Aquí estación orbital C# llamando a estación base VB.NET ¿Me oyen?

Luis Miguel Blanco Ancos

O como el diseño multilenguaje de .NET Framework permite seguir usando nuestras funciones favoritas de Visual Basic desde otros lenguajes

Algunos apuntes sobre la arquitectura del sistema

Una de las especificaciones arquitectónicas de la plataforma .NET: CLS (Common Language Specification), establece que se trata de un sistema multilenguaje, significando esto que puede soportar la presencia de diversos lenguajes para ser programado. El CLS dicta las pautas que debe seguir cualquier lenguaje orientado a .NET (.NET aware language) para poder ser utilizado dentro de la plataforma.

Actualmente disponemos de varios lenguajes diseñados para la programación bajo .NET: Visual Basic .NET, C#, Visual C++, J#, etc.; pero cualquier fabricante de software, utilizando la especificación antedicha, puede acometer la creación de un nuevo lenguaje para añadirlo al conjunto de los ya existentes.

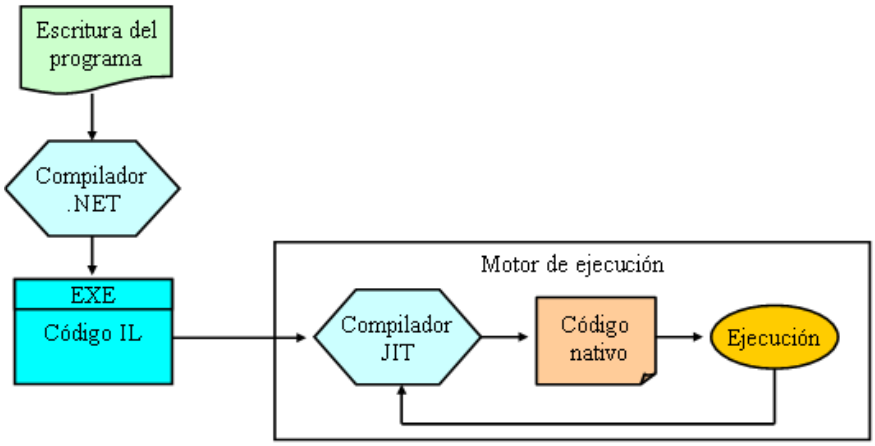
Este soporte multilenguaje se entiende a nivel de ensamblado, es decir, que podemos construir una aplicación articulando diversos ensamblados escritos en distintos lenguajes; lo que no está permitido es la creación de un ensamblado multilenguaje, o lo que es igual, escribir un programa o librería que tenga clases en distintos lenguajes, o una clase en la que se mezclen varios lenguajes.

El lenguaje intermedio. Punto de confluencia

Tras la introducción del punto anterior, el lector se preguntará por el modo en cómo .NET Framework se las ingenia cuando debe acoplar elementos desarrollados en distintos lenguajes, y conseguir además que la mezcla funcione armoniosamente. La clave la encontramos en el lenguaje intermedio MSIL (Microsoft Intermediate Language) o IL, como también le denominaremos en este artículo.

Cuando compilamos código en .NET utilizando el compilador correspondiente al lenguaje con el que hemos escrito dicho código, el resultado obtenido no es directamente código nativo de la plataforma hardware en la que va a ejecutarse la aplicación, como sucede con los compiladores tradicionales, sino código IL. Este código, como su propio nombre indica, se encuentra en un lugar intermedio entre el código que escribe el programador y el que precisa la máquina para ejecutar las instrucciones del programa.

Si no es código directamente ejecutable, ¿cómo consigue la plataforma .NET ejecutar el programa?. La respuesta está en un compilador inmediato o JIT (Just-In-Time), que en tiempo de ejecución se encarga dinámicamente de compilar el código IL del programa a código nativo de la plataforma sobre la que ha de ejecutarse. El siguiente diagrama muestra un esquema de este proceso de ejecución.



