Guillermo "Guille" Som

Usar componentes .NET desde aplicaciones COM

En este artículo veremos cómo crear componentes en .NET que se puedan usar desde aplicaciones que utilicen componentes COM. Pero no lo haremos de la forma fácil, sino que veremos cómo hacer que esos componentes funcionen de la misma forma que lo harían con cualquier entorno de desarrollo capaz de generar componentes COM (o ActiveX), de forma que tengan compatibilidad binaria, para no tener que volver a generar la aplicación cliente si decidimos modificar el componente.

» Crear componentes de .NET para usar desde aplicaciones COM

En el primer artículo de esta serie de artículos sobre interoperabilidad (**dotNetManía** nº 22) vimos cómo usar componentes COM desde aplicaciones creadas con .NET; en este último veremos el caso contrario, es decir, cómo crear componentes en .NET y usarlos desde aplicaciones COM. Debido a que Visual Basic 6.0 es uno de esos entornos de desarrollo que "sabe" manejarse con los componentes COM (o ActiveX), en las pruebas que haremos, usaremos aplicaciones cliente creadas con VB6, las cuales se incluyen en el ZIP con el código de ejemplo.

Debido a que Visual Studio 2005 ya es una realidad desde hace meses, y desde finales de enero lo es también en la versión en español, usaremos ese entorno para crear nuestros componentes .NET "compatibles" con COM. Y como es costumbre, el código que mostraremos a lo largo de este artículo, será en C#, pero en el ZIP con el código de ejemplo, también se incluye el equivalente creado con Visual Basic 2005, en el que veremos pequeños cambios con respecto a como funciona el código de C#, sobre todo en la parte referente al "lanzamiento" de eventos, ya que Visual Basic los maneja de forma más transparente de cómo lo hace C#.

En cualquier caso veremos dos formas de crear los componentes desde .NET, en la primera parte nos centraremos en crearlos sin prácticamente ningún tipo de configuración extra por nuestra parte, ya que dejaremos de la mano de Visual Studio todo lo necesario para la creación de los componentes; pero como he comentado al principio, el punto fuerte será crear componentes "decentes", es decir compo-

Componentes COM y compatibilidad binaria

Cuando trabajamos con componentes COM (ya sean DLL o controles ActiveX), es importante mantener la compatibilidad binaria entre el componente y el cliente que lo utiliza. Al mantener esa compatibilidad binaria nos aseguramos de que cualquier cambio que hagamos en el componente COM no requerirá tener que volver a compilar la aplicación que lo utiliza.

Esto es útil en los casos en que hacemos mejoras en el código del componente pero esos cambios no afectan a las interfaces expuestas por el mismo, de forma que la aplicación cliente seguirá usándolo de la misma forma que lo hacía antes de hacer esos cambios. Con lo cual nos evitamos tener que recompilar tanto el componente como la aplicación cliente.

Debemos recordar que COM se basa en interfaces, y es por medio de esas interfaces expuestas como "controla" la compatibilidad del componente con respecto al cliente que lo usa. Si cambia la interfaz (o interfaces) se debe crear una nueva versión del componente para permitir que los clientes que usaban las interfaces anteriores sigan funcionando como si nada hubiera pasado, y serán los nuevos clientes los que se aprovechen de esos cambios en las interfaces expuestas en el componente.

Estos detalles los tendremos en cuenta cuando hagamos nuestros componentes en .NET y los queramos utilizar desde aplicaciones por medio de la interoperabilidad COM.

Guillermo "Guille" Som es Microsoft MVP de Visual Basic desde 1997. Es redactor de dotNetManía, miembro de *Ineta Speakers Bureau Latin America*, mentor de *Solid Quality Learning* Iberoamérica y autor del libro *Manual Imprescindible de Visual Basic .NET*.
http://www.elguille.info

nentes que utilicen la compatibilidad binaria (ver nota sobre que significa eso de compatibilidad binaria), con idea de que le demos un aspecto más "profesional" a nuestros componentes .NET, de forma que, entre otras cosas, los clientes que los usen no tengan que volver a ser generados cada vez que hagamos cualquier modificación.

En este aspecto, hay que tener claro que Visual Studio (y por extensión .NET) no es un entorno COM, al menos en el sentido de que no es un entorno de desarrollo pensado para crear componentes COM, por tanto no nos avisará de que hemos roto la compatibilidad binaria, o lo que es lo mismo, no nos avisará de que las interfaces expuestas han cambiado con respecto a la última generación que hicimos del componente. Esto es algo que debemos gestionarlo nosotros. Este comentario está dedicado especialmente a aquellos desarrolladores que ya han hecho componentes COM con Visual Basic 6.0, ya que el IDE de Visual Basic 6.0 sí nos avisa cuando rompemos la compatibilidad binaria. De todas formas, ese aviso lo tendremos cuando intentemos usar la aplicación cliente que se encargará de mostrarnos un aviso "clásico" de que "El componente ActiveX no puede crear el objeto" tal como vemos en la figura 1, o de que el objeto no acepta esa propiedad o método, tal como se muestra en la figura 2.

