Minimizando o impacto do uso concomitante de VB e C#

Por Marcio Belo

Desde que comecei a programar na plataforma .net em 2001, surgiu a dúvida cruel: qual linguagem usar? Eliminando opções menos comuns que a Microsoft atualmente nos apresenta – como Perl e Cobol (argh!) - a dúvida cruel no meu caso envolvia duas opções: VB – herdando a sintaxe de uma das mais antigas linguagens de programação – e a novíssima C# (leia-se c Sharp) – totalmente orientada a objeto e criada especialmente para a plataforma.

Meu objetivo aqui não será fornecer argumentos para uso de uma ou outra linguagem, até mesmo porque vários outros articulistas já escreveram a respeito e expuseram as vantagens de uma ou outra linguagem. Embora polêmico, parece existir um consenso entre os gurus em .net de que a escolha de VB ou C# é apenas uma questão de gosto. Concordo plenamente com esse ponto de vista, embora existam algumas funcionalidades que só podem ser obtidas com uma ou outra linguagem. Sobre essas diferenças, recomendo os artigos de Marco Bellisano de título "Visual Basic.NET or C#, that is the question" que pode ser encontrado no sítio DevX (www.devx.com).

O problema que ocorre, após três anos trabalhando na plataforma com projetos pilotos e "pra valer", é que me vejo com a obrigação de manter e/ou criar códigos em ambas as linguagens. Mesmo que na maior parte do tempo eu programe em VB, que no caso da minha empresa foi uma escolha natural por cultura, com freqüência tenho que criar ou, no mínimo, entender código de programação escrito em C#.

Minha constatação foi que era indispensável, para meu bom desempenho como desenvolvedor .net, conhecer ambas as linguagens. Sendo assim, comecei a estudar concomitantemente as duas linguagens, ora estudando VB.NET ora estudando C#. Não foi fácil pra mim e creio que o mesmo esteja ocorrendo com milhares de desenvolvedores mundo afora.

Entretanto, tenho algumas dicas que pode ajudar muito o uso das duas linguagens.

Use sempre o denominador comum do Class Library

Um exemplo clássico de biblioteca do Class Library que é normalmente usada apenas por programadores VB é a da namespace Microsoft.VisualBasic. Nela são encontradas funções e classes para compatibilidade com códigos escritos nas versões anteriores do Visual Basic. É bem verdade que essa namespace pode ser importada para uso num projeto C# mas, além de ser altamente desaconselhável fazê-lo, dificilmente você encontrará um projeto C# que o faça.

Por exemplo, a muito usada função MsgBox() encontra-se disponível nessa biblioteca (Microsoft.VisualBasic.MsgBox()). Essa função é apenas um atalho (wrapper) que invoca a funcionalidade oferecida pela chamada System.Windows.Forms.MessageBox.Show(). Por hábito, muitos programadores VB irão utilizar a função MsgBox da mesma forma como faziam no VB6. Evitar utilizar os wrappers disponibilizados no namespace Microsoft.VisualBasic fará com que você se habitue a usar o recurso "na fonte" e compatibilizará seu código com o que você faria em C#.

Uma forma eficaz de habituá-lo a seguir essa regra é configurar o projeto VB.NET para não facilitar o acesso às funções e classes desse namespace. Quando criamos um projeto VB.NET no VS, por padrão é adicionada a referência para essa namespace na seção Imports da janela de propriedade do projeto. Repare a janela de propriedades do projeto abaixo:

