Fundamentos de la Máquina Virtual Java y el Entorno .NET

Trabajo Práctico

Teoría y Aplicaciones de la Informática 2

Nathalie M. Aquino Salvioni
Juan Carlos Frutos Acosta

Universidad Católica Nuestra Señora de la Asunción
Facultad de Ciencias y Tecnología
Ingeniería Informática
5to. Año – 10mo. Semestre.
Octubre – 2002

Índice

1. Introducción

El presente trabajo trata sobre la Máquina Virtual Java y el Entorno .NET.

Se describe la arquitectura de la Máquina Virtual Java, que es parte medular de toda la tecnología Java. Se hace énfasis en sus componentes principales, como el procesador virtual Java, que se encarga de ejecutar los códigos de operación (bytecodes) generados por los compiladores Java, el verificador de código, que junto con el cargador de clases y el administrador de seguridad, se encargan de implementar los mecanismos empleados para proporcionar seguridad a los usuarios de Java. Además se van viendo las características del lenguaje Java y los puntos fuertes y débiles de la tecnología Java en general.

En cuanto al Entorno .NET se analizan los componentes principales, el Lenguaje Común en Tiempo de Ejecución (CLR), el Sistema Común de Tipos (CTS), el Lenguaje Intermedio de Microsoft (MSIL), entre otros aspectos. También se dan unas características del lenguaje C#.

2. La Máquina Virtual Java – MVJ (Java Virtual Machine - JVM)

2.1 Java

Java es un lenguaje de programación de **Sun Microsystems**, que fue concebido bajo la dirección de James Gosling y Bill Joy. Es un lenguaje de propósito general, de alto nivel, y orientado a objetos puro, en el sentido de que no hay ninguna variable, función o constante que no esté dentro de una clase.

Los tipos de programas más comunes que se pueden hacer con Java son los *applets* (se ejecutan en el navegador de la máquina cliente) y las aplicaciones (programas que se ejecutan directamente en la JVM). Otro tipo especial de programa se denomina *servlet* que es similar a los *applets* pero se ejecutan en los servidores Java.

El lenguaje Java es a la vez compilado e interpretado. Con el compilador se convierte el código fuente que reside en archivos cuya extensión es **.java,** a un conjunto de instrucciones que recibe el nombre de *bytecodes* que se guardan en un archivo cuya extensión es **.class**. Estas instrucciones son independientes del tipo de ordenador. El intérprete ejecuta cada una de estas instrucciones en un ordenador específico (Windows, Macintosh, etcétera). Solamente es necesario, por tanto, compilar una vez el programa, pero se interpreta cada vez que se ejecuta en un ordenador. Esto se ilustra en la Figura 1. Actualmente, los intérpretes pueden ser remplazados por la **generación de código justo en el momento** (*Just in Time Code Generation*), como se explicará más adelante.

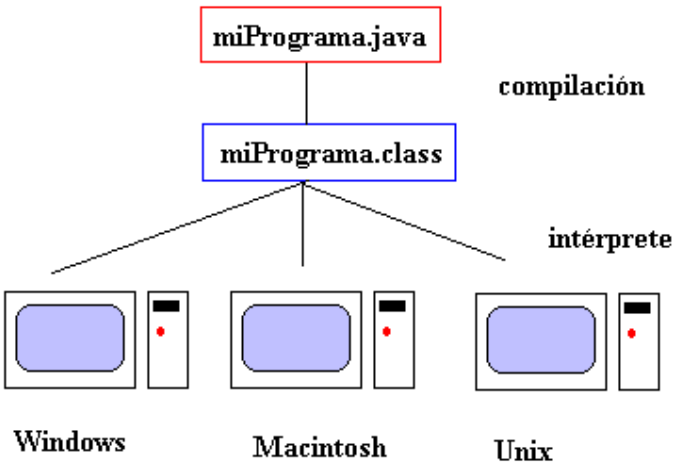

Figura 1. Java. Compilación – Interpretación.

El código que generan los compiladores del lenguaje Java no es específico de una máquina física en particular, sino de una máquina virtual. Aún cuando existen múltiples implantaciones de la Máquina Virtual Java, cada una específica de la plataforma sobre la cual subyace, existe una única especificación de la máquina virtual, que proporciona una vista independiente del hardware y del sistema operativo sobre el que se esté trabajando.

Se dice que el código Java es **portable**, debido a que es posible ejecutar el mismo archivo de clase (*.class*), sobre una amplia variedad de arquitecturas de hardware y de software, sin ninguna modificación.

Java es un lenguaje **dinámico**, debido a que las clases son cargadas en el momento en que son necesitadas (dinámicamente), ya sea del sistema de archivos local o desde algún sitio de la red mediante algún protocolo *URL*.

Java tiene la capacidad de aumentar su sistema de tipos de datos dinámicamente o en tiempo de ejecución. Este "enlace tardío" (*late-binding*) significa que los programas sólo crecen al tamaño estrictamente necesario, aumentando así la **eficiencia** del uso de los recursos.

Debido a que Java nació en la era post-Internet, fue diseñado con la idea de la **seguridad** y la fiabilidad, por lo que se le integraron varias capas de seguridad para evitar que programas maliciosos pudiesen causar daños en los sistemas, sobre los que se ejecuta la implantación de la Máquina Virtual Java.

2.2 La Máquina Virtual Java

La **Máquina Virtual Java** es el entorno en el que se ejecutan los programas Java, su misión principal es la de garantizar la portabilidad de las aplicaciones Java. Define esencialmente un ordenador abstracto y especifica las instrucciones (*bytecodes*) que este ordenador puede ejecutar. El intérprete Java específico ejecuta las instrucciones que se guardan en los archivos cuya extensión es **.class**. Las tareas principales de la MVJ son las siguientes:

- Reservar espacio en memoria para los objetos creados
- Liberar la memoria no usada (garbage collection)
- Asignar variables a registros y pilas
- Llamar al sistema huésped para ciertas funciones, como los accesos a los dispositivos
- Vigilar el cumplimiento de las normas de seguridad de las aplicaciones Java

Siempre que se corre un programa Java, las instrucciones que lo componen no son ejecutadas directamente por el hardware sobre el que subyace, sino que son pasadas a un elemento de software intermedio, que es el encargado de que las instrucciones sean ejecutadas por el hardware. Es decir, el código Java no se ejecuta directamente sobre un procesador físico, sino sobre un **procesador virtual Java**, que es, precisamente, el software intermedio que se mencionó.

En la Figura 2 puede observarse la capa de software que implementa a la máquina virtual Java. Esta capa de software oculta los detalles inherentes a la plataforma, a las aplicaciones Java que se ejecuten sobre ella. Debido a que la plataforma Java fue diseñada pensando en que se implementaría sobre una amplia gama de sistemas operativos y de procesadores, se incluyeron dos capas de software para aumentar su portabilidad. La primera, dependiente de la plataforma, es llamada **adaptador**, mientras que la segunda, que es independiente de la plataforma, se le llama **interfaz de portabilidad**. De esta manera, la única parte que se tiene que escribir para una plataforma nueva, es el adaptador. El sistema operativo proporciona los servicios de manejo de ventanas, red, sistema de archivos, etcétera.

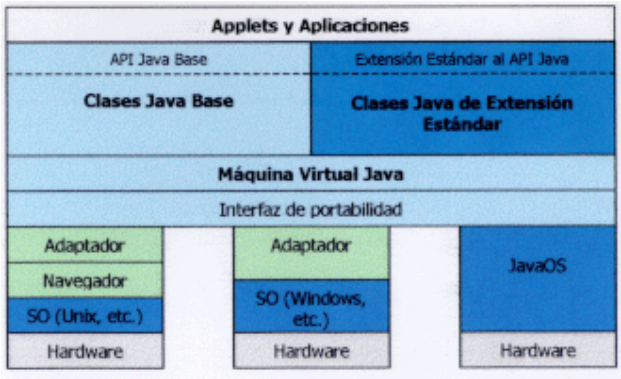

Figura 2. La Máquina Virtual Implementada para una variedad de plataformas.

2.3 La Plataforma Java (Sistema en Tiempo de Ejecución)

Sun utiliza el término **Máquina Virtual Java**, para referirse a la especificación abstracta de una máquina de software para ejecutar programas Java. La especificación de esta máquina virtual, define elementos como el formato de los archivos de clases de Java (*.class*), así como la semántica de cada una de las instrucciones que componen el conjunto de instrucciones de la máquina virtual. A las implantaciones de esta especificación se les conocen como **Sistemas en Tiempo de Ejecución Java**.

Ejemplos de Sistemas en Tiempo de Ejecución son el Navegador de Nestcape, el Explorador de Microsoft y el programa Java (incluido en el JDK – Java Development Kit).

En la Figura 3 se pueden observar los componentes típicos de un sistema en tiempo de ejecución.