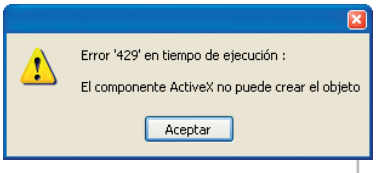

Figura 1. Error al intentar usar un componente que no está registrado (o no existe)

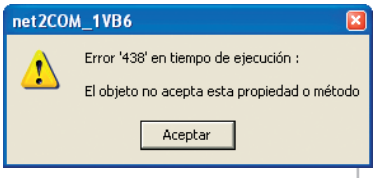

Figura 2. Error al acceder a un método no existente

El primer error (figura 1) lo recibiremos si el componente no está registrado, mientras que el segundo (figura 2) lo recibiremos al acceder a un método que ya no está en el componente.

Crear componentes compatibles COM de forma "rápida" y no recomendable

Como hemos comentado, empezaremos viendo lo que no tenemos que hacer, o al menos lo que "el autor" no recomienda que hagamos. ¿Por qué? Porque el Visual Studio crea una librería de compatibilidad COM (librería de tipos) que solo deberíamos usar para hacer pruebas rápidas, pero nada más. Esa librería de tipos solo expone los objetos que podemos usar, es decir los nombres de las clases, pero no expone (realmente si los expone, pero como si no estuvieran), los métodos, propiedades y eventos que definamos. Por tanto, al usar esos métodos "tenemos que saber que están", ya que el *Intellisense* de Visual Basic 6.0 no nos mostrará esos métodos, ni tampoco lo hará el examinador de objetos, tal como podemos ver en la figura 3.

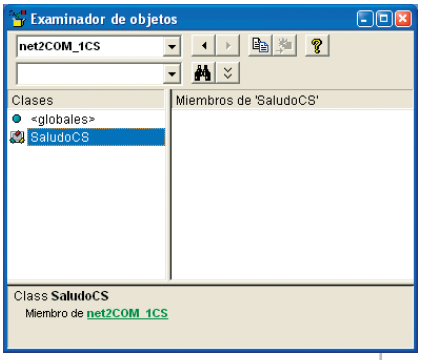

Figura 3. El examinador de objetos de VB6 no muestra los miembros del componente

Para crear este tipo de componente desde Visual Studio (cualquier versión), simplemente haremos lo siguiente:

1. Creamos un nuevo proyecto de tipo *Class Library* (biblioteca de clases).
2. Definimos un método público en la clase que se añade.
3. En las propiedades del proyecto marcamos la opción "Registrar para interoperabilidad COM".
4. En la información de "Ensamblado" debemos marcar "Crear ensamblado visible a través de COM" o en `AssemblyInfo` usamos el atributo `ComVisible(true)`.
5. Generaramos el proyecto y se creará tanto el ensamblado de .NET como la librería de tipos (`.tlb`).
6. Creamos una aplicación EXE estándar en VB6.
7. En "Referencias" añadimos una referencia al componente COM que acabamos de crear con .NET.
8. Utilizamos la clase y el método que hemos expuesto desde .NET.
9. Lo ejecutamos y todo funcionará como esperábamos.

El código del componente de .NET puede ser tan simple como el mostrado en el fuente 1.

```
namespace net2COMCS
{
    public class SaludoCS
    {
        public string Saludar()
        {
            return "Hola desde .NET (C#)";
        }
    }
}
```

Fuente 1. Clase de C# para usar como componente COM

En la aplicación cliente de Visual Basic 6.0 lo usaremos tal como mostramos en el fuente 2.

```
Private Sub cmdSaludoCS_Click()
    Dim oCS As net2COM_1CS.SaludoCS
    Set oCS = New net2COM_1CS.SaludoCS

    Me.Text1.Text = oCS.Saludar

End Sub
```

Fuente 2. Código de VB6 para usar la clase del fuente 1.

