LA PLATAFORMA DE DESARROLLO .NET

La plataforma de desarrollo .NET es la apuesta de Microsoft para el desarrollo, el despliegue y la ejecución de la nueva generación de software y servicios Web. Está concebida para proporcionar independencia de la plataforma de desarrollo y para facilitar el desarrollo de aplicaciones distribuidas e interoperables, todo ello a través de un entorno de desarrollo multilenguaje que facilite la combinación de módulos desarrollados en distintos lenguajes (o la mera reutilización de código). Inicialmente los lenguajes soportados por .NET eran *C#, J#, C++* y *Visual Basic* pero, debido al soporte multilenguaje que ofrece, se han ido añadiendo nuevos lenguajes para aumentar el atractivo hacia desarrolladores diversos. Entre los nuevos lenguajes soportados se encuentran *Cobol, RPG, APL, Perl, Pascal, SmallTalk, Eiffel, Fortran, Haskell, Mercury, Oberon, Oz, Python, Scheme, Standard ML*, etc.

La arquitectura de la plataforma ha sido diseñada totalmente desde cero, con el objetivo de evitar los problemas que se venían arrastrando de las antiguas APIs (Application Programming Interfaces) de Windows, que por motivos de compatibilidad hacia atrás se habían ido manteniendo, arrastrando los problemas de diseño iniciales. Se ha simplificado el modelo de programación, y los servicios que antes eran ofrecidos por funciones encapsuladas en DLLs o mediante objetos COM, en .NET se proporcionan mediante un modelo de programación orientado a objetos. Además, para evitar los problemas derivados de la sobreescritura de nuevas versiones de librerías compartidas por varias aplicaciones, se han introducido mecanismos de versionado para que puedan coexistir sin problemas diferentes versiones de una misma librería.

Por otra parte, .NET *Framework* está basado en estándares, a diferencia de lo que venía ocurriendo con las alternativas anteriores de Microsoft. Las especificaciones del CLR (la máquina de ejecución de .NET) y de C# se han publicado entre los estándares ECMA y además parte del código fuente y de la biblioteca de clases está disponible públicamente. Aunque el entorno .NET se ha proclamado como independiente del sistema operativo, sólo se encuentra disponible en los sistemas operativos de Microsoft. No obstante, el proyecto MONO [Mono], liderado por Ximian y patrocinado por Novell, está consiguiendo con éxito la portabilidad del *Framework* a las plataformas GNU/Linux, Unix y Mac OS X.

A continuación se describirán las principales características del *Framework*, sin entrar en un alto nivel de detalle, pues el objetivo es dar a conocer algunos conceptos a los cuales se referirá en este proyecto en otros capítulos. Para más detalles puede consultarse [Ar02] que, aunque centrado en el lenguaje C#, muestra en profundidad los conceptos más importantes sobre la arquitectura de .NET.

A.1 Interoperabilidad entre lenguajes: el CTS y los Metadatos

La interoperabilidad entre lenguajes ha sido posible gracias a la definición de un Sistema de Tipos Común (CTS, *Common Type System*) y a la introducción de los metadatos. Mediante el CTS se ha podido establecer un marco común para definir los tipos básicos que todo lenguaje soportado por .NET puede tener. Con los metadatos se ha establecido un mecanismo para que los distintos lenguajes puedan obtener la descripción de los tipos y utilizarlos sin necesidad de recurrir a definiciones externas de las interfaces, como es el caso de CORBA, en el que deben generarse por separado las interfaces que publican sus tipos. De esta forma, clases programadas en diferentes lenguajes pueden combinarse sin problemas en una misma aplicación, favoreciendo por tanto la reutilización de código.

Las características del CTS son:

- especifica cómo se declaran, utilizan y gestionan los tipos en el marco de ejecución,
- proporciona un modelo orientado a objetos que soporta la implementación de muchos lenguajes de programación,
- define las reglas que los lenguajes deben seguir para su interoperabilidad.

El CTS se define en el *Common Language System* (*CLS*), que define las reglas que deben cumplir todos los compiladores de .NET para generar código interoperable. Una de estas reglas es que el compilador debe soportar todos los tipos definidos en el CTS, con lo que se asegura que objetos o tipos creados en diferentes lenguajes sean compatibles entre sí. Por otra parte, el CTS define una jerarquía única de objetos, es decir, cualquier tipo definido deriva necesariamente del tipo base *System.Object*, y con ello se garantiza que todos los tipos tendrán una interfaz común derivada de este tipo base y por tanto una funcionalidad básica (los servicios *Equals, Finalize, GetHashCode* y *ToString*).

