**Keywords:** Meta-Programación, Reflexión, Generación de Código, Lenguajes de Muy Alto Nivel, Programación de Aspectos, Programación Intencional, Compiladores Extensibles, Software Automodificable, Lenguajes de Dominio Especifico, Reutilización.

## Motivación:

Los problemas existentes en el diseño y desarrollo de software a gran escala están bien documentados, para enfrentar los problemas que plantea el desarrollo de software, y más importante los problemas que planteará en el futuro es preciso realizar una reingeniería de las herramientas y técnicas utilizadas en la actualidad. Cada vez más especialistas en ingeniería de software [1, 3, 4, 5, 8, 13] concuerdan en la necesidad de esta reingeniería en la infraestructura utilizada para el desarrollo. Más específicamente queda claro que las herramientas de próxima generación no pueden ser lenguajes y compiladores con un diseño e implementación monolítica como sucede en la actualidad con los lenguajes más populares como C++, Java, C#, Ada.

Otra clave en la que se coincide, en cuanto a los requerimientos de las herramientas de próxima generación, es en la necesidad de incluir grandes capacidades de reflexión y extensibilidad a los lenguajes y herramientas del futuro [1, 3, 5, 13], dotando de una capacidad expresiva mayor al fuente de una implementación y sobre todo la capacidad de construir software que pueda autoexaminarse, corregirse y en general transformarse.

El proyecto se origino en un principio como un conjunto de herramientas para facilitar la implantación de software multiplataforma, sin embargo, prontamente se volvió evidente los beneficios que proporcionaría incluir técnicas y herramientas de metaprogramación, reflexión y extensibilidad.

## Objetivos:

El objetivo principal y general del proyecto es el desarrollo de un conjunto de herramientas y técnicas de programación que permitan implementar un framework de desarrollo de software de próxima generación. Éste objetivo general se ha descompuesto en los siguientes más específicos:

- Permitir la utilización de la infraestructura existente en cada plataforma (librerías de clases, componentes, etc.).
- Extender el tiempo de vida de una implementación particular a décadas (20, 30, 50 años).
- Reflexión completa, orientada a objetos, en tiempo de compilación.
- Desarrollo de Software Multiplataforma.
- Portabilidad entre plataformas sin pérdida de rendimiento.
- Permitir implementar el software desarrollado en plataformas de software o hardware inexistentes al momento del desarrollo inicial.
- Componentes Modulares y Abiertos (Compilador Extensible con “Plug-Ins”).
- Múltiples Lenguajes de Alto Nivel como origen.
- Fácil implementación de Lenguajes de Alto Nivel.
- Capacidades RAD incorporadas a la plataforma de desarrollo (compiladores) independientes del lenguaje de programación, entorno y plataforma.
- Lenguajes de Alto Nivel Extensibles.
- Extensiones de Lenguajes programables fácilmente y utilizables por cualquier lenguaje cliente.

## Consideraciones:

Para implementar los objetivos se tuvieron en cuenta las siguientes consideraciones durante el diseño de la tecnología LayerD:

- Es probado históricamente que no se pueden dar grandes saltos en tecnología más halla del limite que nos impone la restringida capacidad de cambio de las personas. Así como no se paso de codificar programas en tarjetas de cartón a utilizar el paradigma de orientación a objetos en lenguajes con entornos de ejecución controlada como Java o C# de un día para el otro, creemos que no será posible una migración directa de todo lo que implica el Paradigma de Orientación a Objetos, en cuanto al diseño e

implementación de software, en la ingeniería de software actual a paradigmas más abstractos como los propuestos Paradigma de Orientación a Lenguajes [4] o el Paradigma de Programación Intencional [5]. Por tanto, en el diseño de LayerD se ha tomado el compromiso de desarrollar herramientas que le sean familiares a los desarrolladores e ingenieros de software, pero que al mismo tiempo le permitan gradualmente implementar funciones propias de enfoques más avanzados como los mencionados paradigmas, la meta-programación masiva, la orientación a aspectos, librerías activas, etc.

- Se considera que el desarrollo de una nueva tecnología que no permita la reutilización de la infraestructura existente o que incurra en la perdida de rendimiento al utilizar los bienes de software disponibles es totalmente inaceptable por las perdidas económicas que ello causaría. Por tanto, el diseño de la tecnología LayerD maximiza la reutilización completa de la infraestructura actual para el desarrollo de software, como ser librerías de clases, herramientas de administración de código, depuradores, etc. Incluso se planea implementar meta-lenguajes "clones" de algunos lenguajes populares en la actualidad para minimizar el tiempo de inserción en el mercado de la tecnología.

- En general, en el diseño e implementación se ha tratado de no perder de vista la necesidad de que una tecnología para desarrollar software de próxima generación debe funcionar como un puente entre el presente y el futuro de la ingeniería de software para poder llegar a ser exitosa, de lo contrario los esfuerzos por producir tecnología que no este preparada para las condiciones actuales del mercado serian en vano.

