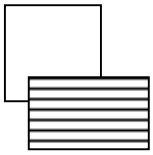


.NET

En estos pocos párrafos vamos a describir los puntos claves del nuevo sistema de Microsoft y lo haremos a partir de ejemplos y con pocas definiciones. Toda la información extra que se necesite se puede encontrar en las MSDN.

¿Cuál es el problema? ¿Por qué cambiar el modelo de trabajo? ¿En qué pensaron para cambiar? Para poder responder a todas estas preguntas vamos a hacer un poco de historia del modelo que estábamos usando actualmente. La mayoría de programadores de plataformas Microsoft sabe lo que es COM, pero para aquellos que no sepan de que se trata vamos a explicarlo. A medida que el software iba creciendo y las aplicaciones también lo hacían en tamaño, siempre se fue buscando la forma de reaprovechar código al máximo. De aquí que un buen diseño en la programación orientada a objetos permite que un programador reaproveche código. Aunque es cierto que no todos los lenguajes actuales tiene OOP y no siempre será posible ocultar el código al cliente para evitarle ciertos males. Debido a esto, en 1993 Microsoft lanza una nueva tecnología: COM (Component Object Model) que nos permite compartir código de un servidor a un cliente, a nivel binario sin compilar las fuentes y reaprovechando código de una forma fácil y cómoda. Por ejemplo, la típica barra de progreso que aparece en todas las aplicaciones convencionales es un objeto COM encapsulado en un ensamblado habitualmente conocido como DLL (Dynamic Link Library). De saber dónde está y cómo se accede a él ya se encarga el sistema operativo mientras que el programador que utiliza el componente sólo necesita declararlo, instanciarlo y usarlo. Del resto se encarga el sistema operativo.

Con esta nueva técnica parecía que la programación daba un gran salto, pero nuevamente las previsiones se hicieron cortas. Los problemas principales de un objeto COM son: primero necesitamos de un fuerte “infraestructura” (generadores de clases, cálculos de referencias a interfaces, etc.) que nos permita cooperar y no siempre es compatible con todas las versiones, todos los clientes, todos los servidores y sin pensar en los distintos compiladores en que se pueden haber hecho cada uno de los componentes de una aplicación. Y segundo los objetos no comparten su implementación interna lo que nos deja “ciegos” de cómo hemos de hacer las cosas. Para resumir, lo que estamos queriendo decir es que la fuerte diferencia entre los componentes nos obliga a introducir algún tipo de mecanismo entre ambos para permitir que se entiendan (por ejemplo un cliente en VB que se quiere comunicar con un servidor en VC++ a través de un string, las implementaciones de los strings en los dos compiladores son distintas). Con estas limitaciones COM estaba estancado y se necesitaba un cambio que nos permitiera romper con estas barreras.

Para ello se reunieron todas las dificultades: que el código sea capaz de ejecutarse en distintas plataformas, que exista una administración automática de la memoria para evitar pérdidas, una ayuda para administrar versiones diferentes del mismo paquete de software, poder usar todas las características de OOP en todos los lenguajes (y no meras aproximaciones...), limitar el acceso de ciertos bloques de código al sistema operativo, disponer una forma más lógica de

organizar las funciones del sistema y por último mantener la compatibilidad con las versiones anteriores de COM.

Entonces, en el momento que se conoce *lo que se tiene hasta el momento*, *el problema* que esto trae implícito y qué *es necesario* solucionar, es cuando Microsoft lanza .NET Framework; un producto de sistema operativo que proporciona soluciones a los problemas que acabamos de mencionar.

Vamos a empezar por explicar cada una de las mejoras que aporta .NET a la comunidad de desarrolladores en plataformas Windows. La clave de todo el producto es el código administrado, que se ejecuta en un entorno llamado CLR (Common Language Runtime) que es otra capa por encima de la API win32 que proporciona un conjunto de mejoras y un conjunto de servicios más rico que el sistema operativo solo.

¿Cómo puede CLR trabajar con cualquier lenguaje? Para responder a esta pregunta antes hemos de especificar el lenguaje. Toda herramienta de desarrollo que quiera trabajar con .NET necesita compilar su lenguaje (VB, VC++, JAVA, Delphi, Fortran, Cobol, etc...) a un lenguaje intermedio llamado MSIL (Microsoft Intermediate Language, abreviado como IL). Todo lenguaje que genere este código intermedio puede acceder a las mejoras del CLR, sin importar con qué lenguaje se ha escrito. El código generado por VB.NET tiene la misma potencia que VC++ o C# u otros lenguajes ya que ahora la parte final de la compilación se hará en el CLR.