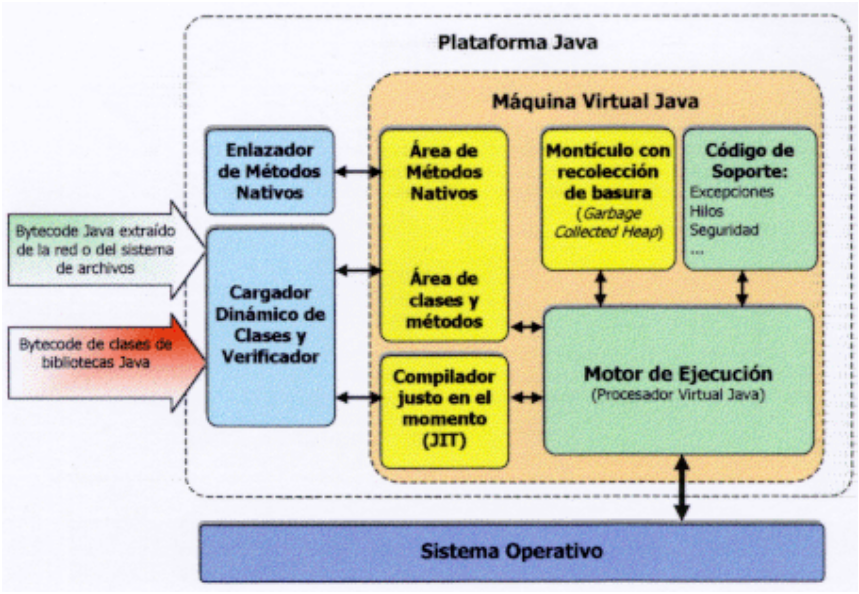

Figura 3. Arquitectura del Sistema de Tiempo de Ejecución Java.

- **Motor de Ejecución:** el procesador virtual que se encarga de ejecutar el código (*bytecode*), generado por algún compilador de Java.

- **Manejador de Memoria:** encargado de obtener memoria para las nuevas instancias de objetos, arreglos, etcétera, y realizar tareas de recolección de basura.

- **Manejador de Errores y Excepciones:** encargado de generar, lanzar y atrapar excepciones.

- **Soporte de Métodos Nativos:** encargado de llamar métodos de C++ o funciones de C, desde métodos Java y viceversa.

- **Interfaz Multi-hilos:** encargada de proporcionar el soporte para hilos y monitores.

- **Cargador de Clases:** su función es cargar dinámicamente las clases Java a partir de los archivos de clase (.class).

- **Administrador de Seguridad:** se encarga de asegurar que las clases cargadas sean seguras, así como controlar el acceso a los recursos del sistema.

Adicionalmente, existe un conjunto de clases Java estándar, fuertemente ligadas a la implantación de cada MVJ en particular. Ejemplos de esto los tenemos en las clases encargadas de funciones como los accesos a los recursos de la red, manejar el sistema de ventanas, los hilos y el sistema de archivos local. Todos estos elementos en conjunto actúan como una interfaz de alto nivel, para acceder a los recursos del sistema operativo. Es esta interfaz la clave de la portabilidad de los programas Java, debido a que independientemente del hardware o sistema operativo sobre el que se esté trabajando, la máquina virtual Java oculta todas estas diferencias.

A continuación se describen con mayor detalle los componentes de un sistema en tiempo de ejecución.

2.3.1 Motor de Ejecución

Es la entidad de hardware o software, que ejecuta las instrucciones contenidas en los códigos de operación (*bytecodes*) que implementan los métodos Java. En las versiones iniciales de Sun, el motor de ejecución consistía de un intérprete de códigos de operación. En las versiones más avanzadas, se utiliza la tecnología de **generación de código justo en el momento** (*Just-in-Time code generation*), en donde las instrucciones que implementan a los métodos, se convierten en código nativo que se ejecuta directamente en la máquina sobre la que se subyace. El código nativo se genera únicamente la primera vez que se ejecuta el código de operación Java, por lo que se logra un aumento considerable en el rendimiento de los programas.

2.3.2 El Conjunto de Instrucciones del Procesador Virtual

Muchas de las instrucciones del procesador virtual Java, son muy similares a las que se pueden encontrar para los procesadores comunes y corrientes, como los Intel, es decir, incluyen los grupos de instrucciones típicos como los aritméticos, los de control de flujo, de acceso a memoria, a la pila, etcétera.

Una de las características más significativas del conjunto de instrucciones del procesador virtual Java, es que están basadas en pila y utilizan "posiciones de memoria" numeradas, en lugar de registros. Esto es hasta cierto punto lógico, debido a que la máquina virtual está pensada para

correr sobre sistemas con procesadores sustancialmente diferentes. Es difícil hacer suposiciones sobre el número o tipo de registros que estos pudiesen tener. Esta característica de estar basada en operaciones sobre pila, impone una desventaja a los programas escritos en Java, contra los lenguajes completamente compilados, como C o C++, debido a que los compiladores de estos pueden generar código optimizado para la plataforma particular sobre la que se esté trabajando, haciendo uso de los registros, etcétera.

Varias de las instrucciones que componen el conjunto de instrucciones del procesador virtual de Java, son bastante más complejas que las que se pueden encontrar en procesadores comunes. Ejemplo de ello, se tienen las casi 20 instrucciones para realizar operaciones, tales como invocar métodos de objetos, obtener y establecer sus propiedades o generar y referenciar nuevos objetos. Es evidente que operaciones de este estilo son de una complejidad considerable y la proyección a sus respectivas instrucciones, sobre el conjunto de instrucciones del procesador de la máquina, implicará a varias decenas de esas instrucciones.

2.3.3 El Verificador de Java

Una de las principales razones para utilizar una máquina virtual, es agregar elementos de seguridad a un sistema. Si un intérprete falla o se comporta de manera aleatoria, debido a código mal formado, es un problema muy serio. La solución trivial a este problema sería incluir código encargado de capturar errores y verificar que el código sea correcto. Es evidente que la principal desventaja de esta solución, es que se vuelve a caer en un sistema sumamente seguro, pero altamente ineficiente.

Los diseñadores de Java tomaron otro camino. Cuando estaban diseñando el conjunto de instrucciones para la máquina virtual de Java, tenían dos metas en mente. La primera era que el conjunto de instrucciones fuera similar a las instrucciones que se pueden encontrar en los procesadores reales. La segunda era construir un conjunto de instrucciones que fuera fácilmente verificable.

En Java, justo después de que se obtiene una clase del sistema de archivos o de Internet, la máquina virtual puede ejecutar un verificador que se encargue precisamente de constatar que la estructura del archivo de clase es correcta. El verificador se asegura que el archivo tenga el número mágico (0xCAFEBABE) y que todos los registros que contiene el archivo tengan la longitud correcta, pero aún más importante, comprueba que todos los códigos de operación sean seguros de ejecutar. Es importante notar que Java no necesita que el verificador se ejecute sobre el archivo de clase, sino que es activado por el sistema en tiempo de ejecución y sólo sobre clases que el mismo sistema decida. Por lo común, las clases verificadas son las provenientes de Internet.

2.3.4 Administrador de Memoria

Java utiliza un modelo de memoria conocido como **administración automática del almacenamiento** (*automatic storage management*), en el que el sistema en tiempo de ejecución de Java mantiene un seguimiento de los objetos. En el momento que no están siendo referenciados por alguien, automáticamente se libera la memoria asociada con ellos.

Existen muchas maneras de implementar recolectores de basura, entre ellas tenemos:

- **Contabilizar referencias:** la máquina virtual Java asocia un contador a cada instancia de un objeto, donde se refleja el número de referencias hacia él. Cuando este contador es 0, la memoria asociada al objeto es susceptible de ser liberada. Aún cuando este

algoritmo es muy sencillo y de bajo costo (en términos computacionales), presenta problemas con estructuras de datos circulares.

- **Marcar e intercambiar (Mark-and-Sweep):** este es el esquema más común para implementar el manejo de almacenamiento automático. Consiste en almacenar los objetos en un montículo (heap) de un tamaño considerable y marcar periódicamente (generalmente mediante un bit en un campo que se utiliza para este fin) los objetos que no tengan ninguna referencia hacia ellos. Adicionalmente existe un montículo alterno, donde los objetos que no han sido marcados, son movidos periódicamente. Una vez en el montículo alterno, el recolector de basura se encarga de actualizar las referencias de los objetos a sus nuevas localidades. De esta manera se genera un nuevo montículo, que contiene únicamente objetos que están siendo utilizados.

Existen otros algoritmos para implementar sistemas que cuenten con recolección de basura.

2.3.5 Administrador de Errores y Excepciones

Las excepciones son la manera como Java indica que ha ocurrido algo "extraño" durante la ejecución de un programa Java. Comúnmente las excepciones son generadas y lanzadas por el sistema, cuando uno de estos eventos ocurre. De la misma manera, los métodos tienen la capacidad de lanzar excepciones, utilizando la instrucción de la MVJ, *athrow*.