Esquema de ejecución de generación y ejecución del código IL

La compilación que lleva a cabo el JIT es, como hemos apuntado, dinámica, lo que quiere decir que no se compila todo el código IL del programa a código nativo cuando comienza su ejecución, sino según se va necesitando, con lo cual conseguimos un mejor rendimiento en el funcionamiento de las aplicaciones.

Llamadas entre lenguajes

Ya que nuestro código, independientemente del lenguaje con el que lo hayamos escrito, va a compilarse a un código común representado por el IL, las diferentes piezas que forman un proyecto: formularios, clases, librerías, etc., pueden estar escritas en diferentes lenguajes .NET, siendo resueltas de forma transparente las llamadas efectuadas entre las mismas.

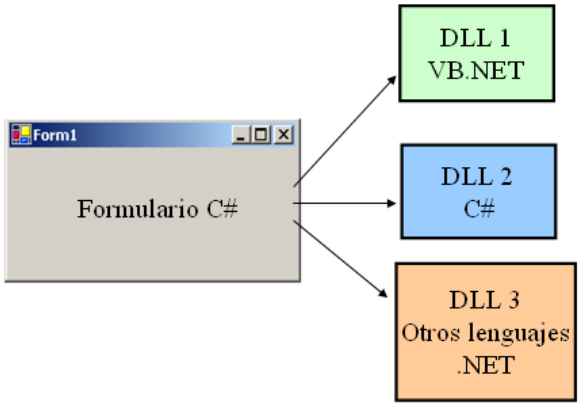


Diagrama de llamadas a ensamblados creados en distintos lenguajes

Las interioridades del código IL

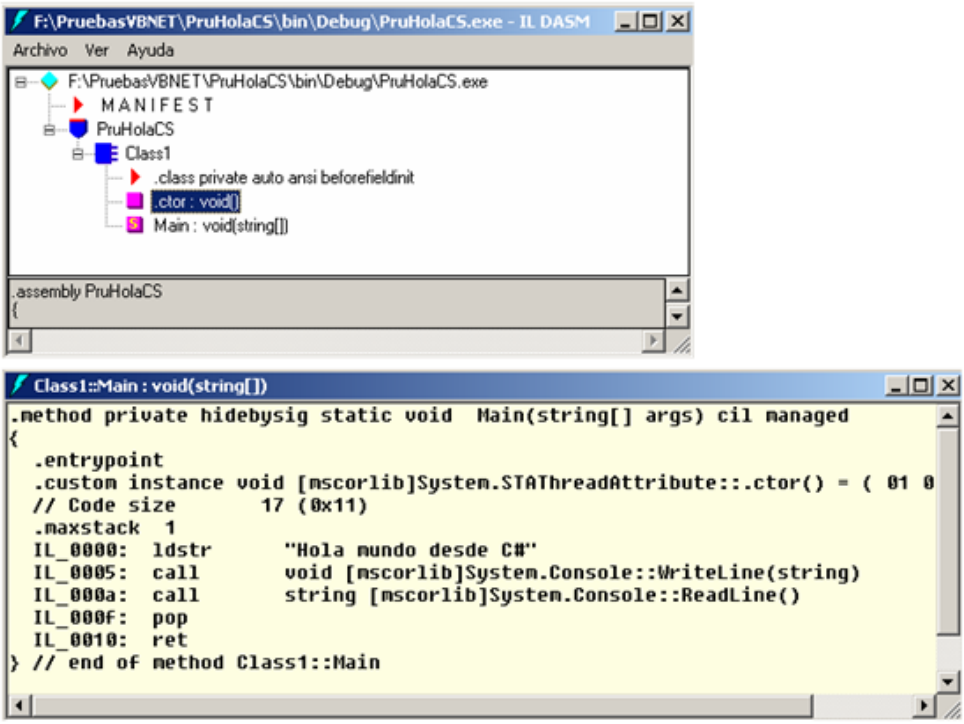
Después de tanto hablar del código intermedio, el lector posiblemente esté intrigado por la forma y contenido del mismo. .NET Framework proporciona una utilidad denominada ILDASM (Intermediate Language Disassembler), que como su nombre indica, se trata de un desensamblador de código intermedio, que nos permite su visualización dentro de un ensamblado (disculpe el lector este pequeño trabalenguas).