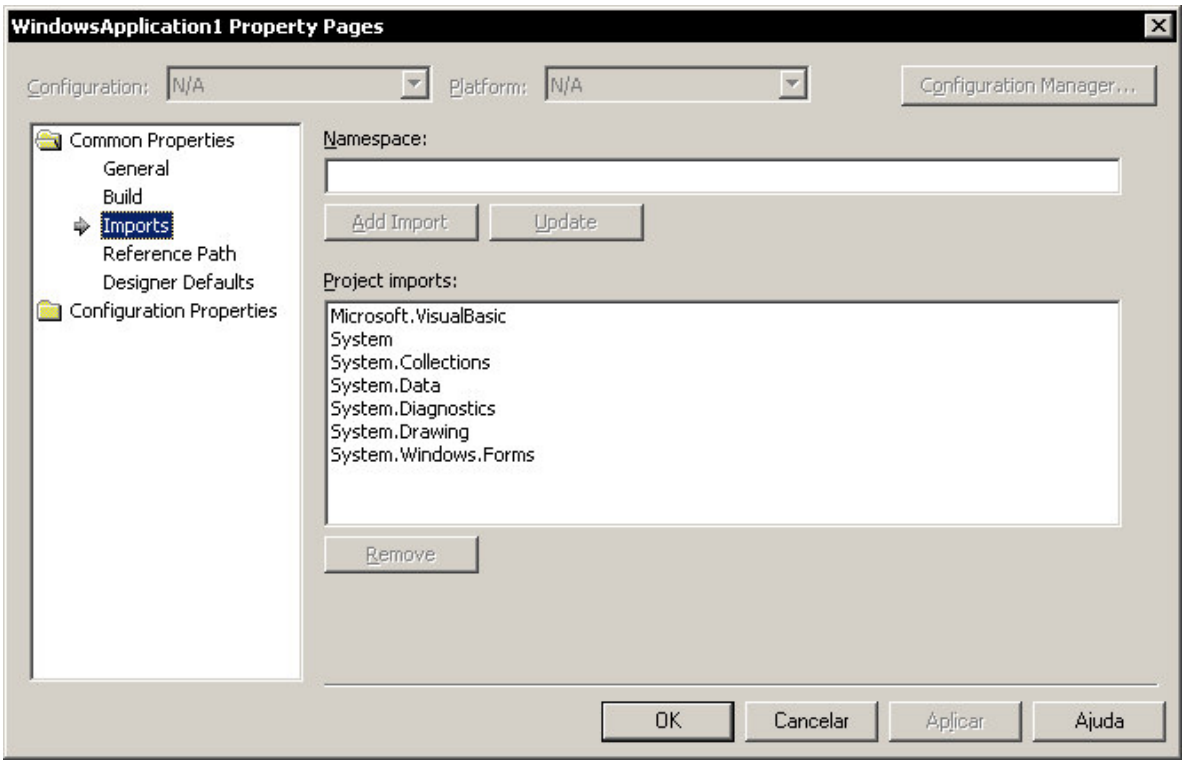


Figura 1 - Referência automática a namespace Microsoft.VisualBasic

Essa facilidade de referenciar namespaces para todo o projeto é bastante útil e está disponível apenas para projetos VB.NET. Uma vez feita essa referência, todos os arquivos de código contidos no projeto passam a considerar aquela namespace, como se tivéssemos referenciado a namespace com a cláusula *Imports* em cada arquivo de código. No C#, por não ter essa facilidade, devemos referenciar a namespace com a cláusula *using* em cada arquivo de código em que queiramos utilizá-la.

Sendo assim, para evitar que utilizemos alguma função ou classe por intermédio do wrapper ao invés da referência original na Class Library do .net, mesmo que sem querer, basta retirar a referência *Microsoft.VisualBasic* da lista Project Imports da janela de propriedades do projeto.

Use sempre declarações e conversões explícitas

No VB6 podíamos usar variáveis sem antes declará-las. Desta forma, as variáveis eram implicitamente declaradas como Variant. Essa prática, além de ir contra os mais básicos princípios de boa programação, gera situações de erro imprevisíveis em tempo de desenvolvimento, sem mencionar o desperdício de memória envolvido em utilizar um tipo de dado genérico como o Variant para muitas vezes utilizá-lo como um mero número inteiro.