Todas las excepciones en Java son instancias de la clase *java.lang.Throwable* o de alguna otra que la especialice. Las clases *java.lang.Exception* y *java.lang.Error*, heredan directamente de java.lang.Throwable. La primera se utiliza para mostrar eventos, de los cuales es posible recuperarse, como la lectura del fin de archivo o la falla de la red, mientras que la segunda se utiliza para indicar situaciones de las cuales no es posible recuperarse, como un acceso indebido a la memoria.

Cuando se genera una excepción, el sistema de tiempo de ejecución de Java, y en particular el manejador (*handler*) de errores y excepciones, busca un manejador para esa excepción, comenzando por el método que la originó y después hacia abajo en la pila de llamadas. Cuando se encuentra un manejador, éste atrapa la excepción y se ejecuta el código asociado con dicho manejador. Lo que ocurre después depende del código del manejador, pero en general, puede suceder que:

- Se utilice un goto para continuar con la ejecución del método original
- Se utilice un return para salir del método
- Se utilice athrow para lanzar otra excepción

En el caso que no se encuentre un manejador para alguna excepción previamente lanzada, se ejecuta el manejador del sistema, cuya acción típica es imprimir un mensaje de error y terminar la ejecución del programa.

Como ya se mencionó, para generar una excepción se utiliza la instrucción *athrow*, que toma un elemento de la pila. Dicho elemento debe ser la referencia a un objeto que herede de la clase *java.lang.Throwable*.

2.3.6 Soporte para Métodos Nativos

Las clases en Java pueden contener métodos que no estén implementados por códigos de operación (*bytecode*) Java, sino por algún otro lenguaje compilado en código nativo y

almacenado en bibliotecas de enlace dinámico, como las DLL de Windows o las bibliotecas compartidas SO de Solaris.

El sistema de tiempo de ejecución incluye el código necesario para cargar dinámicamente y ejecutar el código nativo que implementa estos métodos. Una vez que se enlaza el módulo que contiene el código que implementa dicho método, el procesador virtual atrapa las llamadas a éste y se encarga de invocarlo. Este proceso incluye la modificación de los argumentos de la llamada, para adecuarlos al formato que requiere el código nativo, así como transferirle el control de la ejecución. Cuando el código nativo termina, el módulo de soporte para métodos nativos se encarga de recuperar los resultados y de adecuarlos al formato de la máquina virtual Java.

De manera análoga, el módulo de soporte para código nativo se encarga de canalizar una llamada a un método escrito en Java, hecha desde un procedimiento o método nativo.

2.3.7 Interfaz de Hilos

Java es un lenguaje que permite la ejecución concurrente de varios hilos de ejecución, es decir, el sistema de tiempo de ejecución de Java tiene la posibilidad de crear más de un procesador virtual Java, donde ejecutar diferentes flujos de instrucciones, cada uno con su propia pila y su propio estado local. Los procesadores virtuales pueden ser simulados por software o implementados mediante llamadas al sistema operativo, sobre el cual subyace.

En el conjunto de instrucciones de la máquina virtual Java, sólo existen dos directamente relacionadas con los hilos, *monitorenter* y *monitorexit*, que sirven para definir secciones de código, que deben ejecutarse en exclusión mutua. El resto del soporte de los hilos se realiza atrapando llamadas a los métodos pertenecientes a la clase java.lang.Thread.

2.3.8 Cargador de Clases

Los programas Java están completamente estructurados en clases. Por lo tanto, una función muy importante del sistema en tiempo de ejecución, es cargar, enlazar e inicializar clases dinámicamente, de forma que sea posible instalar componentes de software en tiempo de ejecución. El proceso de cargado de las clases se realiza sobre demanda, hasta el último momento posible.

La Máquina Virtual Java utiliza dos mecanismos para cargar las clases. El primero consiste en un cargador de clases del sistema, cuya función es cargar todas las clases estándar de Java, así como la clase cuyo nombre es estrada vía la línea de comandos. De manera adicional, existe un segundo mecanismo para cargar clases dentro del sistema, utilizando una instancia de la clase *java.lang.ClassLoader* o alguna otra definida por el usuario, que especialice a la anterior.

Los cargadores especializados por los programadores, pueden definir la localización remota de donde se cargarán las clases o asignar atributos de seguridad apropiados para sus aplicaciones particulares. Finalmente, se puede usar a los cargadores para proporcionar espacios de nombres separados a diferentes componentes de una aplicación.

2.3.9 Arquitectura de Seguridad en Java

Java utiliza una serie de mecanismos de seguridad, con el fin de dificultar la escritura de programas maliciosos que pudiesen afectar la integridad de las aplicaciones y los datos de los

usuarios. Cada sistema en tiempo de ejecución Java tiene la capacidad de definir sus propias políticas de seguridad, mediante la implantación de un "administrador de seguridad" (*security manager*), cuya función es proteger al sistema de tiempo de ejecución, definiendo el ámbito de cada programa Java en cuanto a las capacidades de acceder a ciertos recursos, etcétera.

El modelo de seguridad original proporcionado por la plataforma Java, es conocido como la "caja de arena" (*sandbox*), que consiste en proporcionar un ambiente de ejecución muy restrictivo para código no confiable que haya sido obtenido de la red. Como se muestra en la Figura 4, la esencia del modelo de la caja de arena, es que el código obtenido del sistema de archivo local es por naturaleza confiable. Se le permite el acceso a los recursos del sistema, como el mismo sistema de archivos o los puertos de comunicación. Mientras, el código obtenido de la red se considera no confiable. Por lo tanto, tiene acceso únicamente a los recursos que se encuentran accesibles desde la caja de arena.

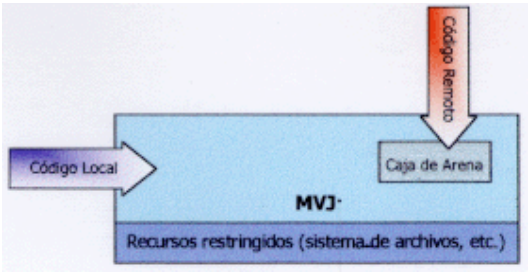

Figura 4. Modelo de seguridad del JDK 1.0.

Como se ha mencionado, la máquina implementa otros mecanismos de seguridad, desde el nivel de lenguaje de programación, como la verificación estricta de tipos de datos, manejo automático de la memoria, recolección automática de basura, verificación de los límites de las cadenas y arreglos, etcétera. Todo con el fin de obtener, de una manera relativamente fácil, código seguro.

En segunda instancia, los compiladores y los verificadores de código intentan asegurar que sólo se ejecuten códigos de ejecución (*bytecodes*) Java, con la estructura correcta y no maliciosos.

Finalmente, el acceso a los recursos importantes del sistema, es administrado entre el sistema de tiempo de ejecución y el administrador de seguridad (*Security Manager*), que es implementado por la clase *java.lang.SecurityManager*, que permite a las implantaciones incorporar políticas de seguridad. De esta manera, es posible para las aplicaciones determinar si una operación es insegura o contraviene las políticas de seguridad, antes de ejecutarla.

El JDK 1.1 introduce el concepto de "*applet* firmado" (*signed applet*), en el que los *applets* que poseen una firma digital correcta, son considerados como confiables. Por lo tanto, reciben los mismos privilegios que el código obtenido del sistema de archivos. Los *applets* firmados, junto con la firma, se envían en un archivo de formato JAR (*Java Archive*). En este modelo, los applets sin firma, continúan corriendo en la caja de arena. En la Figura 5 se puede observar el modelo de seguridad del JDK 1.1.

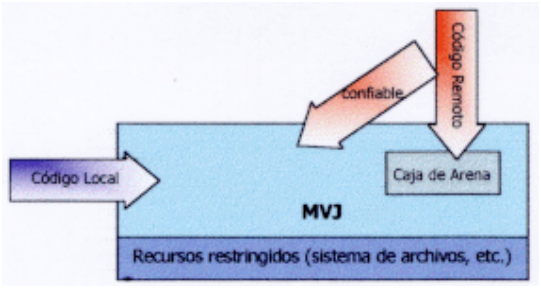

Figura 5. Modelo de seguridad del JDK 1.1.

Finalmente, como se muestra en la Figura 6, en la arquitectura de la plataforma de seguridad de Java 2 se introdujeron diferentes niveles de restricción y se eliminó la idea de que el código proveniente del sistema de archivo local siempre es confiable.