La única pega es que al intentar indicar el método al que queremos acceder, no nos mostrará los métodos disponibles, por tanto debemos "saber" que la clase `SaludoCS` expone

> **NOTA**
>
> Cuando marcamos la opción "Registrar para interoperabilidad COM" que en Visual C# 2005 está en la ficha "Generar" de las propiedades del proyecto, mientras que en Visual Basic 2005 esa opción está en la ficha "Compilar", y al marcarla se marca también la casilla "Crear ensamblado visible a través de COM" y se añade el atributo `ComVisible(true)` en el fichero `AssemblyInfo`, sin embargo en C# tendremos que indicarlo manualmente. Por tanto debemos asegurarnos de que está activada esa opción, ya que solamente al tener las dos opciones marcadas es cuando se genera la librería de tipos que se usará desde el cliente COM.

un método llamado `Saludar`, porque, como ya hemos mencionado, el *Intellisense* de VB6 no sabe qué métodos expone la clase. Esto es lo que se llama *late-binding* o lo que es lo mismo "rezar y esperar que ese método exista", ya que será en tiempo de ejecución cuando se compruebe si el método existe o no, que el método existe, entonces todo irá bien, que no existe y no interceptamos el posible error, obtendremos una pantallita de error como la mostrada en la figura 2.

Crear un componente .NET que exponga los métodos a COM (en tiempo de compilación)

Hacer que el componente de .NET expuesto al mundo de COM muestre los métodos que la clase expone es fácil de conseguir, ya que solo implica añadir un atributo a la clase o al ensamblado completo:

```
ClassInterface(
      ClassInterfaceType.AutoDual)
```

Con este atributo, el cliente COM podrá "ver" los métodos expuestos desde .NET, es decir, tanto el examinador de objetos de VB6 como el *Intellisense* mostrarán los métodos públicos de nuestra clase, los cuales incluyen los que nosotros definamos además de los heredados desde la clase `Object`. En la figura 4 podemos ver una captura del examinador de objetos de VB6.

Si después de añadir el atributo indicado volvemos a compilar el componente, veremos que desde VB6 ya podemos ver los miembros expuestos por nuestra clase creada desde .NET,

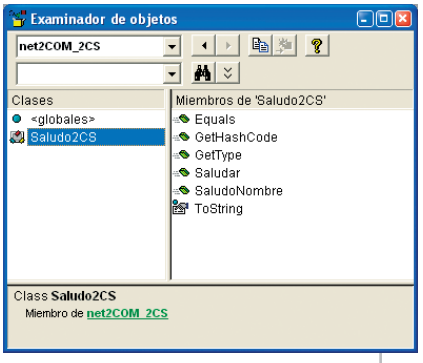

Figura 4. El examinador de objetos de VB6 ahora muestra los miembros del componente

y que *Intellisense* mostrará esos miembros, en este caso estaremos usando lo que se conoce como *early-binding* o "lo que ves es lo que hay", y por tanto en tiempo de compilación se comprueba que el método al que queremos acceder existe en esa clase, de no ser así, la aplicación no se compilará, mostrando un error como el que podemos ver en la captura de la figura 5, con lo que nos aseguramos de que no "venderemos" algo que no funciona.

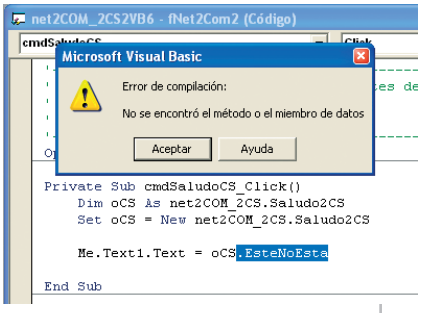

Figura 5. Error en tiempo de compilación al no encontrar el método que queremos usar

Romper la compatibilidad binaria es fácil

Veamos que fácil es romper la compatibilidad binaria, y cómo podemos "recuperarnos" de ese error.

Una vez que tenemos creado nuestro componente de .NET y la librería de tipos para que lo podamos usar desde un cliente de COM, no podemos hacer modificaciones a la clase que hemos expuesto desde el componente. Por ejemplo, no podemos cambiar el nombre del método (o métodos) ni añadir o quitar ninguno de los existentes.

Lo que si podemos hacer es añadir nuevas clases. Esas nuevas clases no afectarán a los clientes existentes y los nuevos sí podrán usarlas, ya que en realidad lo que hace COM es crear una nueva interfaz para esas clases y seguir usando la que ya había, por tanto, podemos mantener la compatibilidad binaria con los clientes anteriores, la única pega es que para poder hacerlo debemos crear nuevas clases, no usar las existentes, pero como veremos en la siguiente parte, este "inconveniente" lo podemos solventar fácilmente.

> **NOTA**
>
> Cada vez que compilemos el componente desde Visual Studio, debemos tener cerrado el IDE de Visual Basic 6.0, ya que si lo dejamos abierto, no se podrá generar porque "otra aplicación lo está usando".