Antes hemos puesto el ejemplo de la implementación de las cadenas String en distintos lenguajes, en .NET esto no importa porque todos usan la misma implementación de un String, ya que el objeto String pertenece a .NET y no al lenguaje. Lo mismo para arrays, enteros, objetos, constructores, etc... En resumen CLR provee de un conjunto de clases que los lenguajes utilizan.

¿Qué sucede cuando se ejecuta una aplicación bajo CLR? Hasta aquí tenemos que cada uno de los lenguajes que trabajan con .NET generan un código intermedio IL. Cuando el CLR detecta que se ejecuta dicho componente o ejecutable lo divide en dos fases: primero si es la primera vez que se ejecuta y segundo si ya se ha ejecutado previamente. Si es la primera vez que se ejecuta el programa, el CLR inicia el compilador "Just in time" (JIT) encargado de generar el código maquina a partir de IL, para la plataforma en la que estamos. Lo que nos permite mejorar el rendimiento de las aplicaciones a nivel de hardware sin tener que preocuparnos. Hasta aquí tenemos que el código .NET es independiente de la plataforma. No es que Microsoft quiera meter .NET en entornos LINUX o UNIX sinó que es una mejora orientada a cambios del hardware de 32 a 64 bits, aunque no nos deberá de extrañar si vemos implementaciones del CLR en otros sistemas operativos de terceros fabricantes. Cabe añadir que tendremos una segunda división si el código ya ha sido compilado y ejecutado con anterioridad, entonces CLR creará un caché que nos permitirá agilizar el código en las siguientes llamadas sin tener que compilar de nuevo.

Todo cambio o decisión lleva ligado unas ciertas mejoras y unos ciertos problemas, pagar por la independencia de la plataforma hace que las aplicaciones sean más costosas en tiempo y ciclos de CPU la primera vez que se

ejecutan, pero las siguientes veces los mecanismo del CLR intentan equiparar las prestaciones del CLR con las actuales.

¿Qué otras ventajas trae consigo .NET?

La primera y más destacable es que dispone de un gestor de memoria: *garbage collector (GC)* que se encarga de todas las pesadas y tediosas tareas del programador para asegurar que el programa no pierde memoria después de varias ejecuciones.
La segunda es que .NET trae consigo un sistema automatizado para controlar las versiones del software.
La tercera, extiende las características de la OOP a todos los lenguajes.
La cuarta, organiza la información o las clases en espacios de nombres de sistema, lo que permite que funciones que se llaman igual no se interfieren entre ellas, por ejemplo el control de los streams, las funciones de error, la apertura de canales, etc... no entran en conflicto aunque se llamen de la misma forma, pues están en distintos espacios de nombres.
La quinta, .NET admite protección y seguridad de código. Por ejemplo a un dll que no conocemos que hace podemos vetarle ciertos accesos al sistema operativo u otros ficheros.
Sexta, proporciona unas clases que permiten trabajar con COM de forma transparente (cliente y servidor).

¿Cuál es el precio?¿Qué perdemos con todo esto? Hasta aquí sabemos qué cosas buenas trae consigo .NET, pero tal como hemos dicho antes ¿quién asume estos costes? La principal dificultad está en el sistema operativo el cual se hace más complejo con todo el código administrado(el CLR y el recolector de basura (GC)) y en segundo caso las pérdidas de ciclos de CPU . Resaltaremos que este último inconveniente es sólo una mera cuestión de tiempo hasta que cpu's más modernas lleguen a los ordenadores .

Llegados a este punto es momento de meternos manos a la obra, y empezar a tocar esta nueva herramienta de desarrollo. Es importante saber que es puede descargar de forma gratuita una copia del SDK .NET desde la pagina oficial de Microsoft en la sección de descargas. Supondremos que hemos hecho la instalación en la carpeta por defecto: <raíz>\WINNT\Microsoft.NET\Framework\vX.Y.ZZZZ. En esta carpeta encontraremos los compiladores de VB, C#, ASP y Javascript entre otros.