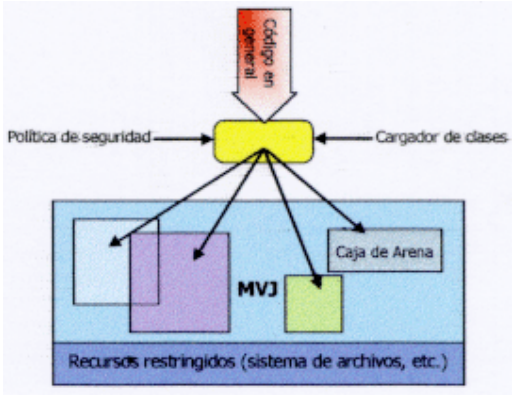

Figura 6. Modelo de seguridad de Java 2.

2.4 Arquitectura

La arquitectura de la JVM se basa en el concepto de una implementación que no es específica de una máquina. Esto es, la arquitectura misma no asume nada acerca de la máquina o de las características físicas y de construcción sobre la cual es implementada. De esta manera, la JVM es una entidad autónoma y única que ejecuta los archivos de clases. La JVM se separa en cinco unidades de funcionalidad distintas, las que se dedican a la tarea de ejecutar los archivos de clases:

- registros
- pila
- montículo de colección de desperdicios
- área de almacenamiento de métodos
- conjunto de instrucciones de la JVM

2.4.1 Registros

Para manejar la habilidad de implementar el trabajo en arquitecturas basadas en registros, la JVM define y utiliza los siguientes que aparecen en la Tabla 1.

Registro	Tamaño	Significado
pc	32 bits	Contador de programa, lleva la secuencia de la ejecución del programa (similar al homónimo de las arquitecturas reales)
optop	32 bits	Mantiene una referencia de memoria al tope de la pila
frame	32 bits	Provee un puntero al marco actual de pila
vars	32 bits	Provee el valor de desplazamiento para variables locales en relación al puntero a la pila

Tabla 1. Registros.

2.4.2 Pila

La arquitectura trabaja en torno al recurso de pila de 32 bits. La diferencia fundamental del trabajo de este tipo de pila que usa el esquema FIFO de recuperación y almacenamiento es que ésta es una pila particionada en tres regiones separadas.

- **Región de variable local:** provee el acceso a las variables locales (en conjunto al registro vars) para acceder a este tipo de elementos. Si las variables ocupan 64 bits, se manejan dos áreas de esta región.

- **Región de ambiente de ejecución:** usada para proveer código de operación (opcode) para mantener los métodos del marco de pila. También mantiene punteros a variables locales, el marco de pila previo y el inicio y final de la región de operandos.

- **Región de operandos:** contiene los operandos para el método en ejecución.

2.4.3 Montículo de Colección de Desperdicios

Permite una reutilización de los recursos que se han ido utilizando, sobre todo las áreas de memoria, lo que permite hacer las colocaciones de memoria y la liberación de ésta.

2.4.4 Área de Almacenamiento de Métodos

Como su nombre lo indica ésta área es el dispositivo de almacenamiento de memoria primaria para todos los métodos en ejecución de las distintas clases del programa Java ejecutable. Equivalente a la memoria RAM.

2.4.5 Conjunto de Instrucciones de la JVM

Esta no es una región ni un área, sino que corresponde a las instrucciones que maneja la JVM. Como en las arquitecturas reales, el Conjunto de Instrucciones indica al procesador cómo realizar una acción. Esto se ha explicado previamente.

2.5 Deficiencias de la MVJ

- Conjunto de instrucciones no ortogonal. Un lenguaje de programación es ortogonal, si tiene el mismo número de instrucciones asociadas a cada uno de los tipos de datos.

- Difícil de extender. Debido a que se utiliza un byte para codificar el código de operación de las instrucciones del procesador virtual Java (de ahí el nombre de bytecode), es difícil agregar nuevas instrucciones.

- No posee un árbol de análisis sintáctico. El código intermedio, usado en la MVJ, es simple y plano, es decir no incluye información acerca de la estructura del método original. En un lenguaje completamente compilado, esta información juega un papel muy importante en la optimización, debido a que permite trabajar con el control de dependencias y de flujo.

3. El Entorno .NET

La Figura 7 muestra la arquitectura básica del entorno .NET.

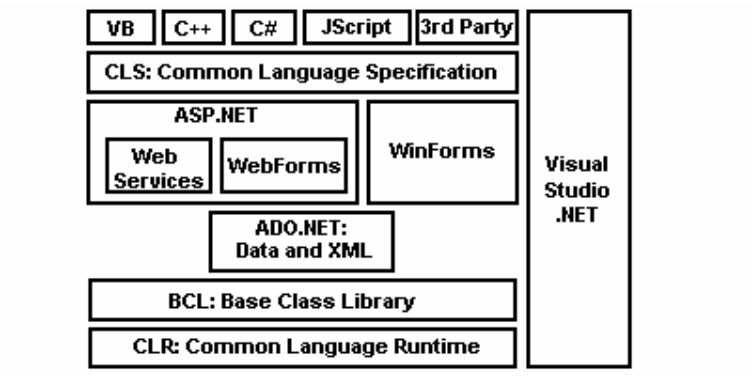

Figura 7: Microsoft .NET Framework

Estos componentes principales del Entorno .NET se describen a continuación:

- **El Common Language Runtime (CLR):** es el mecanismo de ejecución para las aplicaciones .NET. Proporciona varios servicios, incluyendo la carga y ejecución del código, aislamiento de la memoria de las aplicaciones, administración de memoria, manejo de excepciones, acceso a *metadata* (información mejorada de tipos), y la conversión de MSIL (Lenguaje Intermedio Microsoft) a código nativo.

- **La Base Class Library (BCL):** proporciona un amplio conjunto de clases, lógicamente agrupadas en espacios de nombres jerárquicos que proporcionan acceso a las características más importantes del sistema operativo.

- **ADO.NET:** es una actualización evolutiva para la tecnología de acceso a datos ActiveX® Data Objects (ADO) con mejoras importantes destinadas a la naturaleza desconectada del Web.

- **ASP.NET**: es una versión avanzada de Active Server Pages (ASP) para el desarrollo de aplicaciones Web (utilizando Formas Web) y desarrollo de servicios Web.

- **La Common Language Specification (CLS):** es responsable de hacer que muchas de las tecnologías antes mencionadas estén disponibles para todos los lenguajes que soportan .NET Framework. CLS no es una tecnología, y no hay un código fuente para ella. Define un conjunto de reglas que proporcionan un contrato que rige la interoperabilidad entre los compiladores de lenguaje y las bibliotecas.

- **Win Forms:** modelo de programación y conjunto de controles que proporciona una arquitectura sólida para el desarrollo de aplicaciones basada en Windows.

- **Visual Studio.NET:** proporciona las herramientas que le permiten explotar las características del Framework para crear aplicaciones concretas.

3.1 El lenguaje común en tiempo de ejecución (CLR)

El lenguaje común en tiempo de ejecución, o CLR, es el motor de ejecución para las aplicaciones del framework .NET. El CLR puede considerarse como el núcleo de dicho framework, desempeñando el papel de una máquina virtual que se encarga de gestionar la ejecución del código y de proporcionar una serie de servicios a dicho código (el código escrito para ajustarse a los servicios del CLR se denomina código gestionado, mientras que el código que no utiliza el CLR se denomina código no gestionado). Entre los servicios proporcionados por el CLR a las aplicaciones .NET se encuentran los siguientes:

- Gestión del código, encargándose de la carga y ejecución del código MSIL (Microsoft Intermediate Languages).
- Aislamiento de la memoria de las aplicaciones, de forma que desde el código perteneciente a un determinado proceso no pueda accederse al código o datos pertenecientes a otro proceso.
- Verificación de la seguridad de los tipos, garantizando la robustez del código mediante la implementación de un Sistema de Tipos Común o CTS (*Common Type System*).
- Conversión del código MSIL al código nativo, utilizándose para ello técnicas de compilación "Just In Time" (JIT).
- Acceso a los metadatos, que contienen información sobre los tipos, y sus dependencias, definidos en el código.
- Gestión automática de la memoria, encargándose de gestionar las referencias de los objetos y de la tarea de recolección de basura.
- Asegurar la seguridad en los accesos del código a los recursos, la cual estará en función del nivel de confianza del que goce el código.
- Manejo de las excepciones, incluyendo las excepciones entre código escrito en diferentes lenguajes.
- Interoperabilidad con el código no gestionado.
- Soporte de servicios para los desarrolladores, tales como la depuración.

El CLR es el que posibilita la integración entre diferentes lenguajes, proporcionando a su vez una mejora en el rendimiento como consecuencia de los servicios que ofrece, tales como la gestión automática de la memoria. El CLR está formado principalmente por tres componentes:

- Un Sistema de Tipos Común o CTS, formado por un amplio conjunto de tipos y operaciones que se encuentran presentes en la mayoría de los lenguajes de programación.
- Un sistema de metadatos, que permite almacenar dichos metadatos junto con los tipos a los que se refieren en tiempo de compilación, así como obtenerlos en tiempo de ejecución.
- Un sistema de ejecución, que se encarga de ejecutar las aplicaciones del framework .NET, haciendo uso del sistema de información de metadatos para desarrollar los servicios tales como la gestión de la memoria.