Lo que sí podremos hacer en los nuevos clientes COM que usen el componente modificado es usar tanto las clases antiguas como las nuevas, pero si esas nuevas clases definen eventos, esos eventos no los podremos usar desde Visual Basic 6.0.

El código usado para demostrar cómo romper la compatibilidad o crear nuevas clases para mantener esa compatibilidad binaria es el que podemos ver en el fuente 3, en el que está comentado el código que rompería la compatibilidad binaria.

```
namespace net2COM2CS
{
    public class Saludo2CS
    {
        public string Saludar()
        {
            return "Hola desde .NET (C#) v2";
        }

        // Si añadimos nuevos métodos a la clase expuesta,
        // romperemos la compatibilidad binaria,
        // con lo que tendremos que volver a compilar el cliente COM
        //public string SaludoNombre(string nombre)
        //{
        //    return "Hola " + nombre + " desde .NET (C#) v2";
        //}
    }

    public class OtraClase
    {
        // Los eventos no se muestran en el cliente COM
        public delegate void UnEventoEventHandler(string msg);
        public event UnEventoEventHandler UnEvento;

        public string SaludoNombre(string nombre)
        {
            // Producimos el evento
            if(UnEvento != null)
            {
                UnEvento("Se usa el método SaludoNombre");
            }

            return "Hola " + nombre + " desde .NET (C#) v2.1";
        }
    }
}
```

Fuente 3. El código de C# para crear nuevas clases y métodos en el componente

Y el código que podríamos usar desde VB6 es el que vemos en el fuente 4.

```
Private Sub cmdSaludoCS_Click()
    ' El código anterior no necesita cambios
    Dim oCS As net2COM_2CS.Saludo2CS
    Set oCS = New net2COM_2CS.Saludo2CS

    Dim otra As net2COM_2CS.OtraClase
    Set otra = New net2COM_2CS.OtraClase

    Me.Text1.Text = oCS.Saludar

    MsgBox otra.SaludoNombre("Guille")

End Sub
```

Fuente 4. El código del cliente COM (VB6)

Crear componentes de .NET para COM de forma recomendada

Veamos ahora qué es lo que debemos hacer para que nuestro código de .NET genere librerías de tipos que mantengan la compatibilidad binaria.

Como hemos visto, todo el problema que nos surge para poder mantener la compatibilidad binaria del componente COM o, mejor dicho, de la librería de tipos, es porque cambiamos las interfaces que exponemos. Bueno, nosotros no cambiamos ninguna interfaz, es el propio Visual Studio el que se encarga de crear esas interfaces o dicho de otra forma: es el Visual Studio el que crea las interfaces que se exponen a COM.

La forma que tenemos nosotros de "controlar" lo que se expondrá al cliente COM es creando las interfaces que se expondrán, es decir, no dejando de la mano de Visual Studio la creación "automática" de las interfaces, o al menos interviniendo un poco en las interfaces que se deben exponer a COM.

Cuando usamos el atributo `ClassInterface(ClassInterfaceType.AutoDual)`, la utilidad `regasm` (con el parámetro `/tlb`) genera automáticamente una interfaz para nuestro componente; esa interfaz está basada en la clase, es decir, incluye TODOS los miembros que la clase defina, por tanto lo que debemos hacer es indicarle que NO genere una interfaz predeterminada, esto lo conseguimos usando ese atributo indicándole como argumento de llamada el valor `ClassInterfaceType.None` de esta forma no generará una interfaz, por tanto debemos indicar nosotros la interfaz que queremos que se exponga a COM. Esto es tan fácil como definir una interfaz con los miembros a exportar e implementar esa interfaz en nuestra clase.

En el fuente 5 tenemos la versión modificada del ejemplo mostrado en el fuente 1 para que use una interfaz de forma explícita.

```
using System;
using System.Runtime.InteropServices;

// Aplicando el atributo a nivel de ensamblado
// no tenemos que aplicarlo en cada clase.
[assembly: ClassInterface(ClassInterfaceType.None)]

namespace net2COM3CS
{
    public interface ISaludo3CS
    {
        string Saludar();
    }

    public class Saludo3CS : ISaludo3CS
    {
        public string Saludar()
        {
            return "Hola desde .NET (C#) v3";
        }
    }
}
```

Fuente 5. La clase se expondrá con la interfaz que definimos

Compilamos el ensamblado, (sin olvidar de exponerlo a COM), y podemos crear una versión de VB6 que

lo use. El código será similar al del fuente 2, salvo que ahora la clase (y el espacio de nombres) se llama de otra forma.