Los metadatos son información binaria que describe los tipos implementados por una aplicación. Todos los compiladores de los diferentes lenguajes soportados por .NET, además de emitir el correspondiente código intermedio, están obligados a emitir los metadatos sobre cada tipo contenido

en los ficheros fuente. De esta forma, los ficheros así generados son autodescriptivos, manteniéndose siempre sincronizado el código con las descripciones de sus tipos y permitiendo que cualquier lenguaje pueda obtener la información de los tipos que contienen. Además, el desarrollador puede crear nuevos metadatos para añadir información adicional o incluso más funcionalidad semántica a los tipos, mediante el uso de los *atributos*, que serán descritos más adelante.

Ensamblados

Estas unidades autodescriptivas que agrupan varios tipos y sus correspondientes metadatos, se denominan *Ensamblados* (*Assemblies*). Se definen como los bloques de construcción de las aplicaciones para la plataforma .NET, siendo la unidad fundamental de despliegue, de reutilización y de control de versiones, así como la unidad sobre la que se aplican las políticas de seguridad. Un ensamblado es una colección de tipos y recursos que constituyen una unidad lógica de funcionalidad, proporcionando la información que el CLR necesita sobre las implementaciones de dichos tipos.

A.2 El CLR: La máquina virtual de .NET

De igual forma que Java, la plataforma .NET proporciona un marco de ejecución para las aplicaciones que permite gestionar el código en ejecución, liberando al programador de cuestiones como la gestión de memoria, la recolección de basura, la gestión de excepciones, la gestión de los hilos de ejecución o los componentes remotos. Este marco de ejecución (que en Java sería la máquina virtual) es el CLR (*Common Language Runtime*) y podría considerarse como el núcleo del *framework*, pues todas las aplicaciones son gestionadas y supervisadas por él.

Con el objetivo de proporcionar una capa de abstracción adicional del hardware, favorecer una ejecución multiplataforma y la integración entre diferentes lenguajes, se ha diseñado un lenguaje intermedio: el MSIL (*MicroSoft Intermediate Language*). Este lenguaje intermedio es independiente del juego de instrucciones de una CPU específica y proporciona un nivel de abstracción superior a los lenguajes nativos comunes pues incluye instrucciones para el tratamiento de objetos, de excepciones, tablas, etc. Los distintos compiladores de .NET se encargan de generar este lenguaje intermedio y el CLR es el encargado de verificar dicho código y de ejecutarlo, transformándolo a código nativo.

El proceso de ejecución de código pasa por dos etapas: la compilación a código intermedio, y la carga y ejecución de este código intermedio por el CLR. En la Figura 43 se muestra el proceso de transformación que sigue el código fuente hasta poder ser ejecutado. En la primera fase, a partir del

código fuente, el compilador genera un ensamblado que contiene los metadatos que describen los tipos definidos y el código intermedio.

En la segunda fase, cuando se requiera la ejecución de un ensamblado, el código IL es interpretado por el compilador JIT (*Just-In-Time*) que lleva el CLR, es convertido a lenguaje máquina y puesto en ejecución. Si dicho código tiene dependencias con otros ensamblados, gracias a los metadatos el CLR los va cargando sucesivamente, siguiendo el mismo proceso.

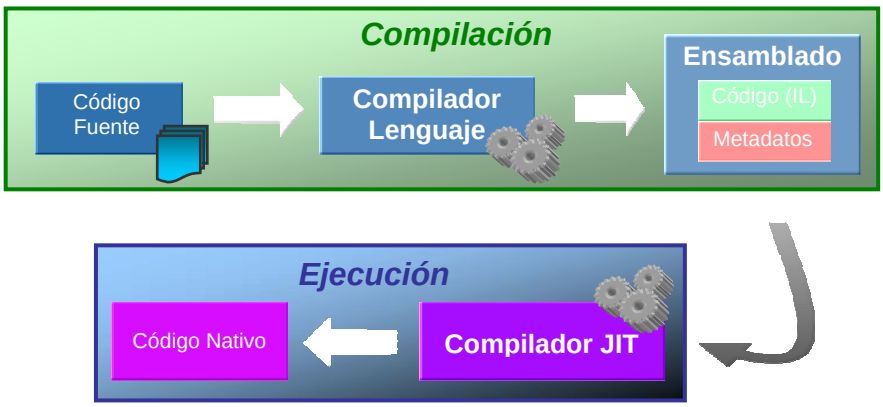

Figura 43 - Proceso de compilación en .NET

El compilador JIT va convirtiendo dinámicamente el código MSIL a ejecutar en código nativo según sea necesario. Únicamente convierte el código que se vaya a ejecutar, almacenándolo por si fuera necesario en futuras llamadas. Existe una versión del compilador, el *prejitter*, mediante la cual es posible compilar completamente cualquier ensamblado a código nativo, con lo que futuras ejecuciones no necesitarán de la compilación dinámica.