**Diseño Básico:**

El diseño global del framework LayerD es el siguiente:

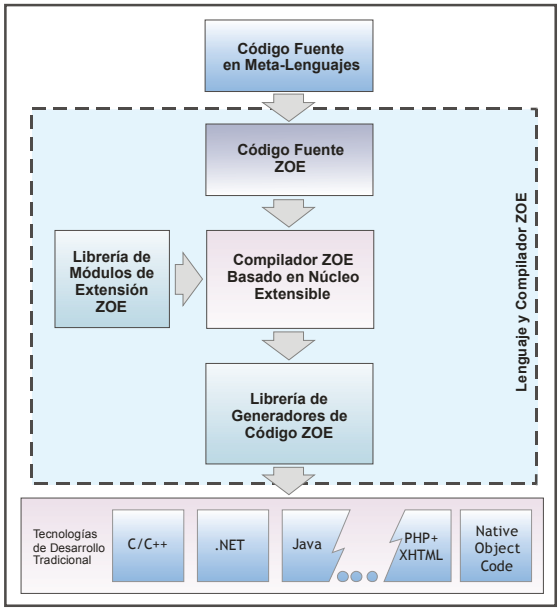



**Figura 1**

Esta compuesto por los siguientes componentes:

**- Meta-Lenguajes de muy alto nivel:** son lenguajes de programación ordinarios como Java, C++, Ada, con la excepción de que generan código en el lenguaje ZOE como salida y no código objeto o bytecode. Un meta-lenguaje no debe necesariamente ser orientado a objetos o un lenguaje tradicional, bien puede ser un lenguaje gráfico o utilizar otro paradigma como el funcional o el lógico.

**- Lenguaje ZOE:** es el lenguaje intermedio del framework. Todo meta-lenguaje debe generar código ZOE, por tanto en una medida todo programa en el framework LayerD es un programa ZOE o debe ser transformado en un programa ZOE. El lenguaje ZOE posee capacidades de Meta-programación y reflexión en tiempo de compilación, además tiene un diseño modular.

**- Librería de Generadores de Código ZOE:** la etapa de generación de código en el compilador ZOE es modular, por tanto es posible "enchufar" generadores de código para diferentes plataformas. El compilador ZOE puede entregar un código diferente a cada Generador de Código adaptado a las necesidades particulares de un generador, esto es con el objetivo de que la implanta-

ción de un Generador de Código ZOE sea una tarea relativamente sencilla.

Los Meta-Lenguajes pueden implementarse muy fácilmente, por requerirse sólo las etapas de análisis léxico y sintáctico dejando el análisis semántico al compilador ZOE.
Los Generadores de Código ZOE son sencillos de construir para plataformas que ya soporten objetos, como ser para la generación de código en lenguajes orientados a objetos tradicionales, como ejemplo se puede citar que un generador de código básico para la plataforma Microsoft .NET se desarrollo en sólo poco más de tres semanas por un único desarrollador.

**El lenguaje ZOE, Meta-programación y Extensiones:**

El lenguaje ZOE se escribe en un dialecto XML, por tanto otorga interesantes posibilidades como ser:
- La integración de información relacionada en el propio código utilizando un espacio de nombres XML diferente.
- La presentación del código en forma legible con diferentes "pieles" (en inglés "skins") usando hojas de transformación XSLT.

Los fuentes del lenguaje ZOE pueden permitirse ser escritos en XML por ser un lenguaje que será utilizado por los compiladores de los Meta-Lenguajes y no por programadores directamente.
Para el lenguaje ZOE se ha inventado una nueva estructura semántica denominada "Classfactory" (y más generalmente "Tipos Factory" [17]). Está estructura, que es una forma especial de clases ordinarias, permite implementar fácilmente capacidades de Meta-programación, e implementa la reflexión en tiempo de compilación del lenguaje lo que permite escribir programas que pueden examinarse a si mismos, generar código, optimizar código, emitir advertencias y errores personalizados, implementar librerías activas [11] y en general realizar cualquier tarea que sea posible durante la compilación de un fuente.

Otras características y beneficios de las extensiones del lenguaje ZOE existen, como ser la posibilidad de desarrollar herramientas RAD, la posibilidad de implementar las extensiones para soportar múltiples plataformas de forma gradual, etc.
Todas las extensiones programadas en cualquier Meta-lenguaje para el lenguaje ZOE, pueden ser utilizadas por cualquier otro Meta-lenguaje (diferente) sin cambios.