Añadir nuevos métodos y mantener la compatibilidad binaria

Ahora viene la parte interesante: Añadir nuevos métodos a la clase y mantener la compatibilidad binaria con los clientes anteriores.

Como hemos comentado, COM se basa en las interfaces expuestas por los componentes, por tanto habrá compatibilidad entre componentes siempre que no "rompamos" el contrato que hemos establecido, y como ya sabemos (ver nº 16 y 18 de **dotNetManía**) esos contratos los hacemos por medio de las interfaces, por tanto, para mantener la compatibilidad con el componente que ya hemos creado, y usado desde un cliente COM, debemos mantener la interfaz que el cliente COM espera encontrar, y la nueva funcionalidad la debemos proporcionar por medio de otra interfaz.

Esa nueva interfaz debe exponer los miembros anteriores además de los nuevos, en nuestro ejemplo, vamos a añadir un nuevo método a la clase, por tanto definiremos una nueva interfaz que contenga los dos métodos, tal como vemos en el fuente 6.

```
public interface ISaludo3CS_2
{
    string Saludar();
    string SaludoNombre(string nombre);
}
```

Fuente 6. La nueva interfaz que vamos a exponer en nuestro componente .NET

En la misma clase que tenemos definiremos el nuevo método, pero también indicaremos que ese método es el que define la interfaz, por tanto también debemos implementarla, sin olvidar de que la otra interfaz debe seguir siendo expuesta. Aquí debemos usar un pequeño truco, y ese truco consiste en definir primero la nueva interfaz, y la antigua la definimos a continuación, tal como vemos en el fuente 7.

El hecho de definir la nueva interfaz primero es para que los nuevos clientes usen esa interfaz de forma predeterminada, de forma que puedan acceder a los miembros expuestos por la misma, los clientes antiguos verán dos interfaces, pero como no hacemos uso de la nueva, no habrá ningún tipo de problema. De hecho si miramos los elementos expuestos por el componente en el examinador de objetos de Visual Basic 6.0, veremos que además de la clase se muestra también la interfaz antigua (ver figura 6).

```
// La interfaz más reciente debemos indicarla antes de la antigua
public class Saludo3CS : ISaludo3CS_2, ISaludo3CS
{
    public string Saludar()
    {
        return "Hola desde .NET (C#) v3 r1";
    }

    public string SaludoNombre(string nombre)
    {
        return "Hola " + nombre + " desde .NET (C#) v3.1";
    }
}
```

Fuente 7. La clase debe implementar las dos interfaces

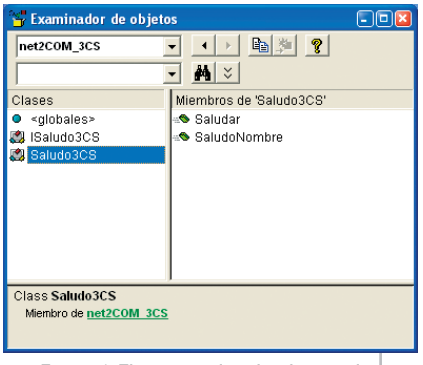

Figura 6. El examinador de objetos de VB6 solo muestra los métodos definidos en la interfaz

Pero también veremos que solo se exponen los miembros definidos en la interfaz, si lo comparamos con la figura 4, comprobaremos que NO se incluyen los miembros heredados de la clase `Object`, si queremos que esos métodos estén expuestos (y disponibles desde COM), debemos incluirlos en la interfaz, tal como vemos en el código fuente 8.

```
public interface ISaludo3CS_2
{
    string Saludar();
    string SaludoNombre(string nombre);
    string ToString();
}
```

Fuente 8. La interfaz debe exponer todos los miembros que queremos usar desde COM

Ni qué decir tiene que si hacemos ese cambio después de haber compilado el componente y el cliente COM, romperemos la compatibilidad binaria, por tanto ese nuevo método debemos incluirlo en una nueva interfaz (en el ZIP con el código de ejemplo se utiliza este nuevo cambio desde VB6 como el proyecto `net2COM_3_2VB6`).

Utilizar eventos de .NET desde un cliente COM

Los eventos definidos en los componentes de .NET es otro de los temas que debemos considerar de forma especial para poder exponerlos a COM. Si definimos un evento de la forma habitual en nuestro componente de .NET, veremos que en el cliente COM solo se muestra la definición del delegado, pero no como un evento. De hecho, si queremos usarlo desde Visual Basic 6.0, comprobaremos que no nos deja.