Primer contacto

Para iniciarnos en .NET lo haremos con un simple ejemplo, una parte escrita en VB.NET (el servidor) y el otra en C# (el cliente). En este ejemplo mezclamos dos lenguajes para mostrar que es muy fácil crear componentes en varios lenguajes. A partir de ahora todos los ejemplos serán en VB.NET, aunque como veremos en .NET escoger el lenguaje es una cuestión de gusto, pues el código que se ejecuta es el mismo. Por ahora no nos preocuparemos por la jerga del lenguaje, nos limitaremos a picar este código con nuestro editor favorito, por ejemplo con el notepad.

1.1 Ejemplo de servidor (server.vb):

```
Imports Microsoft.VisualBasic

Namespace TimeComponentNS

   Public Class TimeComponent

      Public Function GetTime(ByVal ShowSeconds As Boolean) As String

        If (ShowSeconds = True) Then
             Return Now.ToLongTimeString
        Else
             Return Now.ToShortTimeString
        End If

    End Function

   End Class
```

1.2 Ejemplo de cliente (cliente.cs):

```
using System ;
using TimeComponentNS ;

class MainApp
{
        public static void Main()
        {
                TimeComponent tc = new TimeComponent ( ) ;
                Console.Write (tc.GetTime (true)) ;
        }
}
```

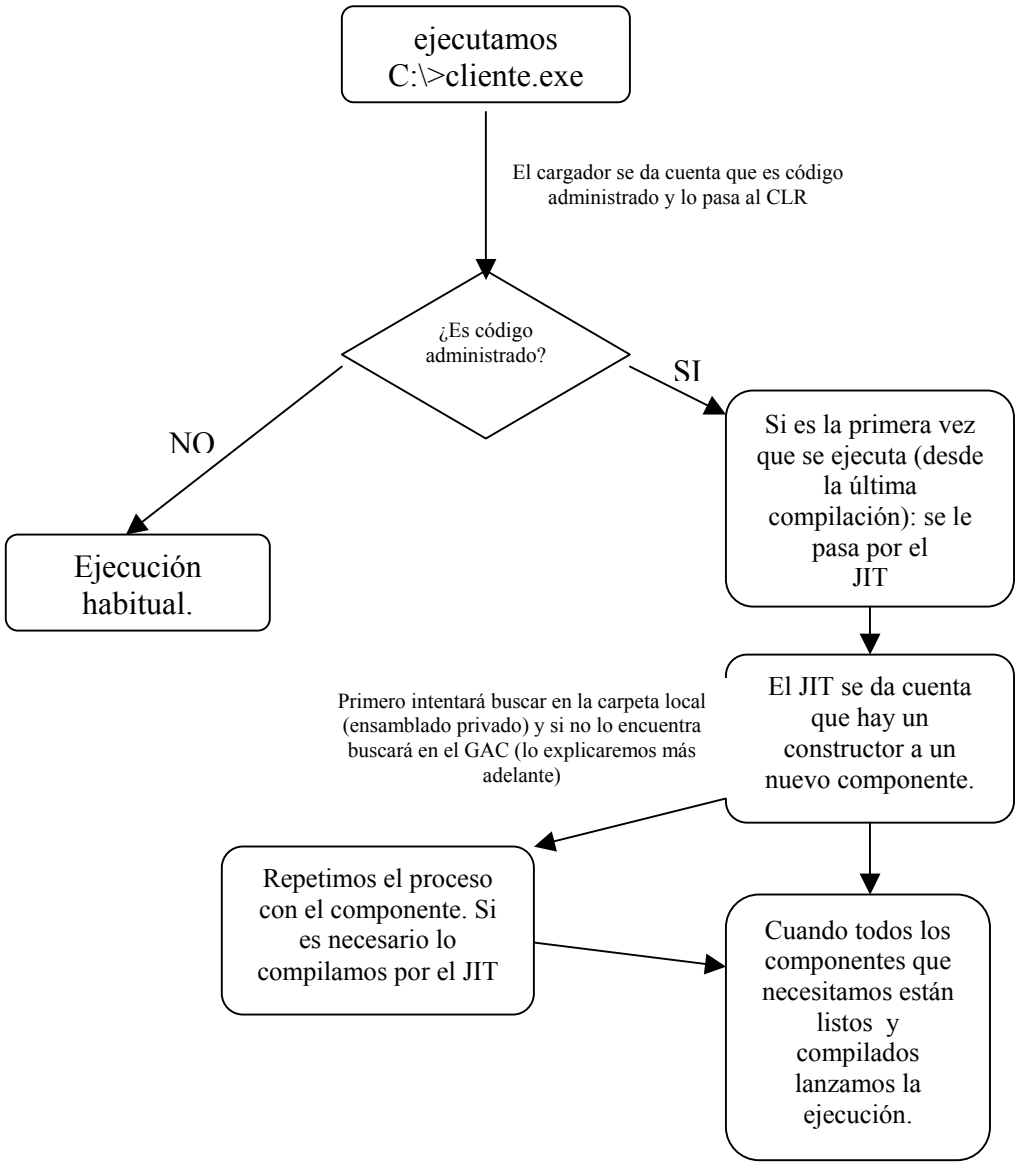
Guardaremos ambos ficheros en el mismo directorio y procederemos a compilarlos.