**Reutilizar la infraestructura existente**

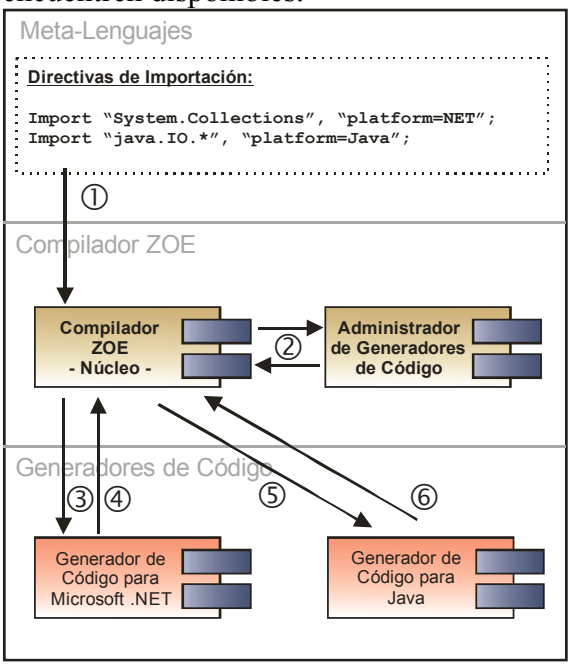
Como se ha planteado, al diseñar una nueva tecnología de desarrollo se cree que es fundamental la integración de las nuevas herramientas y técnicas con las plataformas de desarrollo actualmente en uso.
Para lograr el objetivo de permitir la reutilización de la mayor parte de la infraestructura existente en cada plataforma para la cual se construya software con el framework LayerD sin incurrir en perdida de rendimiento el diseño se basa en los siguientes puntos:

1º - Los tipos disponibles en cada plataforma son importados y traducidos a sus equivalentes en el lenguaje ZOE.
2º - La generación de código se modularizo en componentes "enchufables", los Generadores de Código ZOE.
3º - Las capacidades de meta-programación del lenguaje ZOE permiten encapsular transformaciones de código complejas, permitiendo el desarrollo de librerías activas [11] que transformen el código abstracto de alto nivel a las interfaces y protocolos requeridos en cada plataforma.

**Importación de Tipos**

Si se desea reutilizar la infraestructura existente en las diversas plataformas de desarrollo usadas actualmente se necesita un medio por el cual sea posible conocer las características de dicha infraestructura en el compilador ZOE para poder realizar un correcto análisis semántico, de lo contrario el compilador ZOE nunca podría conocer de qué tipos se dispone en una plataforma en particular.

Adicionalmente se necesita un proceso de importación automatizado y transparente al programador de alto nivel, si se requiriese una compleja elaboración por parte del desarrollador de alto nivel, entonces la reutilización de los bienes de software existente no sería aplicable en la práctica.
El proceso para importar los tipos existentes en cada plataforma se ilustra en el gráfico 2. El gráfico muestra el proceso comenzando por la única tarea necesaria de realizar por el programador de alto nivel para tener acceso a los tipos existentes en una plataforma en particular: **(1)** el programador escribe en el fuente del meta-programa una *directiva de importación*, la cual no difiere notablemente de sentencias extensamente usadas en muchos lenguajes de programación. Las directivas de importación indican de cual plataforma el programador intenta importar los tipos.
Al compilar el programa el compilador ZOE el proceso continúa en **(2)** donde el núcleo del compilador ZOE solicita al *Administrador de Generadores de Código* interfaces a los generadores de código que se encuentren disponibles.




**Reutilización de Software - 1º) Importación de Tipos:**
La información de los tipos y componentes disponibles en cada plataforma de implementación es obtenida por los Generadore de Código, luego analizada y procesada por el compilador ZOE, poniendo la infraestructura existente a disposición del programador de alto nivel.