La figura 8 muestra la relación existente entre los distintos elementos del CLR.:

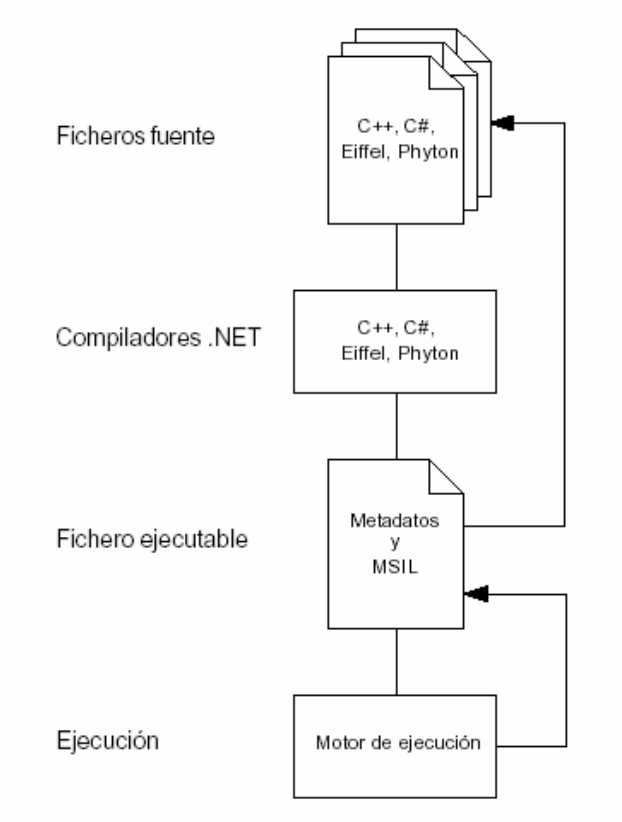

Figura 8. Relación entre elementos del CLR

3.1.1 Sistema de tipos común (CTS)

Para conseguir la interoperabilidad entre lenguajes es necesario adoptar un sistema de tipos común. Así, el sistema de tipos común (CTS) define como se declaran, utilizan y gestionan los tipos en el CLR. El CTS desarrolla las siguientes funciones:

- Establece un framework que permite la integración entre lenguajes, la seguridad de tipos, y la ejecución de código con un alto rendimiento.
- Proporciona un modelo orientado a objetos que soporta la implementación de muchos lenguajes de programación.
- Define una serie de reglas que los lenguajes deben seguir para permitir la interoperabilidad de los mismos.

3.1.1.1 Clasificación de tipos

El CTS se divide en dos categorías generales de tipos:

- Tipos Valor
- Tipos Referencia

La figura 9 muestra esta clasificación en detalle:

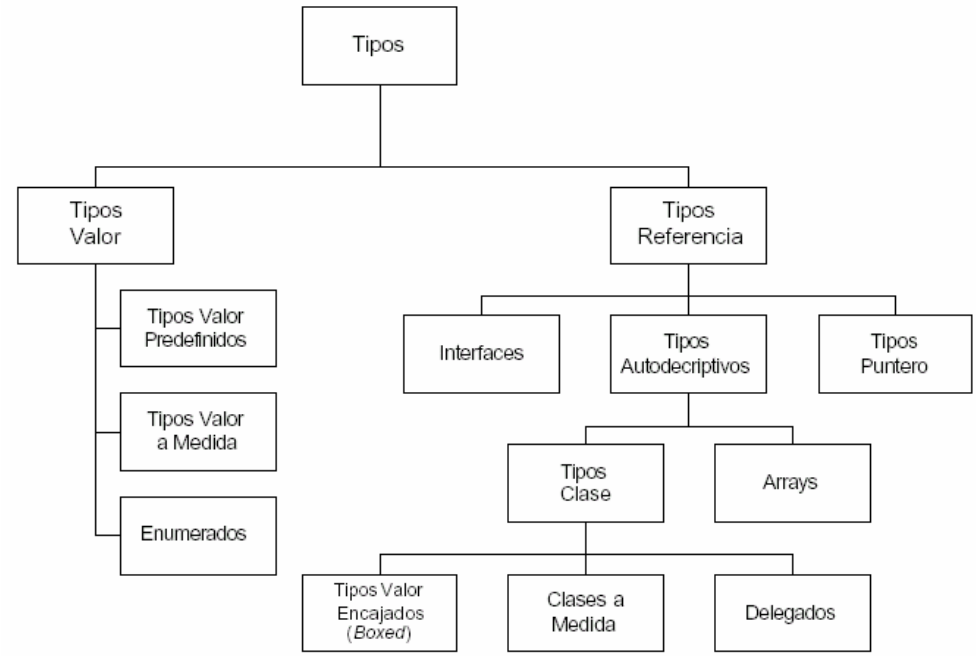

Figura 9. Clasificación de Tipos

3.1.1.2 Definición de tipos

Una definición de un tipo construye un nuevo tipo a partir de tipos existentes. Los tipos valor predefinidos, los punteros, arrays y delegados son definidos al ser utilizados, por lo que a estos tipos se les conoce como tipos implícitos. La definición de un tipo incluye los siguientes elementos:

- Los atributos definidos sobre el tipo.
- La visibilidad del tipo.
- Nombre del tipo.
- El tipo base del tipo definido.
- Las interfaces implementadas por el tipo.
- Las definiciones de cada uno de los miembros del tipo. Dentro de un tipo pueden definirse los siguientes miembros:
    - Eventos
    - Campos (variables).
    - Tipos anidados.
    - Métodos.
    - Propiedades (son mapeadas a métodos get y set).

3.1.1.3 Tipos Valor

Los tipos Valor suelen localizarse en la pila de ejecución, y tienen la propiedad de que no pueden ser extendidos mediante herencia. El framework soporta dos clases de tipos Valor:

- Tipos valor predefinidos: números enteros, de coma flotante, los booleanos, etc.

- Tipos valor definidos por el usuario: pueden poseer todos los elementos típicos en la definición de un tipo, es decir, métodos, campos, propiedades, eventos y tipos anidados.

Un aspecto a tener en cuenta derivado de la existencia de tipos Valor y tipos Referencia es la posibilidad de pasar de una clase de tipo a la otra. Así, todos los tipos Valor (predefinidos y definidos por el usuario) tienen su correspondiente tipo Referencia.

Enumerados: es una forma especial de un tipo Valor, y proporciona nombres alternativos para los valores del tipo primitivo entero (con signo o sin él).

3.1.1.4 Tipos Referencia

Los tipos referencia son la combinación de una localización, la cual les dota de identidad, y una secuencia de bits. Las localizaciones, que denotan las áreas de memoria en las cuales los valores pueden ser almacenados, poseen seguridad de tipos, de forma que sólo pueden asignarse tipos compatibles. A continuación se describen los distintos tipos Referencia del CTS.

Clases: Como en cualquier sistema orientado a objetos, el CTS incluye el concepto de clase. Una clase puede heredar como mucho de una única clase, y puede implementar cualquier número de interfaces.

Delegados: El CTS soporta un tipo de objetos denominados delegados, los cuales tienen una finalidad similar a los punteros a funciones de C++, pero con la diferencia en que estos cuentan con la seguridad del sistema de tipos, de forma que siempre apuntan a un objeto válido, no pudiendo corromper la memoria de otro objeto.

Arrays: Los arrays son definidos especificando el tipo de sus elementos, su número de dimensiones y sus límites inferior y superior para cada dimensión.

Interfaces: Un tipo interfaz es la especificación parcial de un tipo, actuando como contratos que ligan a los implementadores con lo especificado en la interfaz. Una interfaz puede contener métodos, campos estáticos, propiedades y eventos, diferenciándose de las clases y de los tipos Valor en que no puede contener campos (o variables) de instancia.

Punteros: El CTS soporta tres tipos de punteros: punteros gestionados, punteros no gestionados, y punteros no gestionados a funciones. Los punteros gestionados son generados al pasar los argumentos por referencia en la llamada a un método. El CTS proporciona dos operaciones que respetan la seguridad de los tipos (*type safe*) para los punteros: la carga de un valor referenciado por un puntero, y la escritura de un valor a una localización referenciada por un puntero.

3.1.2 Metadatos

Los metadatos son información binaria que describe los tipos implementados por un programa. Los metadatos se almacenan en un fichero Ejecutable Portable (PE) o en memoria, de forma que cuando un fichero con código es compilado, los metadatos son almacenados junto con el código MSIL, de forma que todos los compiladores para .NET están obligados a emitir metadatos sobre cada tipo contenido en un fichero fuente.