Para invocar esta utilidad, el modo más sencillo consiste en seleccionar desde el menú de Windows la opción *Microsoft Visual Studio .NET 2003 + Herramientas de Visual Studio .NET + Símbolo del sistema de Visual Studio .NET 2003*, que abrirá una ventana MS-DOS, en cuya variable de entorno PATH ya se encuentran establecidas las rutas a todas las herramientas de la plataforma.

A través del indicador de comandos ejecutaremos el archivo ILDASM.EXE, que mostrará una ventana en la que utilizando la opción de menú *Archivo + Abrir*, nos permitirá abrir un ensamblado de .NET (archivo .exe o .dll).

Una vez abierto el ensamblado, podemos visualizar su manifiesto y todas las clases de que está compuesto; si además, hacemos doble clic sobre alguno de estos elementos, veremos su código IL correspondiente.

La siguiente figura muestra un ensamblado escrito en C#, que visualiza en consola la cadena *Hola mundo desde C#*.



Visualizando un ensamblado desde la herramienta ILDASM

¿Nostalgia de las antiguas funciones de VB?

Aquellos que hayan trabajado ampliamente con VB conocerán de sobra el excelente conjunto de funciones que este lenguaje proporciona para el manejo de cadenas, fechas, etc.

Si nuestro estimado lector pertenece a este perfil de programador, pero por avatares del destino, se encuentra inmerso en un proyecto .NET a desarrollar con C#, no piense que ha perdido para siempre el uso de sus muy queridas funciones, que tan eficazmente le ayudaban en su trabajo cotidiano; sepa que gracias a la arquitectura multilenguaje de .NET, y en particular al código IL, estas funciones van a seguir estando disponibles, aunque programe en un lenguaje distinto a VB.

No obstante, la llamada a estas funciones desde un lenguaje diferente a VB, como en nuestro caso es C#, no es posible realizarla de un modo tan simple y directo como el que estamos acostumbrados, debido esto a la propia naturaleza y particularidades sintácticas del lenguaje llamador: C#

Comenzando a llamar a VB.NET desde C#

Cuando en VB.NET queremos, por ejemplo, convertir una cadena en mayúsculas utilizando la función Ucase, tan sólo hemos de llamar a esta función de un modo sencillo y directo, como muestra el siguiente código fuente.

```
Ucase("hola mundo")
```

Sin embargo, las cosas no son tan simples desde C#, ya que se trata de un lenguaje distinto de las funciones que necesitamos ejecutar; por ello vamos a ver cómo llamar desde este lenguaje a las clases y módulos de VB.NET, que son las entidades de código más representativas del lenguaje VB que contienen funciones y procedimientos. Comencemos por las clases, por ser un elemento con muchos aspectos en común entre ambos lenguajes.

En primer lugar crearemos un proyecto con el nombre ClaseVB, de tipo biblioteca de clases en VB.NET, en el que escribiremos el siguiente código.

```
Public Class Cliente
   Private msNombre As String

   Public Property Nombre() As String
```

```
    Get
      Return msNombre
    End Get
    Set(ByVal Value As String)
      msNombre = Value
    End Set
  End Property

  Public Function PasaMayusculas() As String
    Return UCase(msNombre)
  End Function

End Class
```

Tras escribir esta clase, seleccionaremos la opción *Generar solución* de VS.NET, para obtener el correspondiente archivo DLL con la clase compilada.