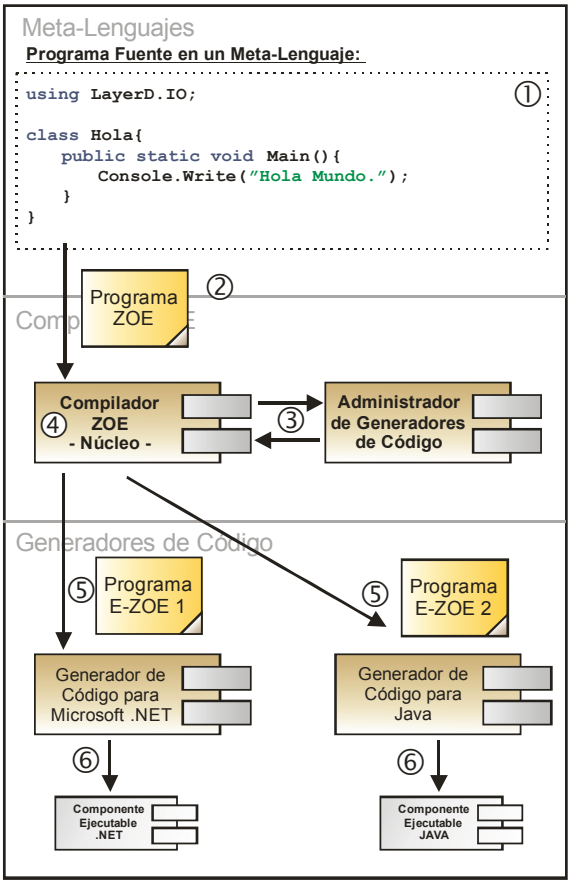
**Figura 2**

Con interfaces directas a los generadores de código el compilador ZOE procesa las directivas de importación escritas por el programador en **(1)** y hace procesar las directivas por el generador de código adecuado teniendo en cuenta la plataforma especificada en la directiva, en el ejemplo del gráfico la primera directiva de importación es procesada en **(3)** y **(4)** por el generador de código para la plataforma Microsoft .NET, la segunda directiva es procesada por el generador de código para la plataforma Java en **(5)** y **(6)**. En el paso **(3)** el compilador ZOE solicita el procesamiento de la directiva de importación al generador de código para .NET, el generador de código obtiene la información de tipos solicitada y genera información de tipos adecuada al lenguaje ZOE la cual devuelve como respuesta al compilador ZOE en el paso **(4)**. Los pasos **(5)** y **(6)** son iguales a los pasos **(3)** y **(4)**, pero esta vez la directiva es procesada por el generador de código Java.

## Generación de Código

Otro punto fundamental en la reutilización de la base de software instalada y en la no perdida de performance tiene que ver con la generación de código ejecutable final.
Los Generadores de Código ZOE aparte de la importación de tipos descripta en el punto anterior son responsables de precisamente generar código ejecutable para la plataforma de software o hardware para la cual fueron diseñados.
Para facilitar el diseño y construcción de la etapa de traducción de código de los generadores el compilador ZOE es capaz de entregar código preprocesado personalizado a cada generador de acuerdo a sus propios requerimientos.
El gráfico 3 muestra el proceso simplificado seguido desde la escritura del código fuente de alto nivel por el desarrollador hasta la producción del componente ejecutable en una o más plataformas. En **(1)** se escribe el programa de alto nivel en algún meta-lenguaje, el compilador del meta-lenguaje genera como salida **(2)** un programa ZOE, en **(3)** el núcleo del compilador ZOE obtie-

ne interfaces directas a los generadores de código disponibles, en **(4)** el núcleo realiza la importación de tipos (como se detallo en el apartado anterior), el análisis semántico del programa ZOE, y realiza el procesamiento relacionado con las capacidades de meta-programación.
Si el resultado de **(4)** es correcto, es decir el programa es semánticamente valido de acuerdo al análisis estático realizado y se han podido procesar todas las acciones de meta-programación indicadas por el programa ZOE, el núcleo del compilador genera una nueva versión del programa ZOE denominada "Programa ZOE Extendido". En **(5)** el núcleo entrega una versión "Extendida" del programa ZOE procesado a los generadores de código, como indica la figura el programa ZOE Extendido entregado a los diferentes generadores de código no es igual, cada generador de código recibe una versión del programa adaptada a los requerimientos especificados por dicho generador.




**Reutilización de Software - 1º) Renderización de Código:**
El código fuente en el meta-lenguaje original se traduce a código ZOE, el núcleo del compilador ZOE procesa el programa y entrega versiones diferentes de código ZOE "Extendido" a cada generador de código.

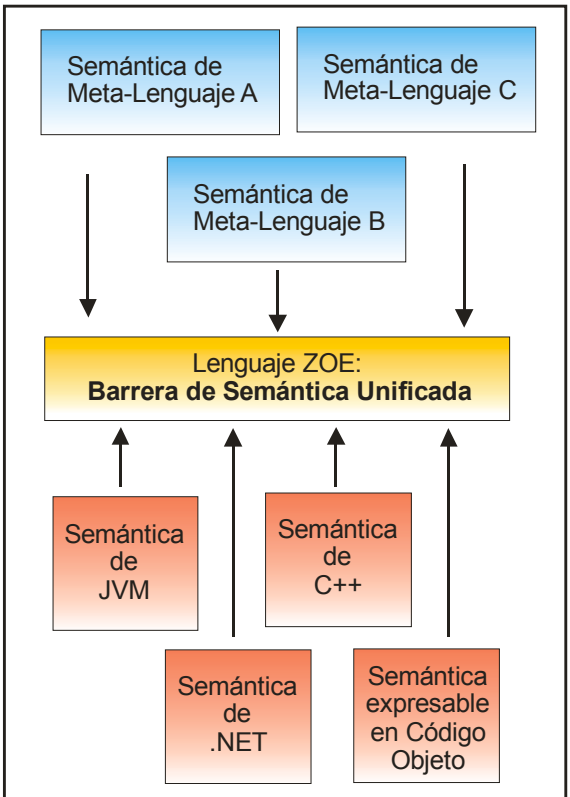
**Figura 3**

Finalmente en **(6)** los generadores de código construyen los componentes ejecutables para la plataforma para la cual fueron diseñados.
El código generado para cada plataforma es nativo a dicha plataforma sin incurrir en ninguna pérdida de rendimiento adicional en comparación con un software escrito originalmente para dicha plataforma.
Utilizando la modularidad del compilador ZOE y la etapa de generación de código es posible utilizar un único meta-lenguaje para programar en más de una plataforma de software o hardware, por otro lado, para poder escribir software multiplataforma, es decir a partir de un único programa fuente implementar en más de una plataforma distinta se requiere aplicar técnicas de encapsulación de meta-programación y liberarías activas, procesos que pueden ser programados utilizando las capacidades de extensión y reflexión en tiempo de compilación del compilador ZOE.