Los metadatos son el puente que enlaza el sistema de tipos común (CTS) y el motor de ejecución del .NET.

Los componentes .NET almacenan el código MSIL junto con los metadatos, constituyendo así unas unidades autodescriptivas denominadas ensamblados.

3.1.2.1 Atributos

Un atributo es un objeto que representa datos que están asociados a elementos de un programa, tales como tipos, métodos, propiedades, etc. Los atributos son un mecanismo de extensión de los metadatos de un programa, almacenándose dichos atributos con los metadatos del elemento al cual están asociados. Los lenguajes suelen contar con instrucciones que proporcionan información declarativa, tales como los modificadores "public" y "private", los cuales proporcionan información adicional sobre los miembros de una clase (en este caso sobre la visibilidad de los mismos); sin embargo, estos tipos de información declarativa suelen estar predefinidos en el lenguaje, y no pueden ser ampliados por los usuarios del lenguaje. Los atributos constituyen un mecanismo para ampliar estos tipos de información declarativa, los cuales suelen denominarse *aspectos*.

Los *aspectos* son propiedades que afectan a la semántica o comportamiento del sistema, representando decisiones de diseño que se encuentran entremezcladas entre los aspectos funcionales de un sistema (la lógica de la aplicación) y que son difíciles de encapsular en una unidad de código. Así, ejemplos de aspectos son las características no funcionales de una aplicación, tales como la seguridad, las transacciones, la concurrencia, la persistencia, las optimizaciones en la ejecución, etc. La plataforma .NET, con la introducción de los atributos, proporciona soporte a una técnica de programación denominada "Programación Orientada a Aspectos", que presenta la ventaja de poder describir con mayor facilidad ciertas características de una aplicación, tales como las propiedades no funcionales. Mediante la "Programación Orientada a Aspectos", el desarrollador le indica al CLR las características que necesita, bien en tiempo de programación o de configuración, mediante el uso de atributos, de forma que el CLR interceptará las llamadas a una clase, examinará sus atributos y proporcionara las características demandadas, eliminando la necesidad de escribir código específico para soportar dichas características.

Tipos de atributos: La plataforma .NET posee dos tipos de atributos: los intrínsecos o predefinidos, los cuales son proporcionados como parte del CLR, y los atributos definidos y creados por los usuarios.

Utilización de los atributos: Los atributos presentan una serie de características relativas a la aplicación de los mismos, las cuales se indican a continuación:

- Objetivo de un atributo. El objetivo de un atributo son los elementos del programa sobre los cuales puede aplicarse dicho atributo, siendo posible restringir los elementos sobre los cuales puede aplicarse un atributo.
- Heredabilidad de un atributo. Es posible marcar un atributo como heredable o como no heredable.
- Múltiples instancias de un atributo. Es posible indicar si múltiples instancias de un mismo atributo pueden ser aplicadas o no a un mismo elemento de un programa.

3.1.3 Sistema de ejecución

El motor de ejecución del CLR es el responsable de asegurar que el código es ejecutado como requiere, proporcionando una serie de facilidades para el código MSIL como:

- Carga del código y verificación.
- Gestión de las excepciones.
- Compilación "Just In Time" (JIT).
- Gestión de la memoria.
- Seguridad.

3.1.3.1 Lenguaje intermedio MSIL

El código intermedio MSIL generado por los compiladores del framework .NET es independiente del juego de instrucciones de una CPU específica, pudiendo ser convertido a código nativo de forma eficiente. El lenguaje MSIL es un lenguaje de un nivel de abstracción mucho mayor que el de la mayoría de los lenguajes máquina de las CPUs existentes, incluyendo instrucciones para trabajar directamente con objetos (crearlos, destruirlos, inicializarlos, llamar a métodos virtuales, etc.), instrucciones para el manejo de excepciones, de tablas, etc.

La principal ventaja del MSIL es que proporciona una capa de abstracción del hardware, lo que facilita la ejecución multiplataforma y la integración entre lenguajes. Otra ventaja que se deriva del uso de este lenguaje intermedio es la cuestión de la seguridad relativa a la verificación del código, pues el motor de ejecución puede examinar la intención del código independientemente del lenguaje de alto nivel utilizado para generarlo. Sin embargo, dado que las CPUs no pueden ejecutar directamente MSIL, es necesario convertirlo a código nativo de la CPU antes de ejecutarlo.

3.1.3.2 Compilación JIT

La traducción de MSIL a código nativo de la CPU es realizada por un compilador "Just In Time" o *jitter*, que va convirtiendo dinámicamente el código MSIL a ejecutar en código nativo según sea necesario. Este proceso se muestra en la figura 10:

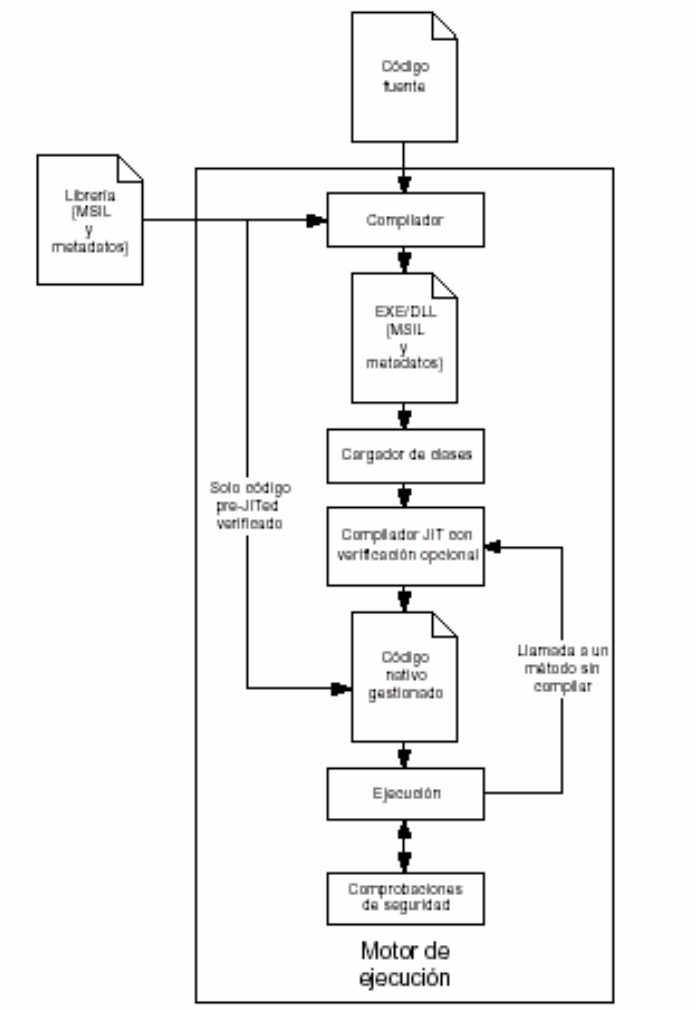

Figura 10. Traducción del MSIL a código nativo

La compilación JIT tiene en cuenta el hecho de que algunas porciones de código no serán llamadas durante la ejecución, por lo que en lugar de invertir tiempo y memoria en convertir todo el código MSIL a código nativo, únicamente convierte el código que es necesario durante la ejecución, almacenándolo por si fuera necesario en futuras llamadas. El cargador crea y liga un *stub* a cada uno de los métodos de un tipo cuando este es cargado, de forma que en la primera llamada a un método el *stub* pasa el control al compilador JIT, el cual traduce el código MSIL a código nativo y modifica el *stub* para que apunte al código nativo recién traducido, de forma que las siguientes llamadas ejecutarán directamente dicho código nativo. Durante la ejecución, el código gestionado recibe del sistema de ejecución servicios como la gestión automática de la memoria, seguridad, interoperabilidad con el código no gestionado, soporte para la depuración entre lenguajes, etc.

Como parte del proceso de compilación del código MSIL, debe pasarse un proceso de verificación (a menos que el administrador especifique una política de seguridad que indique lo contrario) que examine el código MSIL y los metadatos, y determine si el código respeta los tipos seguro (*type safe*) de forma que pueda tenerse la seguridad de que solo son permitidos los accesos a memoria seguros, permaneciendo los objetos aislados unos de otros. También es posible verificar si el código MSIL ha sido generado correctamente, pues el código MSIL incorrecto podría violar la condición del código tipo seguro.