El CLR, además de encargarse de cargar y ejecutar el código, también se encarga de verificar la seguridad de la aplicación que está ejecutando, con el objetivo de evitar operaciones inválidas o que accedan a zonas de memoria no permitidas. Para ello, el CLR utiliza los siguientes mecanismos:

- Verifica la seguridad de los tipos, a través del CTS y aísla la memoria dedicada a la aplicación. Con estas medidas se pretende evitar los fallos comunes de *buffer overrun* (desbordamiento de búfer, a causa del cual se accede al área de memoria de otra aplicación), que los tipos están dentro de los rangos establecidos, etc.
- Comprueba que la aplicación en ejecución dispone de los permisos adecuados sobre los recursos que quiere utilizar, que dependerá de la confianza que el usuario haya depositado sobre dicha aplicación.
- Gestiona las excepciones provocadas, dotando a la aplicación de mecanismos para la gestión de sus errores.

Sin embargo, .NET permite generar código no administrado, es decir, no supervisado por el CLR, con la finalidad de permitir el desarrollo de *drivers* específicos que necesiten acceder a recursos de muy bajo nivel o para el desarrollo de aplicaciones cuya eficiencia sea crítica. El código no

administrado es código nativo, y por lo tanto, no goza de los servicios que ofrece el CLR en tiempo de ejecución. No obstante, también se proporcionan mecanismos para que el código administrado por el CLR pueda comunicarse con código no administrado, que incluye desde objetos COM hasta código incluido en DLLs.

A.3 La Librería de Clases

Con el objetivo de facilitar el desarrollo de aplicaciones, y de la misma forma que Java, .NET también proporciona una extensa librería de clases y tipos reutilizables que permiten al desarrollador invertir el mínimo tiempo posible en el desarrollo de la infraestructura de la aplicación, centrándose más en la lógica de negocio. Además, como estas librerías están construidas sobre la naturaleza orientada a objetos del CLR, su funcionalidad puede ser extendida a través de la herencia. Y como también están escritas en código intermedio, podrán ser utilizadas por cualquier lenguaje cuyo compilador genere código MSIL (es decir, lenguajes soportados por .NET).

Debido a la gran cantidad de clases existentes, éstas se agrupan de forma jerárquica en lo que se denominan *espacios de nombres* (*namespaces*), los cuales agrupan clases de funcionalidades similares e independientes. En la Figura 44 se puede observar la estructura dividida en capas que constituye el entorno .NET.

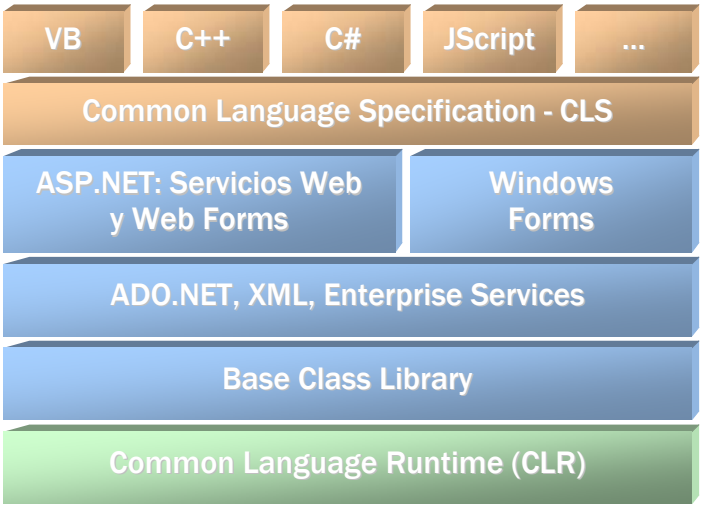

Figura 44 - .NET Framework

En primer lugar se encuentran los distintos lenguajes, que deben cumplir las reglas especificadas por el CLS. En último lugar se encuentra el CLR que es el encargado de cargar y ejecutar las aplicaciones de .NET en la máquina destino. Entre estas dos capas se encuentran una serie de niveles que representan los distintos servicios que ofrece .NET a través de sus librerías:

- *ASP.NET*: proporciona una infraestructura y un nuevo modelo de programación para los desarrollos basados en Web. Está formado por dos modelos de programación: las aplicaciones web y los Servicios Web.
- *Windows Forms*: proporciona los servicios necesarios para el desarrollo de interfaces de usuario de escritorio, introduciendo un único conjunto de clases para cualquier lenguaje de .NET.
- *ADO.NET:* forma el conjunto de servicios necesarios para el acceso a fuentes de datos, utilizando XML como formato de transmisión de datos. Basado en el modelo anterior de ADO, aumenta su escalabilidad e interoperabilidad, y permite el trabajo sobre conjuntos de datos desconectados (datos cuya procedencia no está disponible temporalmente).
- *XML:* ofrece clases para crear y formatear documentos XML .
- *Enterprise Services:* proporciona un conjunto de servicios para la compatibilidad con componentes COM+.
- *Base Class Library*: además de las clases y tipos básicos, contiene todo un conjunto de servicios para el soporte de las operaciones más comunes en el desarrollo de la mayoría de aplicaciones como son el manejo de excepciones, el tratamiento de fechas, la gestión de la entrada/salida, tablas, pilas y colas, etc.

A.4 .NET Remoting

De la misma forma que .NET ha venido a sustituir la tecnología COM para el desarrollo de componentes, .NET Remoting es la nueva tecnología que ha propuesto Microsoft para sustituir a DCOM, el método más extendido en plataformas Win32 para el desarrollo de aplicaciones distribuidas. .NET Remoting es una arquitectura orientada a objetos que facilita el desarrollo de aplicaciones distribuidas, permitiendo que dos objetos que se encuentren en máquinas distintas, comunicadas a través de una red cualquiera, puedan comunicarse entre sí. Además de ofrecer una gran cantidad de servicios y protocolos estándar ya implementados, permite usar y/o construir formatos de codificación o protocolos de comunicación distintos. Frente a otras tecnologías para el desarrollo de aplicaciones distribuidas, presenta como ventaja su facilidad de uso, pues no necesita la generación de interfaces en un lenguaje abstracto como en CORBA o DCOM, ni tampoco ciclos de generación de proxys/stubs como en Java/RMI. También proporciona los mecanismos básicos para los Servicios Web de .NET, implementando estándares como SOAP y WSDL. Para más detalles, puede consultarse [Ramm02] y [Scott03] donde se trata en profundidad la tecnología .NET Remoting.

A.5 Aspectos avanzados de .NET

El entorno .NET añade una serie de tecnologías que, utilizadas convenientemente, permite al desarrollador generar dinámicamente ensamblados, o incluso permitir la evolución dinámica del software desarrollado. A continuación se describirán brevemente algunos de los conceptos más relevantes, como los nuevos tipos añadidos (atributos y delegados) o tecnologías orientadas a la emisión dinámica de código (reflexión, emisión de código y CodeDom).

A.5.1 Atributos

Como se ha comentado anteriormente, los metadatos permiten describir los tipos implementados por una aplicación, pero además, el desarrollador tiene la posibilidad de añadir nuevos metadatos que aporten más información, a través de los *Attributes* de .NET. Los atributos se definen como construcciones del lenguaje que permiten añadir información adicional (metadatos) a los elementos del lenguaje y extender su funcionalidad. Pueden ser asociados a: clases, métodos, interfaces, ensamblados, parámetros, propiedades, etc.

De esta forma, pueden crearse metadatos que representen información de diseño (como documentación sobre los tipos a los que van asociados), información en tiempo de ejecución (como el nombre de una columna de una base de datos o de un campo) o hasta nuevas características de comportamiento en tiempo de ejecución (como indicar que un miembro es serializable, o que se ha de comportar de determinada forma).

El uso de los atributos puede separarse en tres etapas: en la primera se define el atributo como una clase derivada de la clase *Attribute*, utilizando para ello las construcciones comunes de cualquier lenguaje; en la segunda etapa se asocia el atributo a los elementos correspondientes; y en la tercera etapa, en tiempo de ejecución y a través de la reflexión, se obtienen los atributos asociados a un elemento del código para actuar de la forma que se estime oportuna. Además, como los atributos se definen como clases, pueden contener propiedades adicionales que codifiquen información relativa al estado del atributo.