A continuación crearemos un proyecto en C#, de tipo consola, con el nombre LlamarClaseCS, que será el encargado de crear objetos de la clase Cliente escrita en VB.NET; para esto tendremos que establecer una referencia entre ambos proyectos o ensamblados de la siguiente manera.

Situados en el proyecto C#, desde la ventana *Explorador de soluciones* haremos clic derecho en el nodo References, seleccionaremos la opción *Agregar referencia*, y en el cuadro de diálogo del mismo nombre, pulsaremos el botón Examinar, con el que navegaremos por el disco hasta el directorio bin del proyecto ClaseVB, en el que se encuentra el archivo ClaseVB.dll; una vez localizado, lo seleccionaremos y aceptaremos el cuadro de diálogo, con esto ya tenemos referenciada la dll desde el código C#.

A partir de ahora, ya podemos instanciar objetos de la dll VB.NET, teniendo en cuenta que al crear el objeto es necesario especificar el espacio de nombres raíz del ensamblado VB.NET que contiene la clase a la que queremos acceder, este espacio de nombres corresponde al nombre del proyecto VB.NET. El siguiente código fuente muestra un ejemplo.

```
[STAThread]
static void Main(string[] args)
{
        ClaseVB.Cliente oCli;
        oCli = new ClaseVB.Cliente();
        oCli.Nombre="Luis Naranjo";
        Console.WriteLine(oCli.PasaMayusculas());
        Console.ReadLine();
}
```

Para simplificar la escritura de código en C#, también es posible declarar al comienzo del archivo de código el espacio de nombres mediante la palabra clave using, tal y como vemos a continuación.

```
using System;
using ClaseVB;

namespace LlamarClaseCS
{
        class Class1
        {
                [STAThread]
                static void Main(string[] args)
                {
                        Cliente oCli;
                        oCli = new Cliente();
                        oCli.Nombre="Luis Naranjo";
                        Console.WriteLine(oCli.PasaMayusculas());
                        Console.ReadLine();
                }
        }
}
```

En cualquier caso el resultado es el mismo: la ejecución del código VB.NET desde C#.

Llamadas a módulos de VB.NET

Como acabamos de comentar, la ejecución de clases de un ensamblado escrito en un lenguaje diferente al que estamos desarrollando nuestro proyecto no alberga misterio alguno. En este apartado vamos a dar un paso más, abordando un aspecto que quizá no resulte tan obvio: llamar desde C# a una función que se encuentra dentro de un módulo de VB.NET.

En primer lugar, y al igual que en el ejemplo anterior, crearemos en VB.NET un proyecto de biblioteca de clases con el nombre ModuloVB, del que eliminaremos la clase que tiene por defecto y agregaremos el módulo mostrado en el siguiente código fuente.

```
Public Module ManipCadenas
   Public Function DevLongitud(ByVal sCadena As String) As Integer
      Dim nLongitud As Integer
      nLongitud = Len(sCadena)
```

```
    Return nLongitud
  End Function
End Module
```

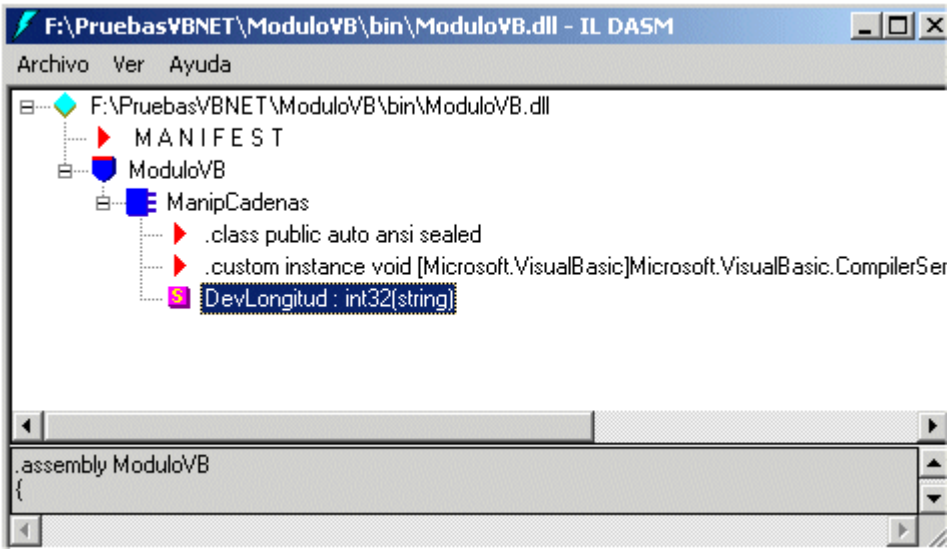
Es muy importante que declaremos el módulo con el modificador de ámbito Public, ya que de otra manera, no será accesible desde el exterior del ensamblado.