En VB6, para definir un objeto que contiene eventos debemos declararlo con la instrucción `WithEvents`, y al usar esta instrucción, el IDE de VB6 comprobará si la clase que declaramos expone o no eventos, si no los expone, obtendremos un aviso de error como el mostrado en la figura 7.

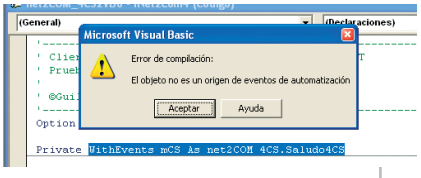

Figura 7. Visual Basic 6.0 nos avisa cuando no hay eventos en una clase

El aviso en realidad se producirá en tiempo de ejecución, pero en tiempo de diseño, al "intentar" indicar el tipo, simplemente no nos mostrará nuestra clase, porque desde el punto de vista de COM esa clase no produce eventos.

Podemos intentarlo usando el atributo `ClassInterface` pasándole `Class`

`InterfaceType.AutoDual` como argumento, que como sabemos creará una interfaz de forma automática, con lo que mandaremos nuevamente al "garete" todo lo que hemos visto de "buenas prácticas". Incluso desde el cliente COM ese evento no se mostrará como tal, sino como dos métodos, tal como podemos ver en la figura 8.

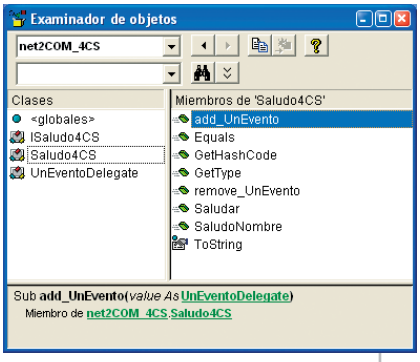

Figura 8. Los eventos de .NET se muestran en COM como dos métodos

La solución es crear una interfaz que defina los eventos que queremos exponer a COM, pero no solo nos basta definir esa interfaz, sino que también debemos decirle a COM que lo que esa interfaz define son eventos, además de que la clase también debe "firmar" el contrato, pero en esta ocasión no lo hace implementando la interfaz, ya que en realidad eso no es necesario, (al menos desde el punto de vista de .NET), sino por medio de un atributo que instruya a `regasm` que esa clase utilizará los eventos definidos en esa interfaz. ¿Un lío? Seguramente. Pero como veremos tiene su lógica, al menos desde el punto de vista de COM que todo lo maneja por medio de interfaces.

NOTA

En Visual Basic 2005 podemos definir eventos sin necesidad de asociarlo a un delegado, pero .NET creará ese delegado por nosotros y por tanto lo expondrá a COM, por tanto, si no queremos que se muestre el delegado definido automáticamente por el compilador de Visual Basic para .NET, deberíamos definir los eventos de la forma "recomendada", es decir, usando delegados. En el código de ejemplo incluido en el ZIP los eventos de Visual Basic están definidos usando delegados.

Antes de ver el código que utilizaría esos eventos, veamos cómo podemos evitar que se exponga a COM la definición del delegado, ya que en realidad no nos hace falta, al menos desde el cliente COM. Esto mismo también será aplicable a todos los métodos públicos de nuestra clase que no queramos exponer a COM, (solo en el caso de que usemos el atributo `ClassInterface` con el valor `AutoDual`, ya que si usamos `None`, solo se expondrá a COM lo que definamos en las interfaces). A lo que vamos, para que el delegado no se muestre en el examinador de objetos del cliente COM debemos aplicarle el atributo `ComVisible(false)`:

```
[ComVisible(false)]
public delegate void
      UnEventoEventHandler(string msg);
```

De esta forma le estamos indicando a la utilidad `regasm` que no exponga la clase o método al que aplicamos ese atributo. Es importante que este atributo lo apliquemos de forma "local" a cada uno de los elementos que NO queramos exponer a COM, pero el atributo definido en `AssemblyInfo` debe tener un parámetro verdadero, sino, no se creará una librería de tipos de nuestro ensamblado.

...lo expuesto en este artículo es más que suficiente para que podamos crear componentes con .NET para que puedan ser utilizados desde clientes COM

Ahora pongamos junto todo lo indicado para que nuestro componente de .NET exponga eventos a un cliente COM.

En el fuente 9 vemos el código de C# y en el fuente 10 tenemos el código de Visual Basic 6.0 que utiliza esos eventos.

Si desde Visual Basic 6.0 mostramos el examinador de objetos, (ver figura 9), comprobaremos que en realidad no hay ninguna interfaz, y que el evento se muestra como parte de la clase.