A.5.2 Delegados

Una innovación del CTS es el tipo *Delegate (*delegados). Son tipos de referencia que encapsulan un método con una signatura específica y se utilizan para proporcionar procesamiento asíncrono de servicios, para inyectar código en llamadas a funciones genéricas (por ejemplo, para proporcionar la forma de ordenar elementos en una función de ordenación) y para el manejo de eventos de forma asíncrona. De forma similar a los punteros a funciones existentes en el lenguaje C, los delegados encapsulan

la referencia a un método y se comportan como variables, por lo que pueden ser pasados como parámetros en otras funciones.

A.5.3 Reflexión

Los servicios de reflexión permiten, a través de los metadatos, obtener en tiempo de ejecución información sobre los tipos definidos en una aplicación o de los módulos a los que acceda. De esta forma, puede obtenerse dinámicamente el nombre del ensamblado que se está ejecutando, interrogar a distintas clases por los métodos que ofrecen, los campos y propiedades públicos, etc. Para ello, mediante los servicios ofrecidos por el espacio de nombres *System.Reflection* puede obtenerse el tipo de cualquier instancia e interrogarlo para obtener todas sus propiedades.

Una primera aplicación que se puede derivar de este mecanismo es el "enlace tardío" (*o Late Binding*) mediante el cual las librerías dinámicas se pueden cargar y enlazar dinámicamente a la aplicación en función de si contienen clases específicas requeridas por la aplicación (por ejemplo, que herede de una determinada clase, o que implementen determinada interfaz). Además, la reflexión también es utilizada por otras librerías de .Net con diversos fines: por ejemplo para serializar objetos (guardar el estado de un objeto que se encuentra en memoria) se utiliza la reflexión para obtener los campos que pueden ser serializados o no.

A.5.4 Emisión de código

Sin embargo, la principal ventaja de los servicios de reflexión proporcionados por la librería es la posibilidad de emitir dinámicamente código. Para ello se utilizan los tipos y sus correspondientes servicios definidos en el espacio de nombres *System.Reflection.Emit* para crear primero un ensamblado en memoria, definir los tipos que lo formarán (junto a sus miembros) y proceder a emitir instrucciones MSIL. Posteriormente, el ensamblado podrá guardarse en disco o dejarse en memoria si no se necesita la persistencia del código generado.

El principal inconveniente que plantea este mecanismo es que requiere por parte del desarrollador un conocimiento profundo del código intermedio, a medio camino entre los lenguajes de alto nivel y el lenguaje máquina, lo que puede provocar numerosos errores de codificación.

A.5.5 CodeDom

Frente a la emisión de código intermedio, pueden utilizarse los servicios del espacio de nombres *System.CodeDom* para generar código dinámicamente a partir de plantillas de código o para la compilación dinámica de código. Para ello, utilizando tipos básicos definidos en dicha librería y que representan cada uno de los tipos disponibles en el *Common Type System*, se construye un grafo de objetos. Este grafo de objetos proporciona un modelo de objetos independiente de cualquier lenguaje de programación y representa la

estructura del código fuente en memoria. Este grafo podrá ser transformado a cualquier lenguaje utilizando para ello un generador de código *CodeDom* específico para el lenguaje de destino. Por otra parte, *CodeDom* también proporciona servicios para compilar dinámicamente ficheros de código fuente a un ensamblado binario.

Se ha de tener en cuenta que *CodeDom* no ha sido diseñado para soportar todas las estructuras y tipos que se pueden encontrar en los distintos lenguajes de programación soportados por .NET, sino tan sólo aquellos tipos básicos comunes a todos los lenguajes, es decir, los definidos en el CTS. Sin embargo, este aspecto ha sido solucionado a través de unas estructuras sintácticas que permiten añadir código específico de un lenguaje (los *codeSnippets*), pero no se garantiza la conversión de dicho código a otro lenguaje distinto.

La principal ventaja es que el desarrollador, tras aprender las estructuras sintácticas de *CodeDom* podrá generar código dinámicamente sin necesidad de conocer el lenguaje intermedio que se genera, utilizando para ello el mismo lenguaje en el que ha desarrollado la aplicación. Otra utilidad derivada de esta tecnología es transformar código fuente de un lenguaje a otro lenguaje, aunque en la actualidad aún no se encuentra totalmente soportada, pues el ritmo en el que se añaden nuevas estructuras al CLR no es el mismo al que se implementan dichos cambios en los distintos compiladores de *CodeDom*.