Seguidamente crearemos un proyecto de consola en C# con el nombre LlamarModuloCS, y estableceremos una referencia hacia el proyecto ModuloVB en la forma explicada anteriormente.

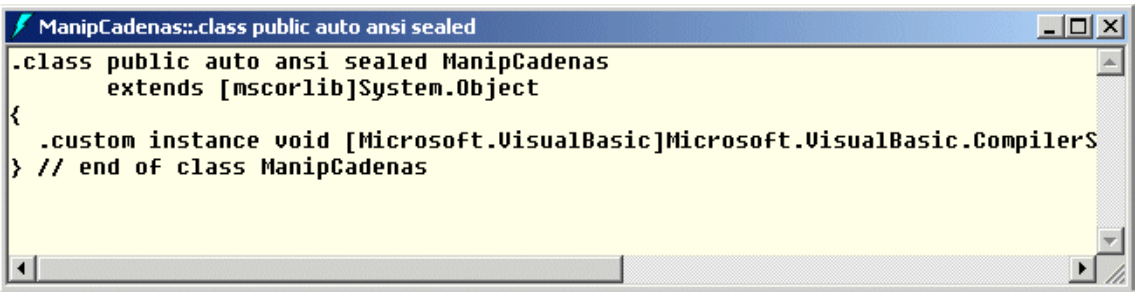
Y ahora viene la gran pregunta: ya que en el proyecto VB.NET tenemos la función DevLongitud dentro de un módulo y no en una clase, ¿cómo podemos llamarla desde C#?. Pues sorpresivamente, el modo de llamada es igual que si estuviéramos accediendo a un método compartido (Shared) de una clase, debiendo especificar, naturalmente, el espacio de nombres en el que se encuentra el módulo, bien en el momento de llamar a la función, o declarándolo mediante using. Veamos un ejemplo en el siguiente código fuente.

```
using System;
using ModuloVB;

namespace LlamarModuloCS
{
        class Class1
        {
                [STAThread]
                static void Main(string[] args)
                {
                        string Dato = "un poco de texto";
                        int LongDato1;
                        LongDato1 = ModuloVB.ManipCadenas.DevLongitud(Dato);
                        Console.WriteLine("Valor de LongDato1: {0}",LongDato1);

                        int LongDato2;
                        LongDato2 = ManipCadenas.DevLongitud(Dato);
                        Console.WriteLine("Valor de LongDato2: {0}",LongDato2);

                        Console.ReadLine();
                }
        }
}
```