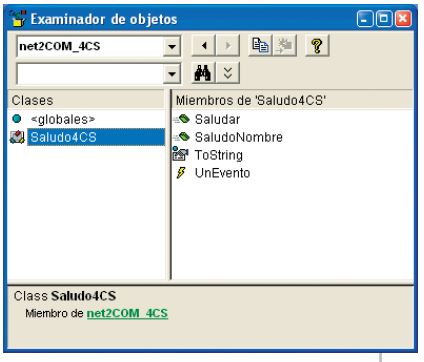

Figura 9. El examinador de objetos de VB6 muestra el evento como parte de la clase

Añadir más eventos a la clase y mantener la compatibilidad binaria

Para ir terminando, veamos cómo añadir nuevos eventos a un componente ya existente, y, por supuesto, sin romper la compatibilidad binaria.

En este caso haremos como antes, es decir, definir una nueva interfaz para incluir los eventos que queremos exponer. Esa interfaz incluirá tanto los nuevos eventos como los antiguos que queramos que las nuevas versiones sigan

```
using System;
using System.Runtime.InteropServices;

// Aplicando el atributo a nivel de ensamblado
// no tenemos que aplicarlo en cada clase.
[assembly: ClassInterface(ClassInterfaceType.None)]

namespace net2COM4CS
{
    // El delegado lo podemos ocultar a COM
    [ComVisible(false)]
    public delegate void UnEventoEventHandler (string msg);

    // Los eventos no se muestran en el cliente COM
    // debemos indicarlos expresamente en una interfaz.

    // Este atributo solo para exponer eventos
    [InterfaceTypeAttribute(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ISaludo4Eventos
    {
        void UnEvento(string msg);
    }

    public interface ISaludo4CS
    {
        string Saludar();
        string SaludoNombre(string nombre);
        string ToString();
    }

    // Debemos indicarle la interfaz en la que está definido del evento
    [ComSourceInterfaces(typeof(net2COM4CS.ISaludo4Eventos))]
    public class Saludo4CS : ISaludo4CS
    {
        public event UnEventoEventHandler UnEvento;

        protected void onUnEvento(string msg)
        {
            // Producimos el evento
            if(UnEvento != null)
            {
                UnEvento(msg);
            }
        }

        public string Saludar()
        {
            onUnEvento("Evento desde el método Saludar");

            return "Hola desde .NET (C#) v4";
        }

        public string SaludoNombre(string nombre)
        {
            onUnEvento("Evento desde el método SaludoNombre");

            return "Hola " + nombre + " desde .NET (C#) v4.1";
        }

        public override string ToString()
        {
            onUnEvento("Evento desde el método ToString");

            return "ToString de la clase Saludo4CS";
        }
    }
}
```

Fuente 9. El código de C# que define una clase que produce eventos

```
Private WithEvents mCS As net2COM_4CS.Saludo4CS

Private Sub cmdSaludoCS_Click()

  List1.Clear

  Text1.Text = mCS.Saludar

  Text2.Text = mCS.SaludoNombre(txtNombre.Text)

  Text3.Text = mCS.ToString

End Sub

Private Sub Form_Load()
  Set mCS = New net2COM_4CS.Saludo4CS

End Sub

Private Sub mCS_UnEvento(ByVal msg As String)
  List1.AddItem msg
End Sub
```

Fuente 10. El código de VB6 que utiliza un componente de .NET que produce eventos

soportando, es decir, la nueva interfaz incluirá los eventos que nos interese que los "nuevos" clientes COM usen. Esto mismo es aplicable a los métodos que queramos exponer a COM desde nuestro componente, ya que todo lo que esté en la nueva interfaz será lo que utilicen los "nuevos" clientes COM, lo de nuevos está resaltado para que quede claro que los clientes anteriores seguirán usando lo que el componente exponía cuando se crearon las aplicaciones clientes.

Para indicar que ahora en vez de una interfaz de eventos hay más, seguiremos aplicando el atributo **ComSourceInterfaces** a la clase, pero en vez de indicar una interfaz de eventos, indicaremos las que hayamos definido, poniendo como primer argumento la última que hemos definido, en nuestro ejemplo sería de esta forma:

```
[ComSourceInterfaces(
 typeof(net2COM4CS.ISaludo4_2Eventos),
 typeof(net2COM4CS.ISaludo4Eventos))]
public class Saludo4CS : ISaludo4CS {
```

Es decir, separamos cada interfaz (usando **typeof**) con una coma. De esta forma podemos indicar un máximo de cuatro interfaces de eventos. Si queremos indicar más de cuatro, debemos usar el constructor que recibe una cadena como parámetro. Esa cadena permite una o más interfaces, en caso de que haya más de una,

separaremos cada una de ellas con un valor nulo, por ejemplo:

```
[ComSourceInterfaces(
 "net2COM4CS.ISaludo4_2Eventos\
      0net2COM4CS.ISaludo4Eventos")]
```