Para meu desapontamento, o VB.NET continua a suportar essa prática porca de programação. O programador pode, ainda, criar programas onde as variáveis não tenham que ser explicitamente declaradas com a cláusula Dim. No caso do VB.NET, entretanto, as variáveis declaradas implicitamente são tipadas como Object, ao invés de Variant do VB6. O tipo Object é a mais básica das classes disponíveis no .net framework. Todas as outras classes são herdadas obrigatoriamente dela. O uso de uma classe genérica, ao invés da classe que de fato precisamos usar, posterga a verificação da consistência do nosso código para o momento da execução dele. Por exemplo, se usarmos o tipo Integer para uma variável que precisa armazenar números inteiros, ao invés de

usarmos o tipo Object, o compilador poderá verificar se uma operação que estamos realizando é permitida ou não, supondo que a conversão implícita esteja desligada. Considere o código abaixo:

```
Dim a As Integer
a = 10
a = a & a
```

Repare que, como estamos usando o Integer, o compilador indicará que a instrução de concatenação não é aplicada aquele tipo de dado.

No caso de declaração implícita, entretanto, um passo importante foi dado no VB.NET para melhorar nosso código: por padrão, as declarações implícitas estão desabilitadas. Confira pela figura 2 que, na janela de propriedades do projeto, a opção Option Explicit com o valor On, o que força a declaração explícita das variáveis.

