVal y As Integer)
    Me.x = x
    Me.y = y
End Sub

Sub Reset(ByVal x As Integer, ByVal y As Integer)
    _x = x
    _y = y
```

```
End Sub
```

(66) ToString()

ToString() メソッドは可能ならいつでも実装すること。

理由 **1**：**Console.WriteLine(Object)** でいつでもプリントできる。
理由 **2**：ユニットテスト等で失敗した場合の表示が分かりやすくなる。

(67) Select Case, If / Else の繰り返しは悪

Select Case 文で分岐する処理が現れた時には、よくない設計の兆候だと考え、ポリモーフィズムで実現できないか再考する。特に同じような **Select Case** が **2** 箇所以上現れたら、必ずポリモーフィズム、**FactoryMethod**, **Prototype** パターン等でリファクタリングすること。**If / Else** の連続も同様。さらに、**null** チェックを行う同様の **If** が多くの場所に現れたら、**NullObject** パターンの利用を検討せよ。

(68) 型の変換

一般的な型変換は、**CType**, **DirectCast** を利用する。またオブジェクト変数のランタイム型が指定された型と同じである場合は **CType** より実行時の性能のよい **DirectCast** を使用すること。データ型変換はデータ型変換関数を使用すること。

例：

```
Dim o As Object = 2.37
Dim i As Integer = CType(o, Integer)
Dim d As Double = DirectCast(o, Double) ' o のランタイム型はDouble
Dim j As Integer = CInt(d)
```

5. コメント

(69) 長いコメント

コメントが複数行に渡る場合は、最初の短い一文で何が言いたいかを書き、その後に長いコメントを付けること。またそのような長いコメントの必要性を感じたときには常に設計をもっとシンプルにできないかを再考し、積極的にリファクタリングすること。

(70) Design by Contract (契約による設計)

契約による設計を行うため、**Debug** オブジェクトの **Assert** メソッドを使用せよ。**Assert** メソッドを使って、契約を表現せよ。

例：

```
Imports System.Collections

Public Class MyStack
    Inherits Stack

    Private capacity As Integer
    Private itemCount As Integer

    Public Overrides Sub Push(ByVal obj As Object)
        Debug.Assert(Not obj Is Nothing)    ' 事前条件
        ' ...
        ' ...
        Debug.Assert(Me.Contains(obj))  ' 事後条件
    End Sub

    Public Function Invariant() As Boolean  '不変条件
        Debug.Assert(0 <= capacity)
        Debug.Assert(0 <= itemCount)
        Debug.Assert(itemCount <= capacity)
```

```
            Return True
        End Function

    End Class
```

注意：ユーザ入力チェックなどを **Assert** してはいけない。バグを捕まえるために **Assert** せよ。

6. パフォーマンス

(71) まず計測

パフォーマンス改善はまず計測から始めよ。当てずっぽうではだめ。

(72) New

.NET では、**New** は時間が掛かる。ヘビーなループの中で **New** が呼ばれる場合、必要ならば出力引数を用いる。

```
    Function GetX() As X
        Return New X(Me.value)
    End Function
```

が遅い場合、呼び出し側に **New** を任せ、

```
    Sub GetX(ByVal x As X)
        x.Value = Me.value
    End Sub
```

とせよ。

(73) 変数への Nothing の代入

使われない変数が大量に発生する場合、積極的に Nothing を代入せよ。特に、配列の要素(パフォーマンス要求が厳しい場合)。

理由：ガベージコレクションを助ける。

7. その他

(74) 自分で新しく作る前に相談

他人が作成したクラスに対するある操作が新たに必要となるとき、自分でそのクラスを Inherits して新たなクラスを作成したり、そのクラスをインスタンス変数として持つクラスを作成するより、まずそのクラスの作成者に相談すること。汎用的な形でその要望を満たしてくれれば、全体をコンパクトにできる。

(75) 複雑な設計は悪

設計で迷った場合、多くのケースは ‘Simplicity’ を重視した方がよい。また、後のメンテナビリティにも ‘Simplicity’ は重要である。

(76) パフォーマンス調整は測定後

最初からパフォーマンスを気にしたコーディングをするべきではない。読みやすさ、保守のしやすさを優先する。パフォーマンスは測定してから改善する。

(77) トリッキーなコードは悪

.NET の平均プログラマに分かるようなコードを書く。演算子の順序、初期化に関する規則など、誰もが必ずしも自信をもって答えられないような仮定を持ち込まず、() を使って演

算順序を明確にしたり、明示的な初期化を行った方が読みやすい。

```
悪い例: Return CBool(IIf(cond = 0, a < b And b < c, d = 1))
良い例: Return CBool(IIf((cond = 0), (a < b) And (b < c), (d = 1)))
```

悪い例：

```
' 単位行列を作るが、時間もかかるし誰も読めない。
For i = 1 To i <= n
    For j = 1 To j <= n
        m(i - 1, j - 1) = (i / j) * (j / i)
    Next
Next
```

(78) 100%正しいことはない

ここに書かれていることに、100% 準拠する必要はない。迷ったら考えを整理し、相談すること。十分な理由があってルールから外れることはよくある。コミュニケーションができるチームの助けとなることが、このコーディング標準の目的である。

8. 謝辞

このコーディング標準をオリジナル版から VB.NET 版に変更するにあたって、平尾繁樹さん、小井土亨さん、穴見隆典さん、あまぴょんさん、よねＫＥＮさん、金野清隆さんから貴重なご意見を頂きました。ありがとうございました。また、オリジナルからの変更を快く了承して頂いた、平鍋健児さんに感謝致します。

9. 参考資料

Kenji Hiranabe, Java コーディング標準(オリジナル)
http://objectclub.esm.co.jp/eXtremeProgramming/CodingStd.doc

Writing Robust Java Code 高橋さんによる日本語訳版
http://www.alles.or.jp/~torutk/oojava/codingStandard/writingrobustjavacode.html

Writing Robust Java Code オリジナル(英語)
http://www.ambysoft.com/javaCodingStandards.html

プログラミング Microsoft .NET Framework
Jeffrey Richter 著 吉松 史彰 監訳 日経 BP ソフトプレス ISBN4-89100-303-0

Yoshihiro Kawabata, C# コーディング標準
http://www.kawabata.com/dotnet/CodingStdCS.pdf

クラス ライブラリ開発者向けのデザイン ガイドライン
http://www.microsoft.com/japan/msdn/library/default.asp?url=/japan/msdn/library/ja/cpgenref/html/cpconnamingguidelines.asp

SharpDevelop C# Coding Style Guide
http://www.icsharpcode.net/TechNotes/SharpDevelopCodingStyle.pdf

以上