1.3 Compilación:

```
El servidor
c:\>vbc timecomponent.vb /t:library /baseaddress:0x12340000

El cliente
c:\>csc /reference:timecomponent.dll timeclient.cs
```

Nota: para poder ejecutar el compilador debemos tener instalado el SDK .NET y tener habilitado el path en la variable d'entorno.

Una vez compilados, si no tenemos ningún problema, aparecerán dos ficheros: *el cliente* que es un ejecutable y el *servidor* que es una dll. Ambos estarán escritos en lenguaje intermedio y metadatos. Si ejecutamos el cliente notaremos como tarda cierto tiempo en mostrar la salida y si lo volvemos a ejecutar una segunda vez durará menos. Esto es debido al compilador JIT (Just-in-time) y a la caché de componentes del CLR. El procedimiento exacto ha sido este:



En este punto podemos pensar: ¿Cuanto lío para ejecutar una aplicación? ¿No perdemos mucho tiempo con IL y el JIT? Y aunque es cierto que perdemos ciclos de reloj con el código administrado a favor del modelo habitual, todas las aplicaciones que nos podamos descargar desde el web funcionan mejor con el código administrado y nos permite despreocupamos de versiones, compatibilidades, dll's que no están, acceso a datos que no tenemos instalados y además el código está optimizado para la máquina cliente.
Existe la posibilidad que estas ventajas no nos acaben de convencer y sigamos dudando de la fiabilidad del modelo porque por ejemplo una aplicación para un cliente que controle el stock de su almacén no necesita de todo esto. Pues bien, para este tipo de aplicaciones más habituales hay una herramienta en el SDK: NGEN.EXE que crea una imagen nativa a partir de un ensamblado administrado y la instala en la caché de imágenes nativas del equipo local. Al