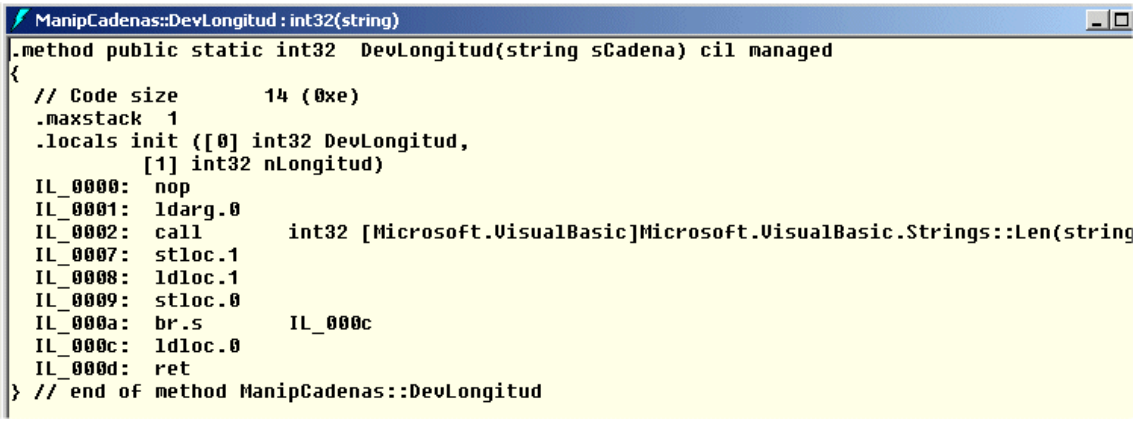
La causa de que podamos llamar a un módulo de esta forma radica en que al ser compilado, un módulo es convertido a una clase pública, y sus funciones a métodos compartidos. Para comprobar este hecho, sólo tenemos que abrir el ensamblado en el que está el módulo (en este ejemplo, el archivo ModuloVB.dll) con la utilidad ILDASM, y ver la información que nos ofrece, tal y como muestran las siguientes figuras.



Ensamblado VB.NET que contiene el módulo ManipCadenas visto desde ILDASM



Declaración del módulo ManipCadenas VB.NET en código IL



Declaración de la función DevLongitud perteneciente al módulo VB.NET en código IL

Acceso a las funciones clásicas de VB, el siguiente paso lógico

Y llegamos al punto que inicialmente ha dado origen a este artículo, la capacidad de llamar a las funciones tradicionales de VB desde C#.

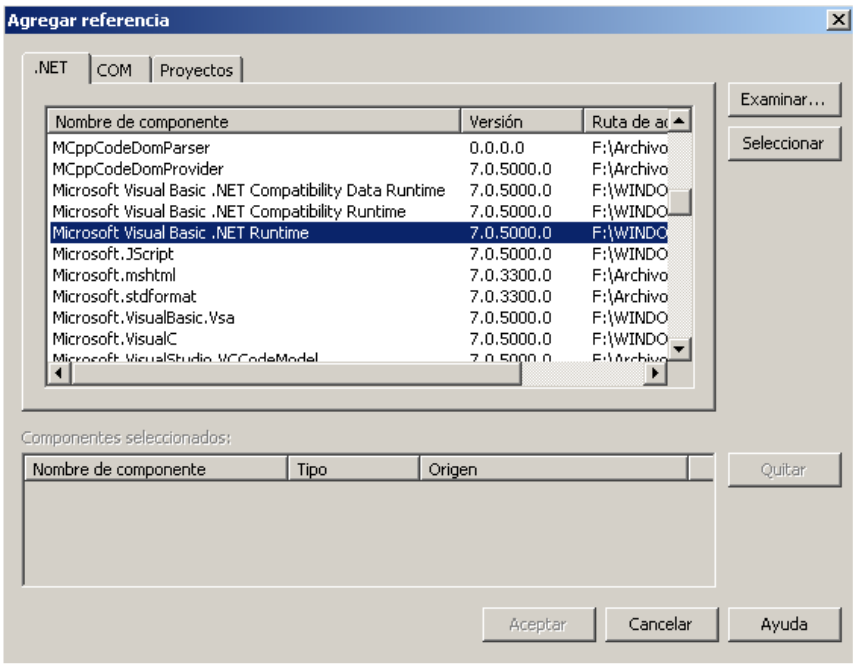
Si bien podíamos haber abordado este punto más directamente, esperamos que las explicaciones expuestas a lo largo de los anteriores apartados sirvan al lector para formar una base de conocimiento más sólida a la hora de acometer desarrollos en los que intervengan factores de este tipo; pero no nos desviemos de nuestro hilo argumental.