El compilador JIT se distribuye en tres versiones:

- *Jitter* normal. Este compilador examina el código MSIL con el objetivo de optimizar el código nativo generado. Es el que se suele utilizar por defecto.
- *Jitter* económico. Funciona de forma similar al normal, pero no realiza ninguna optimización al compilar, limitándose a sustituir cada instrucción MSIL por la instrucción/es equivalentes en código nativo. Como consecuencia, el proceso de compilación se realiza de una forma mucho más rápida, y aún a pesar de producir un código nativo mucho menos eficiente, la ejecución sigue siendo mucho más rápida. Está pensado para ser utilizado en dispositivos empotrados que dispongan de poca potencia de CPU y poca memoria, como los dispositivos móviles.
- *Prejitter*. Se distribuye como una herramienta mediante la cual es posible compilar completamente cualquier ensamblado y convertirlo a código nativo, de forma que posteriores ejecuciones del mismo se harán usando la versión ya compilada, ahorrándose el tiempo de realizar la compilación dinámica.

Este esquema puede producir la impresión de que introduce mucha sobrecarga, sin embargo, se piensa que en el futuro el código gestionado se ejecutará más rápidamente que el no gestionado, pues cuando el jitter traduce el código MSIL a código nativo posee mucha mas información del entorno de ejecución que la que posee un compilador tradicional. Así, el jitter puede detectar, por ejemplo que la máquina sobre la que se encuentra es un Pentium III y generar instrucciones especiales para dicho procesador, mientras que un compilador tradicional siempre tendrá que limitarse a una máquina concreta, que por lo general tendrá prestaciones inferiores a las máquinas existentes en el mercado, pues la opción de producir múltiples versiones de una aplicación para las distintas versiones de una misma familia de procesadores parece poco factible.

3.1.3.3 Recolector de basura

El recolector de basura es el responsable de eliminar los objetos de la memoria *heap* que no van a ser referenciados nunca más, compactando el resto de objetos, y actualizando tras esto la referencia a la última posición de memoria libre del *heap*. El proceso de recolección de basura puede ser lanzado automáticamente por el CLR o por una aplicación que lo invoca explícitamente. Para averiguar qué objetos no van a ser referenciados nunca más, el recolector de basura comienza por obtener las referencias "raíces", que son aquellos objetos referenciados directamente por la aplicación. Una vez obtenidas dichas referencias "raíces", el recolector obtiene a su vez los objetos referenciados por cada referencia "raíz", y así sucesivamente hasta que obtiene el conjunto de todos los objetos válidos, tras lo cual el recolector de basura es libre de eliminar los objetos no válidos y compactar el *heap*.

3.1.3.4 Seguridad

Tradicionalmente, las arquitecturas de seguridad se han basado en proporcionar aislamiento y control de acceso basándose en cuentas de usuario del sistema operativo, asumiendo que todos los programas ejecutados por un usuario poseen el mismo nivel de confianza. Alternativamente, los programas que no gozan de total confianza son ejecutados en modo restringido, de forma que son aislados del entorno sin poder acceder a la mayoría de los servicios. Ambos modelos son demasiado extremistas, por lo que .NET proporciona un modelo de seguridad intermedio más refinado, en el cual los programas gozan de distintos niveles de confianza en función de una serie de factores.

El sistema de seguridad de .NET se basa en el código gestionado, de forma que las reglas de seguridad se aseguran por el CLR. La mayor parte del código gestionado es verificado para

asegurar la correspondencia de tipos y el comportamiento definido de otras propiedades. Este mecanismo de verificación también garantiza que el flujo de ejecución sea transferido únicamente a localizaciones bien definidas, como los puntos de entrada de los métodos, lo que elimina la posibilidad de saltar a una localización arbitraria. El proceso de verificación impide que el código que no respete los tipos sea ejecutado, y evita una serie de errores como el desbordamiento de los buffer, la lectura de una posición de memoria arbitraria, etc.

La plataforma .NET proporciona dos modelos de seguridad para las aplicaciones: la seguridad basada en la evidencia y la seguridad basada en roles.

Seguridad basada en la evidencia

Por evidencia se entiende una serie de informaciones sobre el código a ejecutar que son pasadas a la política de seguridad. Algunas de dichas informaciones son relativas al lugar, URL, zona (Internet, Intranet, local u otro criterio de seguridad por zonas), y nombre fuerte del ensamblado a ejecutar. Basándose en dichas informaciones la política de seguridad puede determinar el conjunto de permisos apropiado para dicho ensamblado. Dichas evidencias o informaciones pueden ser obtenidas de múltiples fuentes, como el CLR, el navegador, ASP.NET o la consola.

Seguridad basada en roles

En algunas ocasiones es apropiado que las decisiones de seguridad se tomen en base a la identidad o rol asociado con el contexto de ejecución, es decir, con la identidad o rol que realiza la llamada. El farmework .NET permite implantar este modelo de seguridad mediante los conceptos de identidades y principales (*principals*). Las identidades encapsulan la información sobre un usuario autenticado, mientras que un principal encapsula la identidad junto con la información relacionada del rol. El esquema utilizado para la autorización puede estar basado en las cuentas de usuario de Windows, o basado en un sistema genérico de nombres de usuario no relacionado con las cuentas del sistema operativo.

Aislamiento en el almacenamiento

El framework .NET proporciona una facilidad especial para almacenar datos, incluso cuando el acceso a los ficheros no se permite. Básicamente, a cada ensamblado se le proporciona un espacio aislado en el disco, de forma que únicamente puede acceder a dicho espacio. Este espacio puede ser utilizado para mantener ficheros de log, de configuración, etc.

Criptografía

El framework .NET proporciona un conjunto de objetos criptográficos que soportan el cifrado, la firma digital, técnicas de *hash*, y generación de números aleatorios, lo cual es implementado por medio de una serie de algoritmos conocidos como RSA, DSA, Triple DES, DES, MD5, SHA1, etc. En .NET, la criptografía es utilizada para firmar digitalmente los ensamblados, de forma que cualquier cambio que sufra el ensamblado podrá ser detectado. Así mismo, las librerías de .NET ofrecen soporte para la utilización de certificados digitales, los cuales pueden ser utilizados opcionalmente en los ensamblados.

3.2 Librería de Clases Base (BCL)

La librería de Clases Base del framework .NET es una colección reutilizable de clases, o tipos, que están altamente integrados con el framework, y que permiten acceder a los servicios ofrecidos por el CLR y a las funcionalidades más frecuentemente utilizadas a la hora de desarrollar aplicaciones. Esta librería de clases está construida sobre la naturaleza orientada a objetos del CLR, proporcionando una serie de tipos cuya funcionalidad puede ser extendida mediante la herencia. Al estar escrita la librería en MSIL, esta puede ser utilizada desde cualquier lenguaje cuyo compilador genere MSIL, de forma que a través de las clases suministradas es posible desarrollar cualquier tipo de aplicación, desde las tradicionales aplicaciones de ventanas, consola o servicio de Windows NT hasta servicios Web y aplicaciones ASP.NET.

Dada la amplitud de la BCL, ha sido necesario organizar las clases que contiene en esquemas lógicos de nombrado denominados espacios de nombres (*namespaces*), los cuales son utilizados para agrupar clases con funcionalidades similares y son totalmente independientes de los ensamblados que contienen a los tipos de la librería de clases.

3.3 Especificación del Lenguaje Común (CLS)

El CLR proporciona, mediante el sistema de tipos común CTS y los metadatos, la infraestructura necesaria para lograr la interoperabilidad entre lenguajes, pues todos los lenguajes siguen las reglas definidas en el CTS para la definición y el uso de los tipos, y los metadatos definen un mecanismo uniforme para el almacenamiento y recuperación de la información sobre dichos tipos. Pero a pesar de esto, no hay ninguna garantía de que la funcionalidad de los tipos escritos por un desarrollador en un lenguaje determinado pueda ser completamente utilizado por otros desarrolladores que utilizan otros lenguajes, pues cada lenguaje de programación utiliza los elementos del CTS y de los metadatos que necesita para soportar su propio conjunto de características, pudiendo existir características del CLR que no son soportadas por un lenguaje concreto.

Para asegurar que el código escrito en un lenguaje sea accesible desde otros lenguajes se ha definido la Especificación del Lenguaje Común o CLS (Common Language Specification), que establece el conjunto mínimo de características que deben soportarse para asegurar la interoperabilidad, siendo dicho conjunto de características mínimas un subconjunto del CTS. El CLS ha sido diseñado para ser lo suficientemente grande como para que incluya las construcciones que son utilizadas comúnmente en los lenguajes, y lo suficientemente pequeño para que la mayoría de los lenguajes puedan cumplirlo.

Así, para que un objeto pueda interactuar con otros objetos, independientemente del lenguaje en el que hayan sido implementados, estos objetos deben exponer únicamente las características que están incluidas en el CLS. Los componentes que están adheridos a las reglas del CLS y utilizan sólo las características incluidas en el CLS se denominan como componentes conformes con CLS (*CLS-compliant*). Sin embargo, el concepto de conforme con CLS tiene un significado más específico dependiendo de si trata de código conforme con CLS o de herramientas de desarrollo conformes con CLS.

