

BREVE HISTORIA DE LA PROGRAMACIÓN PARA WINDOWS

Hace algunos años la única forma como se podía programar para Windows era hacer uso de un compilador de C o C++ y de un **API** de Windows. El API es una gran colección de funciones que se relacionan, las que nos permiten comunicarnos con el sistema operativo. Por medio del API de **Win32** se programaban las ventanas, botones y demás elementos.

El problema de este tipo de programación es que el API de Win32 es realmente complejo y enorme. Con miles de funciones en su interior, por lo que pocos programadores podían conocerlo en su totalidad. Pero la complejidad no solamente estaba en la cantidad de funciones, también en la sintaxis y la forma como se programa.

Para facilitar la programación de aplicaciones para Windows surgen diferentes opciones; la finalidad de estos intentos era poder hacer las aplicaciones sin tener que pasar por la complejidad de Win32. Uno de estos intentos fue conocido como **OWL**; sin embargo, obtuvo más éxito **MFC**, creado por Microsoft.

MFC es un conjunto de clases que envuelve a Win32 y facilita su programación. Con MFC los procesos más comunes se agrupan en funciones de tal forma que con una simple llamada a una función de MFC se puede hacer una determinada tarea, para la que antes necesitábamos por lo menos 10 llamadas en Win32 y muchos parámetros. Sin embargo Win32 está debajo de MFC; la programación MFC simplifica mucho las cosas, pero muchos programadores que venían del paradigma de programación estructurada no se sentían a gusto con él.

Otra de las opciones que surgieron es **Visual Basic**, este lenguaje logró gran popularidad, especialmente en Latinoamérica. Visual Basic también trabaja por arriba de Win32, pero basa su sintaxis en el antiguo lenguaje **Basic**. Es muy sencillo de aprender y una de las características que le dio gran popularidad fue la facilidad con la que se podían crear interfaces de usuario y conectividad a bases de datos. Pero hasta antes de la versión .NET, este lenguaje tenía ciertos limitantes ya que no se podía llevar a cabo programación orientada a objetos con él.

Otro lenguaje que surge, pero con su propio **Framework**, es **JAVA**; su principal ventaja es ser multiplataforma. Una de sus características es el uso de un **runtime**, la aplicación en lugar de correr directamente en el microprocesador, se ejecuta

en un programa llamado runtime y este se encarga de ejecutar el código en el microprocesador correspondiente. Si se tiene el runtime para Windows, sin problema se ejecuta el programa de JAVA.

Cuando nosotros deseábamos tener un programa que se pudiera ejecutar, era necesario **compilarlo**. Cada uno de los lenguajes tenía su propio **compilador**, por ello no era sencillo poder compartir código de C++ con código de Visual Basic ya que el traducir entre lenguajes era difícil. Para poder compartir código entre los lenguajes surge un mo-

delo conocido como **COM**, éste nos permite crear componentes binarios, esto quiere decir que es posible programar un componente en Visual Basic y un programador de C++ puede tomarlo y hacer uso de él. Esto se debe a que el componente ya es código compilado y no código fuente en el lenguaje de origen; la programación de COM también tenía sus complejidades y surge **ATL** para ayudar en su desarrollo.

Con todo esto, llega el momento en el cual es necesario ordenar, facilitar y organizar el desarrollo de las aplicaciones para Windows, con esta filosofía surge .NET.

Descubrir .NET

El **Framework** de .NET es una solución a toda la problemática en torno al desarrollo de aplicaciones, brinda grandes beneficios no solamente al desarrollador, sino también al proceso de desarrollo. En primer lugar .NET permite trabajar con código ya existente, podemos hacer uso de los componentes COM, e incluso, si lo necesitáramos usar el API de Windows. Cuando el programa .NET está listo es mucho más fácil de instalar en la computadora de los clientes, que las aplicaciones tradicionales ya que se tiene una integración fuerte entre los lenguajes.

Un programador de C# puede entender fácilmente el código de un programador de Visual Basic .NET y ambos pueden programar en el lenguaje con el que se sienten más cómodos. Esto se debe a que todos los lenguajes que hacen uso de .NET comparten las librerías de .NET, por lo que no importa en qué lenguaje programemos, las reconocemos en cualquiera. A continuación conoceremos los diferentes componentes de .NET: **CLR**, **assembly** y **CIL**.

CLR

El primer componente de .NET que conoceremos es el **Common Language Runtime**, también denominado **CLR**. Este es un programa de ejecución común a todos los lenguajes. Este programa se encarga de leer el código generado por el compilador y empieza su ejecución. Sin importar si el programa fue creado con C#, con Visual Basic .NET o algún otro lenguaje de .NET el CLR lo lee y ejecuta.

Assembly

Cuando tenemos un programa escrito en un lenguaje de .NET y lo compilamos se genera el **assembly**. El assembly contiene el programa compilado en lo que conocemos como CIL y también información sobre todos los tipos que se utilizan en el programa.

CIL

Los programas de .NET no se compilan directamente en código ensamblador del compilador, en su lugar son compilados a un lenguaje intermedio conocido como CIL. Este lenguaje es leído y ejecutado por el runtime. El uso del CIL y el runtime es lo que le da a .NET su gran flexibilidad y su capacidad de ser multiplataforma.