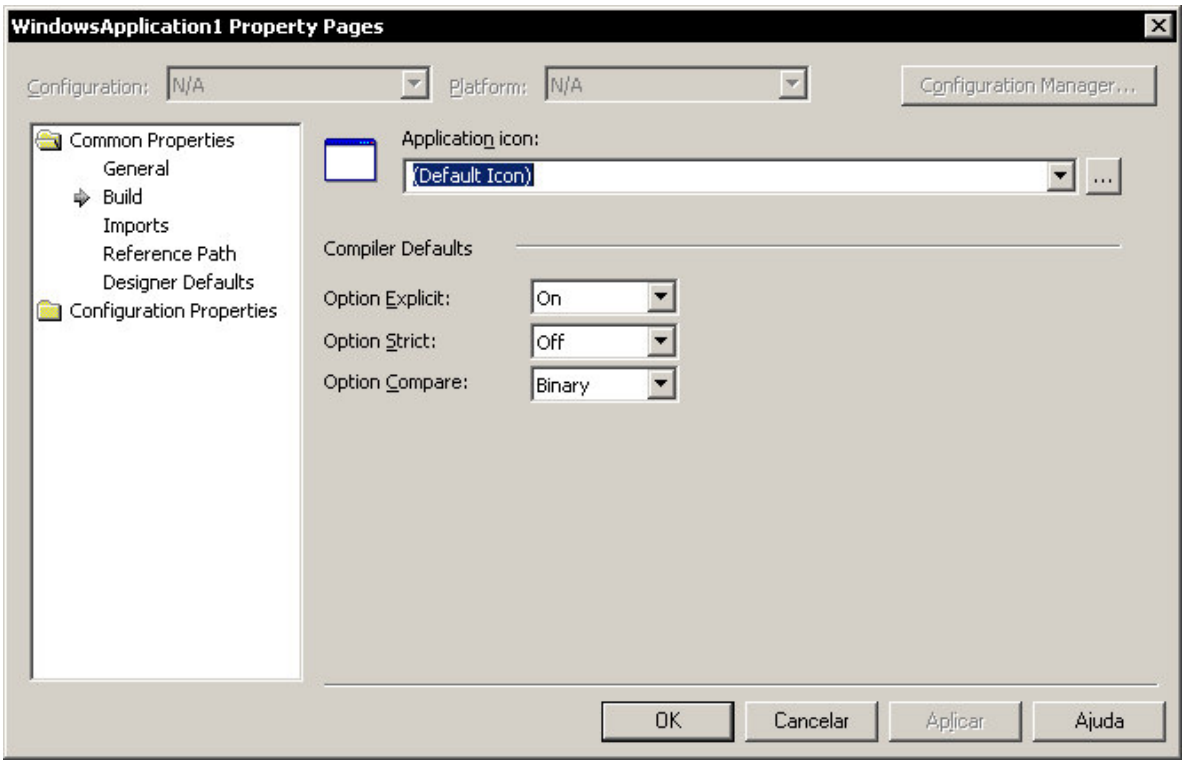


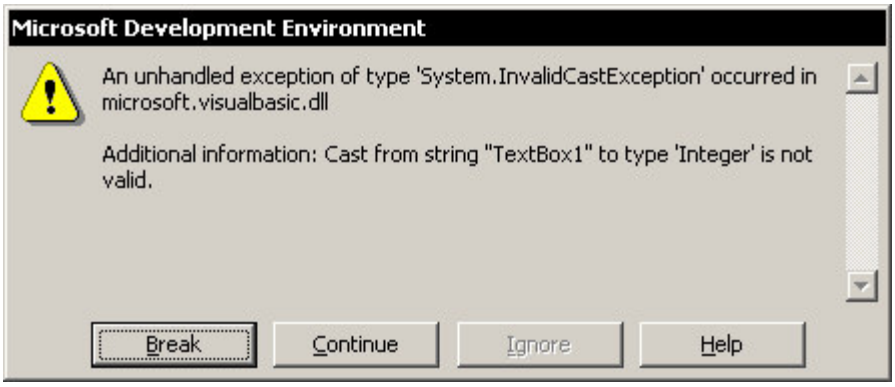
Figura 2 - Opções padrão de declarações e conversões implícitas

Embora as declarações implícitas sejam por padrão não permitidas em VB.NET, o mesmo não ocorre com as conversões. Novamente herdadas da antiga versão do VB, as conversões implícitas não são uma prática tão menos porca com as declarações. Comodidade para o programador, inferno para os usuários da aplicação. Novamente, ao usarmos conversões implícitas, estamos postergando para o tempo de execução a validação de nosso código.

Veja um exemplo a seguir:

```
Dim a As Integer
a = TextBox1.Text
```

O código acima é permitido pelo VB.NET por padrão. Ao atribuir o valor de uma String para uma variável Integer, a linguagem realiza a conversão de forma implícita. O perigo escondido por trás dessa prática é que ela habitua o programador a não se preocupar com situações de exceção que o código pode gerar. Se o usuário da aplicação acima digitar uma String que não pode ser convertida para Integer, a exceção abaixo será gerada:



No C# não existe declarações nem conversões implícitas. Para impedir que um programador desatento utilize esses malditos recursos, confira se as opções Option Explicit e Option Strict das propriedades do projeto estão com o valor On.

Use métodos sobrecarregados ao invés de argumentos opcionais

No VB6, podemos declarar que um ou mais parâmetros de um método (Sub ou Function) são opcionais. Isso facilita a vida do programador que usa o método, uma vez que ele precisa fornecer apenas os valores que são necessários. Entretanto, o programador que implementa o método deve codificá-lo de forma a usar valores padrões quando um ou mais argumentos não são fornecidos. No VB isso era feito usando a função IsMissing. No VB.NET houve uma pequena mudança: somos obrigados a indicar um valor padrão para cada um dos argumentos opcionais.

Entretanto, a mesma facilidade oferecida pelos argumentos opcionais pode, usando o estilo orientado a objetos de programar, ser usufruída através da sobrecarga de métodos. Veja os exemplos a seguir:

Modo VB6:

```
Public Sub CalcImposto(ByVal sal As Decimal, Optional ByVal aliquota As
Decimal = 10)
  '...
End Sub
```

Modo Orientado a Objetos:

```
Public Sub CalcImposto(ByVal sal As Decimal, ByVal aliquota As Decimal)
  '...
End Sub

Public Sub CalcImposto(ByVal sal As Decimal)
  CalcImposto(sal,10)
End Sub
```

Use o comando Return para retornar o resultado de uma função

No VB6, o retorno de uma função era feito atribuindo o valor resultado a uma pseudovariável de mesmo nome da função. No VB.NET podemos continuar a usar essa forma ou usar o comando Return. A diferença é que o comando Return sai da função de forma imediata, retornando o valor para o procedimento chamador. Na atribuição para a pseudovariável o fluxo de execução continua até que o fim da função seja alcançado e só então o valor é retornado.