## Semántica Unificada

Cualquier intento por desarrollar una tecnología que permita homogeneizar el desarrollo en diversas plataformas de software y hardware que no posea una semántica clara y común para los programas implementados con dicha tecnología sería inaceptable. Si la declaración de una clase en un meta-lenguaje tuviera significados diferentes de acuerdo a la plataforma en la que se implemente el software, la escritura de programas reutilizables y realmente abstractos sería imposible. Por esto al diseñar el framework LayerD se utiliza al lenguaje ZOE como unificador de las diversas semánticas que poseen plataformas de implementación diferentes como al comparar Java, .NET y C++.
El gráfico 4 muestra el concepto de la unificación semántica realizada por el lenguaje ZOE. Como muestra el concepto ilustrado en la figura 4, todo meta-lenguaje debe generar código expresable mediante un programa ZOE, por tanto todo programa des-

arrollado utilizando el framework LayerD comparte la semántica del lenguaje intermedio ZOE. La semántica de un meta-lenguaje puede diferir de la semántica del lenguaje ZOE siempre y cuando pueda expresarse con construcciones disponibles en el lenguaje intermedio.
En sentido inverso la semántica de cada plataforma de salida es adaptada a la semántica del lenguaje ZOE, igual es posible para la semántica de una plataforma de salida diferir de la semántica del lenguaje ZOE siempre que sea posible expresar mediante una transformación, estática preferiblemente, la semántica definida por el lenguaje ZOE.
Esta unificación semántica es importante porque permite garantizar que un programa de LayerD poseerá el mismo significado en todas las plataformas de salida, si una plataforma de salida no soporta una característica semántica (por ejemplo herencia múltiple) el programa no podrá construirse para dicha plataforma, pero bajo ninguna circunstancia se generará un componente cuyo comportamiento no sea el esperado o bien no éste determinado.



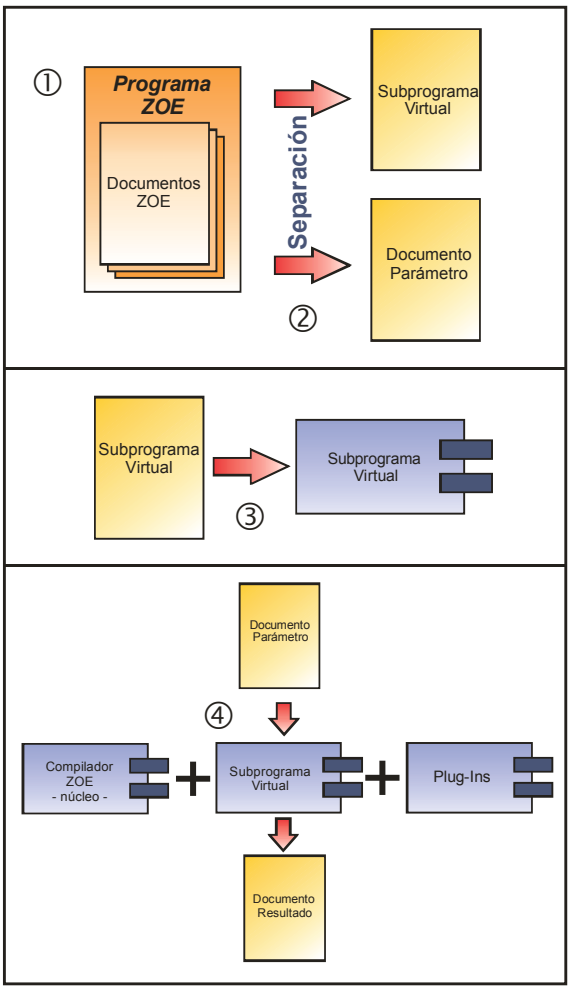

**El lenguaje ZOE como barrera y unificador semántico**

**Figura 4**

Con la unificación semántica un programador de alto nivel sólo debe aprender una única semántica y utilizar dichas reglas para diseñar y construir programas para todas las plataformas que soporte el framework LayerD.
Al diseñar la semántica del lenguaje ZOE se ha seguido una serie de reglas de diseño específicas y se ha tenido en cuenta las principales plataformas orientadas a objetos en uso actualmente.