Consideraciones finales

Creo que lo expuesto en este artículo es más que suficiente para que podamos crear componentes con .NET para que puedan ser utilizados desde clientes COM, como es el caso de Visual Basic 6.0, y usarlos de la forma correcta, sin que el crecimiento de nuestro componente interfiera con los clientes que ya están usando las versiones anteriores. Ni qué decir tiene que todo lo aquí explicado es para los casos en que necesitemos ampliar la funcionalidad del componente y permitir que los clientes que ya están usando los componentes anteriores sigan funcionando a las mil maravillas. Pero debemos tener en cuenta que hay ciertas características de .NET que COM no sabe manejar, como es el caso de la sobrecarga, ya sea de métodos, constructores u operadores. Por tanto debemos saber lo siguiente:

- Las clases de los componentes de .NET siempre deberían inlcuir un constructor público sin parámetros. Si no lo definimos, no podremos crear nuevos objetos de esa clase, y si esa clase es la única expuesta por el componente, no se generará la librería de tipos.
- Cuando existen sobrecargas de métodos, en COM se utilizan nombres diferentes para cada una de las sobrecargas. El orden en el que se declaren esas sobrecargas en la interfaz indicará el nombre que se usará en COM. El primero no cambiará, pero cada sobrecarga se nombrará con un guión bajo seguido de un número, empezando con 2. Por ejemplo: `Mostrar`, `Mostrar_2`, `Mostrar_3`, etc. También podemos definir en la interfaz el nombre que queremos que se exponga a COM y lo podemos definir en la clase de .NET como queramos, sin embargo en VB es más fácil crear este "engaño" ya que las implementaciones de los miembros de una interfaz se hace de forma explícita. En los proyectos numerados como **`prueba 5`** que se incluyen en el ZIP del código hay ejemplos para ver cómo solventar estos casos.
- Si indicamos arrays como parámetros, éstos deben estar declarados por referencia (`ref` en C#, `ByRef` en VB).

Cuando creemos nuevas versiones de los componentes de .NET es muy importante que NO cambiemos la versión del ensamblado, ya que si lo hacemos no se mantendrá la compatibilidad binaria. Si debemos hacerlo, lo recomendable es mantener la compatibilidad con la versión del componente COM mediante el atributo: `ComCompatibleVersion` al que le indicaremos en el constructor la versión del ensamblado original, por ejemplo:

```
[assembly: ComCompatibleVersion(1, 0, 0, 0)]
```

> La ventaja de crear componentes COM desde .NET es que en el código de las clases que expongamos podemos usar todos los tipos soportados por .NET, incluso los tipos y colecciones genéricas

También es importante indicar la descripción del ensamblado de .NET, ya que esa descripción será la que se use a la hora de añadir la referencia en la aplicación del cliente COM, tal como podemos ver en la figura 10.

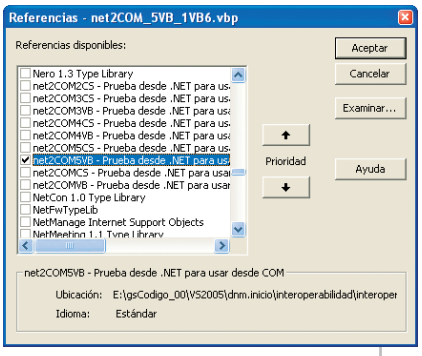

Figura 10. Lo que indiquemos en la descripción del ensamblado es lo que se mostrará en las referencias

Conclusiones

La ventaja de crear componentes COM desde .NET es que en el código de las clases que expongamos podemos usar todos los tipos soportados por .NET, incluso los tipos y colecciones génericas, si bien, como es lógico pensar, los tipos de datos expuestos públicamente deben ser tipos que COM conozca, aunque esto no implica que internamente no nos aprovechemos de las ventajas de los tipos que el propio .NET define.

El código incluido en el ZIP contiene proyectos tanto para Visual Basic 2005 como C# 2005, aunque todo lo explicado se puede usar también en las versiones anteriores de Visual Studio, también se incluye el código de ejemplo de las diferentes versiones de Visual Basic 6.0 y la explicación correspondiente para poder generar paso a paso las distintas versiones de los componentes expuestos a COM.

Confío en que todo lo explicado sea de utilidad para poder actualizar esos proyectos que se resisten a dejar el mundo COM para convertirlos definitivamente al mundo .NET.