El Framework de .NET tiene lo que se conoce como las **especificaciones comunes de lenguaje** o **CLS** por sus siglas en inglés, estas especificaciones son las guías que cualquier lenguaje que desee usar .NET debe de cumplir para poder trabajar con el runtime. Una ventaja de esto es que si nuestro código cumple con las CLS podemos tener interoperabilidad con otros lenguajes, por ejemplo, es posible crear una librería en C# y un programador de Visual Basic .NET puede utilizarla sin ningún problema.

Uno de los puntos más importantes de estas guías es el **CTS** o **sistema de tipos comunes**. En los lenguajes de programación, cuando deseamos guardar información, ésta se coloca en una variable, las variables van a tener un tipo dependiendo de la información a guardar, por ejemplo, el tipo puede ser para guardar un número entero, otro para guardar un número con decimales y otro para guardar una frase o palabra. El problema con esto es que cada lenguaje guarda la información de manera diferente, algunos lenguajes guardan los enteros con **16 bits** de memoria y otros con **32 bits**; incluso algunos lenguajes como C y C++ no tienen un tipo para guardar las **cadena**s o frases.

Para solucionar esto el Framework de .NET define por medio del CTS cómo van a funcionar los tipos en su entorno. Cualquier lenguaje que trabaje con .NET debe de usar los tipos tal y como se señalan en el CTS. Ahora que ya conocemos los conceptos básicos, podemos ver cómo es que todo esto se une.

Cómo se crea una aplicación .NET

Podemos crear una aplicación .NET utilizando un lenguaje de programación, para este efecto será C#; con el lenguaje de programación creamos el código fuente del programa (instrucciones que le dicen al programa qué hacer).

Cuando hemos finalizado con nuestro código fuente, entonces utilizamos el compilador. El compilador toma el código fuente y crea un assembly para nosotros, es-

te assembly tendrá el equivalente de nuestro código, pero escrito en CIL; esto nos lleva a otra de las ventajas de .NET: nuestro código puede ser optimizado por el compilador para la plataforma hacia la cual vamos a usar el programa, es decir que el mismo programa puede ser optimizado para un dispositivo móvil, una PC normal o un servidor, sin que nosotros tengamos que hacer cambios en él.

III .NET ES MULTIPLATAFORMA

El Framework de .NET se puede ejecutar en muchas plataformas, no solo en Windows. Esto significa que podemos programar en una plataforma en particular y si otra plataforma tiene el runtime, nuestro programa se ejecutará sin ningún problema. Un programa .NET desarrollado en Windows puede ejecutarse en Linux, siempre y cuando se tenga el runtime correspondiente.

Cuando nosotros deseamos invocar al programa, entonces el runtime entra en acción, lee el assembly y crea para nosotros todo el entorno necesario. El runtime empieza a leer las instrucciones CIL del assembly y conforme las va leyendo las compila para el microprocesador de la computadora en la que corre el programa; esto se conoce como **JIT** o **compilación justo a tiempo**. De esta manera conforme se avanza en la ejecución del programa se va compilando; todo esto ocurre de manera transparente para el usuario.

El Framework de .NET provee, para los programas que se están ejecutando, los servicios de **administración de memoria** y **recolector de basura**. En lenguajes no administrados como C y C++ el programador es responsable de la administración de memoria, en programas grandes esto puede ser una labor complicada, que puede llevar a errores durante la ejecución del programa. Afortunadamente lenguajes administrados como C# tienen un modelo en el cual nosotros como programadores ya no necesitamos ser responsables por el uso de la memoria. El recolector de basura se encarga de eliminar todos los objetos que ya no son necesarios, cuando un objeto deja de ser útil el recolector lo toma y lo elimina. De esta forma se liberan memoria y recursos.

El recolector de basura trabaja de forma automática para nosotros y ayuda a eliminar toda la administración de recursos y memoria que era necesaria en Win32. En algunos casos especiales como los archivos, las conexiones a bases de datos o de red se tratan de recursos no administrados, para estos casos debemos de indicar explícitamente cuando es necesario que sean destruidos.

Cómo conseguir un compilador de C#

Existen varias opciones de compiladores para C#, en este libro utilizaremos la versión de C# que viene con **Visual Studio 2010**, pero también es posible utilizar versiones anteriores. Este compilador, al momento de publicación de este libro, utiliza la versión 4.0 del Framework. La mayoría de los ejemplos deben poder compilarse y ejecutarse sin problema, aún utilizando versiones anteriores del Framework, esto es válido incluso para aquellos programadores que decidan trabajar con el llamado Mono, como alternativa libre y gratuita.

III EL COMPILADOR JIT

Al **compilador JIT** también se le conoce como **Jitter**. Forma parte del runtime y es muy eficiente, si el programa necesita volver a ejecutar un código que ya se ha compilado, el Jitter en lugar de volver a compilar, ejecuta lo ya compilado, mejorando de esta forma el desempeño y los tiempos de respuesta de cara al usuario.

Una mejor opción, en caso de que no podamos conseguir la versión profesional de Visual Studio 2010, es la versión **Express**. Esta versión es gratuita y se puede descargar directamente de Internet, lo único que se necesita es llevar a cabo un pequeño registro por medio de cualquier cuenta de **Hotmail** o **passport**.

Para descargar el compilador de **C# Express** de forma completamente gratuita, podemos acceder al sitio web que se encuentra en la dirección **www.microsoft.com/express**, dentro del portal de Microsoft. Para realizar esta descarga sólo será necesario completar un formulario de registro.

Una vez que hemos descargado el compilador, podemos proceder a llevar a cabo la instalación; esta tarea es muy similar a la instalación de cualquier otro programa de Windows, pero no debemos olvidar registrarlo antes de 30 días.