## Código Fuente Dinámico con "Comportamiento Asociado"

Los generadores de código modulares, los múltiples meta-lenguajes de alto nivel como entrada y la semántica unificada proporcionan herramientas útiles para elevar el nivel de abstracción del software implementado y forman la base del soporte multiplataforma del framework de desarrollo propuesto por el proyecto.
Adicionalmente a los diseños modulares y abiertos de los diversos componentes dentro del framework LayerD, el lenguaje intermedio ZOE posee en su diseño capacidades de meta-programación y extensión.
Un programa ZOE se compone de uno o más *archivos fuentes ZOE* más un *archivo de programa* el cual describe los fuentes que componen el programa, las características requeridas por las plataformas de salida y las plataformas de salida para la cual fue diseñado el programa originalmente. El procesamiento cuando el programa ZOE es procesado por el compilador ZOE se muestra en la figura 5.




**Proceso de Compilación de un Programa ZOE**

**Figura 5**

En el punto **(1)** de la figura 5, el compilador ZOE realiza el análisis semántico del *programa ZOE* recibido como entrada, luego se realiza un proceso denominado *Separación* **(2)** el cual genera dos documentos a partir del programa de entrada con el análisis semántico ya realizado, estos dos documentos son el *Subprograma Virtual* y el *Documento Parámetro*. Nótese que la transformación es automática siguiendo una serie de reglas semánticas y que dicho proceso se realiza independientemente de los errores encontrados durante el análisis semántico en el paso **(1)**. El *Subprograma Virtual* generado en el paso **(2)** es efectivamente el *comportamiento asociado* al *programa ZOE*, en el siguiente paso el compilador construye un componente de ejecución dinámica **(3)** con el *Subprograma Virtual*.

El siguiente proceso se denomina *Fabricación* y esta indicado en el punto **(4)** de la figura 5. Durante la *fabricación* se ejecuta el componente construido a partir del *Subprograma Virtual* junto con el núcleo del compilador ZOE y los plug-ins que utilice dicho *Subprograma Virtual*, lo importante del proceso de *fabricación* es el procesamiento del *Documento Parámetro*, generado en el paso **(1)** a partir del *programa ZOE* original, y la generación como salida del *Documento Resultado*. El *Documento Resultado* es simplemente el *Documento Parámetro* procesado por la ejecución del *Subprograma Virtual*, es decir "*transformado*" por el *Subprograma Virtual*.

Una vez generado el *Documento Resultado*, éste es tratado como un *programa ZOE* y se vuelve a realizar el proceso a partir del paso **(1)**, si en la *separación* **(2)** el *Subprograma Virtual* obtenido es lo que se denomina un *Subprograma Virtual Nulo* entonces el proceso de compilación termina, el *Documento Resultado* (ahora tratado como un programa original) es procesado para generar código *ZOE Extendido* y es pasado al generador de código adecuado; si el *Subprograma Virtual* obtenido no es nulo se ejecuta todo el proceso completo y se genera otro *Documento Resultado*, el proceso continua hasta que en la separación **(2)** se obtiene un *Subprograma Virtual Nulo*. Por tanto, el proceso de compilación de un programa ZOE es recursivo, teniendo *N* tiempos de compilación, cada tiempo de compilación poseerá un *Subprograma Virtual* que transformará sucesivamente el *Documento Parámetro* en cada proceso de fabricación **(4)**.

El *Subprograma Virtual* es un programa lineal muy simple compuesto por una única función la cual llama en turno a los plug-ins disponibles, el orden de llamada a los plug-ins y las funciones invocadas dependerán del *programa ZOE* de entrada. Cada vez que se llama a un plug-in se pasa como parámetro una referencia al *Documento Resultado* y una referencia al núcleo del compilador ZOE; con estas interfaces los plug-ins pueden inspeccionar el código del *Documento Parámetro* (recordemos que es una

porción del programa ZOE) y realizar análisis y transformaciones en dicho código, con la interfaz al núcleo del compilador ZOE los plug-ins pueden obtener información sobre el proceso de compilación, emitir errores o advertencias, obtener la información de tipos disponible, entre otras funciones.

Adicionalmente la mayoría de las llamadas a los plug-ins recibirán argumentos que representan porciones de código en el programa original como árboles de expresiones y bloques de instrucciones, también en general devolverán porciones de código como árboles de expresiones o tipos; luego el *subprograma virtual* se encarga de la composición de esas estructuras retornadas en el punto correcto del *Documento Parámetro* de acuerdo al lugar del programa donde fue invocada dicha llamada al plug-in.

La figura 6 muestra un programa ZOE y su Documento Parámetro y Subprograma Virtual asociados (para el primer tiempo de compilación). En la figura se muestra código en el meta-lenguaje "Meta D++" el cual posee una sintaxis muy similar al lenguaje C++.