Como mencionábamos en uno de los primeros apartados, si somos unos devotos seguidores de las funciones *de toda la vida* de VB estamos de suerte, ya que desde .NET Framework podemos seguir usándolas puesto que se encuentran dentro del espacio de nombres Microsoft.VisualBasic, el cual contiene una serie de módulos que organizados en categorías, incluyen las funciones para las operaciones que ya realizábamos en VB6 y versiones anteriores del lenguaje. La siguiente tabla muestra una relación muy resumida de algunos de estos módulos y funciones. Consulte el lector la documentación de la plataforma .NET para una información más detallada.

Espacio de nombres Microsoft.VisualBasic	
Módulo	**Función**
Strings	Ucase
	LCase
	Len
	Mid
	Trim
DateAndTime	DateAdd
	DateDiff
	MonthName
	DatePart
FileSystem	Dir
	FileOpen
	Input
Interaction	MsgBox
	InputBox
	Beep
	Shell

Relación de módulos y funciones del espacio de nombres Microsoft.VisualBasic

Por lo tanto, y como ya conocemos la forma en que hemos de llamar desde C# a un módulo de VB.NET, todo lo que tenemos que hacer para conectar con estas funciones es establecer desde nuestro proyecto en C# una referencia al ensamblado Microsoft Visual Basic .NET Runtime, como se muestra en la siguiente figura.



Agregar referencia al ensamblado Microsoft.VisualBasic

A modo de ejemplo, vamos a crear un proyecto C# de tipo Aplicación Windows con el nombre LlamarFuncClasicasCS, y tras establecer la mencionada referencia, agregaremos un botón al formulario, en cuyo evento Click escribiremos el siguiente código fuente.

```
private void btnEjecutar_Click(object sender, System.EventArgs e)
{
        // manipulación de cadenas
        string sUnTexto;
        sUnTexto = Interaction.InputBox("Escribir algo de texto",
                "Usando funciones de VB","",25,25);

        if (Strings.Len(sUnTexto) > 0)
        {
                string sMayusculas = Strings.UCase(sUnTexto);
                Interaction.MsgBox("El texto en mayúsculas es: " +
                        sMayusculas,
                        MsgBoxStyle.Information,
                        "Llamando a VB");
        }
        else
        {
                Interaction.MsgBox("El texto está vacío",
                        MsgBoxStyle.Information,
```

```
                    "Llamando a VB");
        }

        // manipulación de fechas
        string sFecha = Interaction.InputBox("Introducir una fecha",
                "Usando funciones de VB","",25,25);

        if (Information.IsDate(sFecha))
        {
                DateTime dtFecha;
                dtFecha = Convert.ToDateTime(sFecha);
                dtFecha = DateAndTime.DateAdd("d",5,dtFecha);

                Interaction.MsgBox("Fecha resultante: " +
                        Strings.Format(dtFecha,"dddd dd/MMMM/yyyy"),
                        MsgBoxStyle.Information,
                        "Llamando a VB");
        }
        else
        {
                Interaction.MsgBox("No es una fecha válida",
                        MsgBoxStyle.Information,
                        "Llamando a VB");
        }
}
```

Como puede comprobar el lector, con la salvedad de tener que especificar el módulo al que pertenece la función llamada, el uso de la misma resulta tan familiar como si lo estuviésemos ejecutando desde un programa escrito en VB.NET.

Finalizando llamadas

Llegamos al final de nuestro recorrido. Esperamos que los comentarios y ejemplos expuestos a lo largo de este artículo sean de ayuda al lector a la hora de afrontar desarrollos en los que intervengan llamadas a código escrito en múltiples lenguajes.