ejecutarla el ensamblado se carga y se ejecuta con mayor rapidez ya que restaura las estructuras de datos y código en la caché de imágenes nativas en lugar de generarlas dinámicamente. Si ejecutamos c:\>ngen /show veremos los componentes que tenemos compilados y cargados en la caché.

```
Microsoft (R) CLR Native Image Generator - Version 1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
CustomMarshalers, Version=1.0.5000.0, Culture=neutral,
                                      PublicKeyToken=b03f5f7f11d50a3a
mscorlib, Version=1.0.5000.0, Culture=neutral,
                                     PublicKeyToken=b77a5c561934e089 <domain neutral>
System, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
System.Design, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
System.Drawing, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
System.Drawing.Design, Version=1.0.5000.0, Culture=neutral,
                                      PublicKeyToken=b03f5f7f11d50a3a
System.Windows.Forms, Version=1.0.5000.0, Culture=neutral,
                                      PublicKeyToken=b77a5c561934e089
System.Xml, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

Salida NGEN.EXE

De todos los paquetes cargados en la caché destacamos el mscorlib que contiene, por ejemplo, los elementos necesarios para configurar el tiempo de ejecución para la criptografía RSA y los componentes del paquete System, el cual veremos más adelante. Aprovechamos estas líneas para mencionar que son los Namespace, (espacios de nombres .NET), tal como habíamos apuntado antes los espacios de nombres son una separación lógica de las funciones, clases , interfaces, etc. La posibilidad de repetir los mismos nombres en espacios de nombres distintos nos facilitará la comprensión y la capacidad de recordar los métodos. Todos los objetos y funciones del CLR de .NET residen en el espacio de nombres System, pero éstos a su vez forman distintas dll's separadas (System.dll, System.Drawing.Design.dll, System.Data.dll, System.Drawing.dll, System.XML.dll System.Windows.Forms.dll, etc..). Por ejemplo las múltiples clases que podamos tener de error no entran en conflicto las unas contra las otras si están separadas por un espacio de nombres.

La nueva unidad: el ensamblado

Los espacios de nombres son una solución en el momento que desarrollamos código y cuando compilamos pero ¿qué sucede una vez hemos desplegado la aplicación? Un ensamblado es una colección lógica de uno o varios archivos (ejecutables, dll, ficheros de recursos) que contienen el código de la aplicación. A este ensamblado se le añade un **manifiesto** que contiene una descripción en metadatos del código y de los recursos que se encuentran dentro de este. ¿En el ejemplo anterior hemos generado un ensamblado? La respuesta es sí. El servidor es un ensamblado de un único fichero al igual que el cliente. Ciertamente los ficheros residen en el directorio y a priori sólo nosotros sabemos que ambos ficheros son dos unidades independientes, y sólo el manifiesto expresará esta condición al igual que si queremos que varios ficheros formen un ensamblado sólo lo indicará el manifiesto. Es por este motivo que el

programador es el responsable que cuando el cargador lea el manifiesto éste pueda encontrar cada uno de los componentes.

Para ver el manifiesto de nuestro servidor podemos recurrir a una herramienta dentro del SDK de .NET: ILDASM.EXE que es un desemsemblador para código IL en el podremos ver muchos detalles del IL. En los siguientes cuadros aparece el código IL de nuestro servidor y cliente.

Número de identificación del componente y control de la versión

MANIFEST del servidor:

```
.assembly extern /*23000001*/ 'mscorlib'
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                    // .z\V.4..
  .ver 1:0:5000:0
}
.assembly extern /*23000002*/ 'Microsoft.VisualBasic'
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )                     // .?_....:
  .ver 7:0:5000:0
}
.assembly /*20000001*/ 'TimeComponent'
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module 'TimeComponent.dll'
// MVID: {2913199B-8837-4C59-9707-A6D7B789B778}
.imagebase 0x12340000
.subsystem 0x00000002
.file alignment 512
.corflags 0x00000001
// Image base: 0x038d0000
```

Código IL del método GetTime del servidor

```
.method /*06000002*/ public instance string
        'GetTime'(bool 'ShowSeconds') cil managed
// SIG: 20 01 0E 02
{
  // Method begins at RVA 0x2058
  // Code size       39 (0x27)
  .maxstack  1
  .locals /*11000001*/ init (string V_0,
           valuetype ['mscorlib'/* 23000001 */]'System'.'DateTime'/* 01000002 */ V_1)
  IL_0000:  /* 03   |                  */ ldarg.1
  IL_0001:  /* 2C   | 12               */ brfalse.s  IL_0015
  IL_0003:  /* 28   | (0A)000002       */ call       valuetype ['mscorlib'/* 23000001 */]'System'.'DateTime'/* 01000002 */
                                                                ['Microsoft.VisualBasic'/* 23000002
                                                                */]'Microsoft.VisualBasic'.'DateAndTime'/* 01000003
*/::'get_Now'() /* 0A000002 */
  IL_0008:  /* 0B   |                  */ stloc.1
  IL_0009:  /* 12   | 01               */ ldloca.s   V_1
  IL_000b:  /* 28   | (0A)000003       */ call       instance string ['mscorlib'/* 23000001 */]'System'.'DateTime'/* 01000002
                                                                        * /::'ToLongTimeString'() /*     0A000003 */
  IL_0010:  /* 0A   |                  */ stloc.0
  IL_0011:  /* 2B   | 12               */ br.s       IL_0025
  IL_0013:  /* 2B   | 10               */ br.s       IL_0025
  IL_0015:  /* 28   | (0A)000002       */ call       valuetype ['mscorlib'/* 23000001 */]'System'.'DateTime'/* 01000002 */
                                  ['Microsoft.VisualBasic'/* 23000002   */]'Microsoft.VisualBasic'.'DateAndTime'/* 01000003
                                                                                        */::'get_Now'() /* 0A000002 */
  IL_001a:  /* 0B   |                  */ stloc.1
  IL_001b:  /* 12   | 01               */ ldloca.s   V_1
  IL_001d:  /* 28   | (0A)000004       */ call       instance string ['mscorlib'/* 23000001 */]'System'.'DateTime'/* 01000002
                                                                        */::'ToShortTimeString'() /*  0A000004 */
  IL_0022:  /* 0A   |                  */ stloc.0
  IL_0023:  /* 2B   | 00               */ br.s       IL_0025
  IL_0025:  /* 06   |                  */ ldloc.0
  IL_0026:  /* 2A   |                  */ ret
} // end of method 'TimeComponent'::'GetTime'
```

MANIFEST del client:

```
.assembly extern /*23000001*/ 'mscorlib'
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                    // .z\V.4..
  .ver 1:0:5000:0
}
.assembly extern /*23000002*/ 'TimeComponent'
{
  .ver 0:0:0:0
}
.assembly /*20000001*/ 'TimeClient'
{
  // --- The following custom attribute is added automatically, do not
uncomment -------
  //  .custom /*0C000001:0A000001*/ instance void ['mscorlib'/* 23000001
*/]'System.Diagnostics'.'DebuggableAttribute'/* 01000002 */::.ctor(bool,
  //
bool) /* 0A000001 */ = ( 01 00 00 01 00 00 )
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module 'TimeClient.exe'
// MVID: {77585669-8825-43E4-ACC8-9723A07E9A3C}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x03a20000
```

Componente remoto o servidor que necesita el cliente para funcionar con la versión correspondiente.

Código IL del método Main del cliente:

```
.method /*06000001*/ public hidebysig static
       void  'Main'() cil managed
// SIG: 00 00 01
{
  .entrypoint
  // Method begins at RVA 0x2050
  // Code size       19 (0x13)
  .maxstack  2
  .locals /*11000001*/ init (class ['TimeComponent'/* 23000002 */]'TimeComponentNS'.'TimeComponent'/*
01000003 */ V_0)
  IL_0000:  /* 73   | (0A)000002      */ newobj     instance void ['TimeComponent'/* 23000002
*/]'TimeComponentNS'.'TimeComponent'/* 01000003 */::.ctor() /* 0A000002 */
  IL_0005:  /* 0A   |                 */ stloc.0
  IL_0006:  /* 06   |                 */ ldloc.0
  IL_0007:  /* 17   |                 */ ldc.i4.1
  IL_0008:  /* 6F   | (0A)000003      */ callvirt   instance string ['TimeComponent'/* 23000002
                 */]'TimeComponentNS'.'TimeComponent'/* 01000003 */::'GetTime'(bool) /* 0A000003 */
  IL_000d:  /* 28   | (0A)000004      */ call       void ['mscorlib'/* 23000001 */]'System'.'Console'/* 01000004
                                                                                  */::'Write'(string) /* 0A000004 */
  IL_0012:  /* 2A   |                 */ ret
} // end of method 'MainApp'::'Main'
```

Constructor

Llamada al componente

Con el manifiesto tenemos una guía de los elementos que necesitamos y sus versiones correspondientes para que el componente inicie. Detallar más acerca del IL no cabe en este curso de ASP.NET. A quien le interese ver cómo está hecho por dentro el IL encontrará toda la información en las MSDN.

Continuemos ahora con los ensamblados y con cómo se instalan éstos en nuestra máquina.
Antes hemos mencionado que el servidor y el cliente debían estar en la misma carpeta; esto es debido a que se hace un **ensamblado privado** de dichos componentes, es decir, cuando el CLR quiera cargar un ejecutable o dll, éste empezará por buscar los componentes que necesite (los del manifiesto) en la misma carpeta en que esté dicho programa. Un ensamblado privado no tendrá mucha utilidad si lo que queremos es publicar algún componente para que otro programa lo quiera usar. Publicar o compartir componentes es lo mismo y se hace a través de la carpeta: \WINNT\assembly que recibe el nombre de GAC (global assembly cache) en el que residen todos los ensamblados compartidos. Accediendo a través del explorador de windows se tiene una interfaz muy cómoda de usar. Arrastrando y soltando podemos añadir componentes.

Probemos de añadir nuestro componente del tiempo al GAC mediante arrastra y soltar, y automáticamente aparece un mensaje de error con el siguiente mensaje: *The located assembly 'Timecomponent.dll' is not a strongly named* . El problema con el que nos encontramos es que si todos los componentes compartidos residen en la misma carpeta es muy fácil que dos componentes tengan el mismo espacio de nombres y no podamos resolver una llamada a un componente. En las versiones anteriores de COM toda la información, que se ahora se almacena en el GAC, antes se almacenaba en el registro de windows (control de la versión y ubicación del componente) pero era un auténtico drama cuando se debían reemplazar componentes sobrescritos por error o no con versiones más recientes. Entonces se tenían que usar herramientas que limpiaran el registro de entradas perdidas. En fin, un auténtico rompecabezas. En cambio, con esta nueva solución no tenemos tantos problemas como antes y no hay que buscar el mismo componente en distintas entradas del registro. En contra hemos de preocuparnos para que los componentes no se puedan solapar.

Para evitar que los componentes se solapen .NET se aprovecha de la tecnología de clave pública y clave privada para cifrar los componentes. Hay una herramienta en el SDK de .NET llamada SN.EXE que genera claves públicas. Si ejecutamos c:\>SN –K llave.snk nos generará un fichero llamado llave.snk que está en formato interno de la máquina y por tanto no podemos verlo a través del notepad, sí a través de Visual Studio .NET . Ahora lo que tenemos que hacer es volver a compilar el servidor y usar el modificador /keyfile para añadir la clave a nuestro ensamblado, veamos un ejemplo:

Compilación con nombre seguro:

```
vbc timecomponent.vb /t:library /baseaddress:0x12340000 /keyfile:llave.snk
```

Nota: recordad que los ficheros han de estar en la misma carpeta.

Podemos testear si nuestro componente está firmado correctamente mediante la instrucción –T del comando SN.EXE . Volvemos a compilar el cliente para que tome la nueva clave del componente. Si

```
….
.assembly extern TimeComponent
{
  .publickeytoken = (31 13 DD 48 91 9E AC 03 )
// 1..H....
  .ver 0:0:0:0
}
.....
```

visualizamos el manifiesto del componente cliente apreciaremos como aparece al clave pública del componente servidor.

Y por último sólo nos quedará arrastrar y soltar el nuevo componente servidor en la carpeta \winnt\assembly (GAC) y podremos usar este componente desde cualquier punto de nuestro ordenador.

Nota: una de las formas de publicar componentes COM a través de internet y usando el IIS fue la tecnología DCOM que permitía usar COM distribuido (la alternativa a CORBA) a través de la red. Cuando entremos en la sección de servicios web veremos como todo el proceso de antes se simplifica enormemente.

Segundo contacto

Otra vez con nuestro editor favorito escribimos el siguiente código:

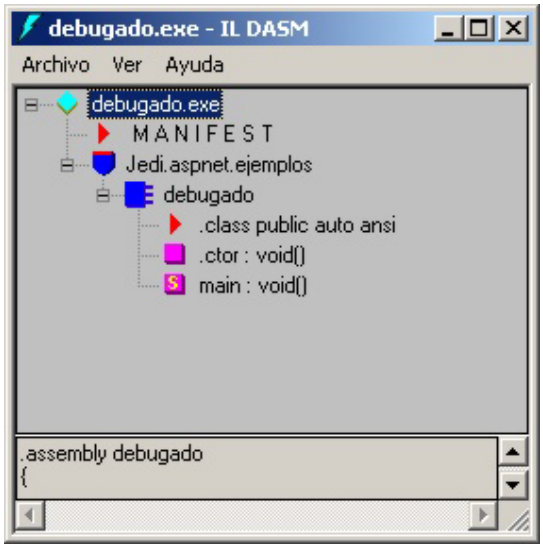
```
Imports System
Imports Microsoft.VisualBasic


Namespace Jedi.aspnet.ejemplos

    Public Class debugado

        Public Shared Sub main()

            Console.WriteLine("Pruebas del debuger")
            Console.Read()