El programa cliente utiliza un plug-in de nombre "HttpReader", nótese que el programa no difiere en manera apreciable de un programa ordinario escrito en la actualidad en cualquier otro lenguaje orientado a objetos como C#, Java o C++, esto es como se ha dicho en la introducción con la intención de que la utilización de mecanismos avanzados de meta-programación y reflexión en tiempo de compilación sea transparente al programador de alto nivel y no requiera aprender conceptos radicalmente nuevos permitiendo una migración gradual a paradigmas más abstractos y con beneficios que el paradigma de orientación a objetos no permite explotar.

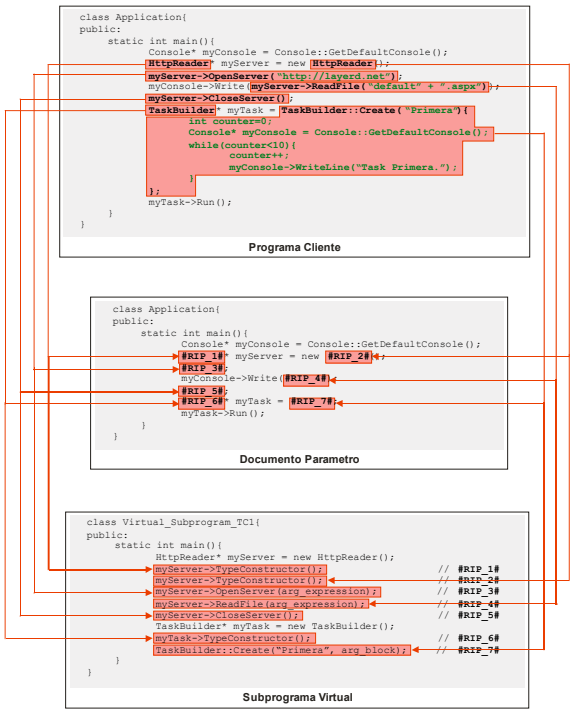



**Figura 6**

El *Documento Parámetro* y el *Subprograma Virtual* mostrados en la figura 6 son conceptuales, los actuales difieren en complejidad (sobre todo el *subprograma virtual* mostrado esta muy simplificado en relación al verdaderamente construido por el compilador ZOE), sin embargo muestran el concepto de su estructura.

El programa cliente de ejemplo en la figura 6 muestra como los plug-ins del compilador ZOE son utilizados por los programas en los meta-lenguajes de forma idéntica (o al menos muy similar) a cualquier otra clase ordinaria. Es importante notar la relación que existe entre la declaración de una variable del tipo de un plug-in, y la subsiguiente utilización de dicha variable. La figura 6 muestra como cada nombramiento de tipos del plug-in y cada invocación a un miembro del tipo del plug-in generan una invocación relacionada en el *subprograma virtual* y adicionalmente un *"#RIP#"* o *"punto de inserción de retorno"* en el documento parámetro.

A partir del *subprograma virtual* mostrado en la figura 6 es sencillo comprender la relación que existe entre el programa original y dicho subprograma. Efectivamente el *subprograma virtual* es el *"comportamien-*

*to asociado"* al código fuente del programa cliente original. Cada invocación a una función del plug-in *"HttpReader"* en el subprograma virtual procesará los argumentos (que serán árboles de expresiones y bloques de instrucciones, marcados en verde en la figura 6) y devolverá una estructura representando una porción de programa que será *"inyectada"* en el correspondiente *punto de inserción de retorno* asociada a cada invocación.
Luego de la ejecución del *subprograma virtual*, todos los *"#RIP#"* del *documento parámetro* habrán sido reemplazados por las estructuras que hayan devuelto las invocaciones a los miembros del plug-in "HttpReader" y se transformará en el *Documento Resultado*.
Los plug-ins no sólo son utilizados como clases ordinarias por los programas clientes, además son desarrollados como clases ordinarias con el modificador de declaración especial "factory". Explayarse sobre la escritura y funcionamiento exacto de las "Classfactorys" (que es el termino acuñado para las clases que implementan los plug-ins del lenguaje ZOE) queda fuera del alcance del presente trabajo, para más información vea [17]. Cabe aclarar que las Classfactorys pueden ser escritas en un mismo programa junto con clases ordinarias o en programas separados. El proceso completo de compilación cuando se incluyen clases ordinarias y classfactorys en un mismo programa es más complejo que el mostrado en el presente apartado por requerirse un análisis de dependencia y ciclos completos de compilación para cada sección de extensión, vea [18].

**Desarrollo Actual**

El framework propuesto se compone de múltiples artefactos requiriéndose para su funcionamiento completo al menos un meta-lenguaje y su compilador asociado que genere código ZOE, el compilador ZOE, y al menos un generador de código.
Actualmente los componentes desarrollados son los siguientes:

***Meta-Lenguajes:***
- Meta D++: un lenguaje orientado a objetos que mapea "uno a uno" todas las características del lenguaje intermedio ZOE, utiliza sintaxis heredada de C++ y C#. Su compilador esta construido y funcionando aunque se requiere más trabajo y depuración para considerarlo estable.
- Argentino: es el nombre provisorio para un meta-lenguaje orientado a objetos con sintaxis en español orientado a la enseñanza y a programadores de alto nivel de habla hispana. El compilador esta desarrollado en etapa de depuración.