3.3.1 Código conforme con CLS

Para que el código de un programa sea conforme con CLS, este debe exponer su funcionalidad mediante características incluidas en el CLS en los siguientes lugares:

- Definiciones de clases públicas.
- Definiciones de los miembros públicos de clases públicas, o de miembros accesibles mediante herencia.
- Parámetros y tipos de retorno de los métodos públicos pertenecientes a clases públicas, o de los métodos accesibles mediante herencia.

3.3.2 Herramientas conformes con CLS

Los niveles de conformidad CLS para los compiladores y otras herramientas de desarrollo son los siguientes:

- Conformidad CLS a nivel de consumo. Las herramientas que cumplen este nivel son capaces de acceder a todas las características proporcionadas por librerías conformes con CLS. Sin embargo, con herramientas que solo cumplan este nivel de conformidad no es posible construir código que extienda a los tipos contenidos en librerías conformes con CLS.
- Conformidad CLS a nivel de extensión. Las herramientas que cumplen este nivel son capaces de utilizar y extender las características proporcionadas por librerías conformes con CLS. Las herramientas que se sitúan en este nivel siguen las reglas del nivel de consumo, más un conjunto de reglas adicionales.

3.4 Lenguajes

Todos los lenguajes .NET emplean el .NET Common Language Runtime (CLR) y las Base Class Library (BCL), aplicando un conjunto común de características mínimas definidas por el Common Language Specification (CLS). Asimismo, cada lenguaje soporta el Common Type System (CTS) proporcionando compatibilidad a través de lenguajes. Mientras ciertos lenguajes soportan nativamente el CTS, otros tales como C++ administrado proporcionan soporte a través de extensiones para el lenguaje base.

3.4.1 Opción de lenguaje

Dentro de CLR, todos los lenguajes comparten un gran conjunto de recursos entre los que se incluyen:

- Un modelo de programación orientado a objetos (herencia, polimorfismo, manejo de excepciones y colección de basura)
- Modelo de seguridad
- Sistema de tipos
- Base Class Library (BCL) (Biblioteca de Clases Base)
- Desarrollo, depuramiento y herramientas de perfilamiento
- Administración de ejecución y código
- Traductores y optimizadores de MSIL a código nativo

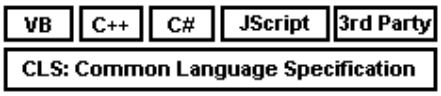

Figura. 11: CLS y Lenguajes

Este alto grado de interoperabilidad de lenguajes afecta fundamentalmente la manera en que elige los lenguajes de programación e implementa los diseños. La opción del lenguaje se vuelve más una preferencia personal y depende de la sintaxis que crea más adecuada. El CLR también puede canalizar una aceptación más amplia de lenguajes especiales, debido a que tendrán las mismas capacidades que los lenguajes más conocidos.

3.4.2 C# (C Sharp)

C# es un moderno lenguaje orientado a objetos que permite que los programadores desarrollen rápidamente una amplia gama de aplicaciones para la nueva plataforma Microsoft .NET. Debido a su elegante diseño orientado a objetos, C# es una fabulosa opción para crear una amplia gama de componentes, desde objetos de negocios de alto nivel hasta aplicaciones de nivel del sistema. Debido a que están soportados por la plataforma .NET, estos componentes se pueden convertir en servicios Web, permitiéndoles ser invocados a través de Internet, desde cualquier lenguaje que se ejecute en cualquier sistema operativo.

- Productividad y seguridad. El lenguaje está diseñado para ayudar a los desarrolladores a lograr más con menos líneas de código y menos oportunidades de error. Una gran característica de C# es que está diseñado para eliminar costosos errores de programación. El moderno diseño de C# elimina los errores de programación comunes. Por ejemplo:
    - La colección de basura libera al programador de tener que administrar memoria manualmente.
    - Las variables en C# son inicializadas automáticamente por el ambiente.
    - Están prohibidos los *casts* inseguros.

  El resultado final es un lenguaje que hace mucho más sencillo a los desarrolladores el escribir y mantener programas que resuelven problemas de negocios complejos. También está diseñado para reducir los costos de desarrollo continuos mediante el soporte integrado para versionamiento. La actualización de componentes de software es una tarea que lleva a errores

- Poder, expresividad y flexibilidad. C# proporciona mejor correlación entre los procesos de negocios y la implementación. Con el alto nivel de esfuerzo que las corporaciones gastan en la planeación de los negocios, es imperativo tener una conexión cercana entre el proceso de negocios abstracto y la actual implementación en software.

- Interoperabilidad. Los programas C# pueden utilizar objetos existentes, sin importar el lenguaje utilizado para crearlos.

- Acceso a la plataforma. Para aquellos desarrolladores que lo requieren, C# incluye una característica especial que permite a un programa invocar cualquier API nativo de C. Dentro de un bloque de código especialmente marcado, los desarrolladores pueden

utilizar punteros y características C/C++ tradicionales tales como administración manual de memoria y aritmética de punteros.

- Un real lenguaje de desarrollo basado en componentes.

4. Conclusiones

A partir de lo expuesto anteriormente, se puede concluir que los conceptos de Máquina Virtual, Lenguaje Intermedio y Plataformas de Ejecución independientes de lenguajes y sus implementaciones, en general, ofrecen ciertas ventajas tales como buenos mecanismos de seguridad y protección, posibilidad de desarrollar y ejecutar para un hardware del que no se dispone, independencia del hardware, pervivencia de sistemas antiguos. Se favorece en gran medida la portabilidad, interoperabilidad y flexibilidad.

Por otro lado, entre las desventajas, tenemos que la implementación de una máquina virtual puede ser costosa y lenta, agregan gran complejidad al sistema en tiempo de ejecución.

En cuanto a los lenguajes, la principal desventaja de los basados en máquina virtual, es que efectivamente son más lentos que aquellos completamente compilados, debido a la sobrecarga que genera tener una capa de software intermedia entre la aplicación y el hardware de la computadora.

Si tratamos de hacer comparaciones entre la tecnología Java y la de .NET, se destaca el hecho de que la primera se basa principalmente en que código fuente escrito en el lenguaje Java es el que se compila al lenguaje intermedio para luego ser ejecutado por la máquina virtual. Sin embargo, .NET tiene una perspectiva más amplia, pues tiene como uno de sus objetivos que varios lenguajes de programación puedan compilarse a su lenguaje intermedio, para posteriormente, ejecutarse en su máquina virtual, y de hecho, esto ya es una realidad, porque además de C# hay varios otros lenguajes que hacen lo mencionado. La especificación del lenguaje intermedia es lo bastante flexible como para permitir que el código de lenguajes de programación bastante diferentes puedan ser compilados a MSIL sin perder su potencia de expresividad.

Finalmente, puede resaltarse el hecho de que en la mayoría de los textos encontrados, se considera al entorno .NET como poseedor de un mejor desempeño por sobre el JVM. Este detalle es realmente importante por el hecho de que el entorno .NET no se encuentra exclusivamente enfocado a un solo lenguaje.

5. Bibliografía

- BASTIAS, LUIS.
  Máquina Virtual de JAVA. Noviembre 1998.
  http://www.dcc.uchile.cl/~lbastias/java/JVM.html

- CARPE GARCÍA, FRANCISCO; SEVILLA RUIZ, DIEGO.
  Estudio de la plataforma .NET. Diciembre – 2001.
  http://www.ditec.um.es/cgi-bin/dl/ProyectoNET.pdf

- FRANCO GRACIA, ANGEL.
  La Máquina Virtual JAVA. Enero 2000.
  http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/introduccion/virtual.htm

- LINDHOLM, TIM; YELLIN, FRANK.
  The Java Virtual Machine Specification. 2nd Edition. Sun Microsystems, Inc
  http://www.javasoft.com/docs/books/vmspec

- MENCHACA MENDEZ, ROLANDO; GARCIA CARBALLEIRA, FELIX.
  Arquitectura de la Máquina Virtual JAVA
  http://www.revista.unam.mx/vol.1/num2/art4/

- MSDN, Microsoft Corp.
  Interoperabilidad entre lenguajes en el .NET Framework. Enero 2001.
  http://www.microsoft.com/latam/msdn/articulos/2001/01/art04/default.asp

- Portal para desarrolladores .NET frameworks.
  En que consiste .NET?
  http://www.clikear.com/otros/plataformanet/plataforma.asp

- RUBIOLO, DANIEL; MEIER J. D.; et al.
  Microsoft .NET Explained.
  http://www.palermo.edu.ar/cyt/noticias/MICROSOFT_NET_EXPLAINED__3_.DOC

- VENNERS, BILL.
  Inside the Java 2 Virtual Machine. Editorial McGraw Hill 1998
  http://www.artima.com/insidejvm/ed2/ch05JavaVirtualMachinePrint.html

- VENNERS, BILL.
  *The Lean, Mean Virtual Machine.*1996.
  http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm-p1.html