        End Sub

    End Class

End Namespace
```

Nombre: debugado.vb

Y lo compilamos con:

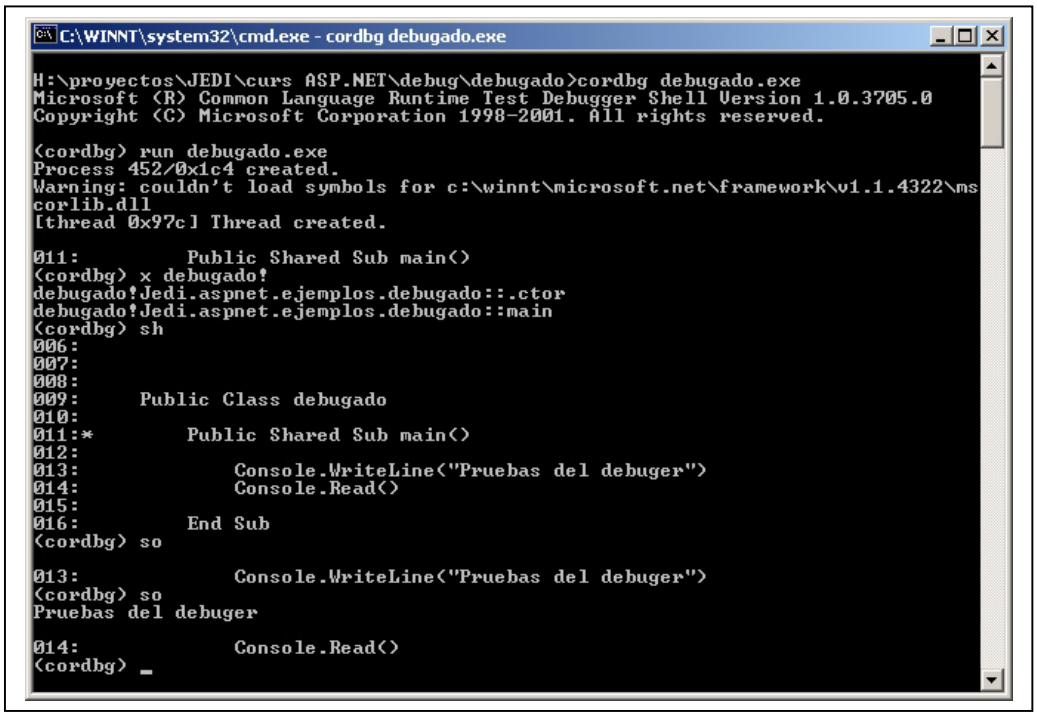
```
vbc debugado.vb /t:exe /debug
```

El resultado de esta compilación son dos ficheros: debugado.exe y debutado.pdb con estos dos ficheros veremos como podemos debugar en línea de comandos. Empezamos por abrir de nuevo ILDASM.EXE sobre el nuevo fichero ejecutable debugado.exe. Como antes, el manifiesto contendrá la enumeración de todos los ensamblados que necesita este ejecutable, pero quiero destacar ahora que el método principal es de tipo **shared** la inclusión de la **s** como identificador de tal categoría. Si examinamos el código IL sin tantas opciones como antes, es el siguiente:



```
.method public static void  main() cil managed
{
 .entrypoint
 .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = ( 01 00 00 00 )
 // Code size       20 (0x14)
 .maxstack  8
 IL_0000:  nop
 IL_0001:  ldstr      "Pruebas del debuger"
 IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
 IL_000b:  nop
 IL_000c:  call       int32 [mscorlib]System.Console::Read()
 IL_0011:  pop
 IL_0012:  nop
 IL_0013:  ret
} // end of method debugado::main
```

Sin perder de vista este código ahora aprenderemos a usar el debuger en línea de comandos: CORDBG.EXE debugado.exe . Los comandos que debemos saber son: **s** para avanzar paso a paso, **so** para avanzar función a función, **x <nombre_ejecutable>!** para saber las funciones que contiene, **sh** para mostrar el código fuente, **?** para mostrar todas las operaciones posibles y **q** para salir.



Gracias a la API del debuger en CLR podemos ver los ejecutables "hasta el fondo". Cabe recordar que debemos usar la opción /debug en la compilación. La última herramienta que nos queda por ver es DUMPBIN.EXE que se instala con Visual Studio .NET . Como hemos hecho antes, abrimos nuestro ejecutable con DUMPBIN.EXE /ALL debugado.exe . La salida que se nos muestra está dividida en varias secciones: la primera parte son metadatos del ejecutable (lo mismo que hemos visto en el ILDASM), la segunda parte es el binario del fichero debugado.pdb y el resto de opciones del ensamblado, etc...