Como no C# só existe retorno através do comando Return, prefira essa maneira para compatibilizar a codificação.

Não use sobrecarga de operadores permitida em C#

A sobrecarga de operadores em C# é um recurso muito poderoso, porém muito perigoso. Herdado de C++, a sobrecarga de operadores permite mudar o comportamento de um operador básico da linguagem como, por exemplo, o operador representado pelo símbolo +. A sobrecarga desse operador, aplicada a uma classe qualquer, poderia implementar um outro comportamento daquele tradicional, no caso, somar.

Como esse recurso não existe no VB, evite usá-lo sempre que possível.

Não use tratamento não estruturado de exceções

O VB.NET permite, por motivo de compatibilidade, o tratamento de exceções ao estilo VB6, usando a declaração On Error. Essa maneira é conhecida com tratamento de erros não estruturado e possui desvantagens que vão além do espoco desse artigo. A maneira O.O. de realizar tratamento de exceções é a estruturada, usando os comandos Try...Catch...Finally. O C# permite apenas o tratamento estruturado, portanto, não utilize o famigerado On Error para tratamento de erros quando estiver codificando em VB.NET.

Tratamento Não-Estruturado:

```
Public Sub Teste()
  Dim a, b, r As Double
  On Error GoTo TrataErro
  r = a / b
  Exit Sub ' Codigo dentro de uma Sub
TrataErro:
  MessageBox.Show("Divisão por zero.")
  Resume Next
End Sub
```

Tratamento Estruturado:

```
Public Sub Teste()
  Dim a, b, r As Double
  Try
    r = a / b
  Catch
    MessageBox.Show("Divisão por zero.")
  End Try
End Sub
```

Não use código não-gerenciado

Código gerenciado é como é chamado qualquer trecho de programação, no contexto .NET, que não rode sobre a máquina virtual (CLR) dele. O uso de trechos de código não-gerenciado, que podem ser mesclados com trechos gerenciados, é um recurso muito poderoso que só pode ser usado em C#.

Pela minha experiência, acho totalmente desaconselhável usar tal combinação numa mesma aplicação. Alguns articulistas argumentam que trechos não-gerenciados podem produzir resultados de performance impossíveis com os gerenciados. Esse ponto de vista é relativo e na minha opinião a vantagem alegada não justifica o perigo e complexidade do uso de tal mesclagem.

Considerações sobre os módulos do VB.NET

O VB.NET herdou do VB6 a facilidade dos módulos. Um módulo é uma classe estática, não pode ser herdada e todas as declarações feitas dentro dele são implicitamente estáticas. Para quem não conhece o conceito de classes e membros estáticos, sugiro a leitura de algum material sobre O.O. que trate do assunto.

Um módulo, portanto, seria o equilavente a criar uma classe estática e declarar todos os seus membros também como estáticos, da mesma forma como seria feito no C#. A grande vantagem do módulo é que os membros (variáveis e métodos) declarados dentro dele podem ser referenciados por todo o restante do código do projeto sem necessidade de qualificar o módulo no qual ele está declarado. Por exemplo, se utilizo um único objeto Connection (o que é muito comum) e quero que ele seja visto por todos os outros códigos do meu projeto, basta declarar a variável referente ao objeto citado dentro do módulo.

O problema dessa facilidade está no fato de que, ao utilizar o membro do módulo, o programador não sabe onde ele está declarado, diminuindo a clareza e dificultando a manutenção do código. Entretanto, considero que se houver apenas um módulo num projeto VB.NET e que, além da Sub Main, existam poucas declarações, acho valido o comodismo que ele proporciona.

Exemplo de utilização de módulos:

```
Module Principal
  Public sistema As String
End Module

Class Teste
  Public Sub UsaVariavel()
    sistema = "Alo Mundo!"
  End Sub
End Class
```

Marcio Belo é Analista de Sistemas e Professor de Computação e pode ser encontrado através da página http://www.mbelo.hpg.com.br