***Lenguaje y Compilador ZOE:***
- El lenguaje se encuentra definido en la mayor parte, sin embargo aún queda un considerable trabajo a realizar en la definición semántica de algunos conceptos. El compilador se encuentra desarrollado en los siguientes componentes: analizador semántico principal, controlador del núcleo, administrador de generadores de código; y se esta trabajando en los componentes de: separador de código, administrador de plug-ins, controlador de fabricación.

***Generadores de Código ZOE:***
- Generador de Código a .NET: se encuentra desarrollado tanto la etapa de importación de tipos como la generación de código. Actualmente soporta plataformas .NET 1.0 y 1.1, soporta la importación de ensamblados .NET 2.0 sin incluir los tipos parametrizados.
- Generador de Código Java: se encuentra desarrollado tanto la etapa de importación de tipos como la generación de código. Soporta plataforma Java hasta versión 1.4.
En base a los componentes desarrollados es posible escribir programas en uno de los dos meta-lenguajes disponibles (Meta D++ y Argentino), realizar el análisis semántico con el compilador ZOE y generar código para las plataformas .NET y Java. Sin embargo a la fecha no se pueden explotar las capacidades de meta-programación y extensión del compilador ZOE por encontrarse en una etapa temprana de desarrollo. Se es-

pera realizar las primeras pruebas completas del concepto a finales de Mayo de 2007.

## Comparaciones con otros Sistemas de Meta-programación y compiladores Extensibles

Los múltiples meta-lenguajes como entrada a la plataforma pueden compararse con los lenguajes de la plataforma Microsoft .NET, con la salvedad importante que el código generado por los compiladores de los meta-lenguajes no es código ejecutable directamente ni una representación intermedia de código ejecutable, sino código en otro lenguaje de alto nivel, en lenguaje ZOE. A diferencia de un compilador para la plataforma .NET un compilador para el framework LayerD puede ser mucho más simple de implementar, si el meta-lenguaje utiliza la semántica del lenguaje intermedio sólo debe realizar el análisis léxico y sintáctico del programa fuente y la generación del árbol sintáctico en el formato requerido por el lenguaje ZOE (el cual posee con dicho fin formato de árbol). Por tanto un compilador para un meta-lenguaje no requiere necesariamente un analizador semántico, un optimizador de código ni un generador de código.

El lenguaje intermedio ZOE puede compararse en su estructura sintáctica con otros lenguajes que se escriben en XML como SuperX++ y o:XML [16], sin embargo a diferencia de estos lenguajes el lenguaje ZOE no esta destinado a programadores de alto nivel, sino a una representación intermedia común de programas y raramente un programador escribirá a mano programas ZOE, en general todo programa ZOE será generado por un compilador de un meta-lenguaje.

Las características de meta-programación y extensión del lenguaje ZOE son diferentes a otros diseños de compiladores extensibles como [6, 7, 10, 16 (Mozart)] en el sentido que no permite cambiar la sintaxis de un programa fuente, sólo el significado semántico y de manera controlada por el programa cliente de las extensiones. Adicionalmente, en comparación con estos compiladores extensibles, la programación de las extensiones para el lenguaje ZOE es comparativamente más sencilla por no requerirse manipular explícitamente árboles de sintaxis abstracta, tablas de símbolos o construcciones similares. Los sistemas de extensión basados en MOPs (Meta Objects Protocol) como Open C++ [12] y Open Java [14] guardan alguna similitud en la forma que permiten transformar el código en tiempo de compilación, sin embargo a diferencia de estos sistemas, en el framework propuesto, se puede utilizar en un único programa cliente múltiples extensiones, mientras que Open C++ y Open Java cada clase posee asociada una única "Meta-Clase", por tanto la reutilización de código y las posibilidades de las transformaciones es más limitada en estos sistemas. Adicionalmente las extensiones del lenguaje ZOE exigen el cumplimiento de una semántica y seguridad de tipos estricta que no exigen otros sistemas de meta-programación.

En las capacidades de manipulación y transformación de código en tiempo de compilación del lenguaje ZOE pueden encontrarse similitudes con los sistemas de macros programables y meta-programación de lenguajes funcionales como Lisp, Scheme, o lenguajes orientados a objetos como Smalltalk o Self. Sin embargo, la manipulación de código en el lenguaje ZOE sólo es permitida en tiempo de compilación, utiliza un enfoque orientado a objetos "imperativo" y no "puro" como Smalltalk, adicionalmente la posibilidad de transformación y análisis de código en el lenguaje ZOE son mayores que en estos sistemas por permitir analizar y transformar código que no fue proporcionado como argumento a las macro-estructuras por parte del programa original; siempre que el programa cliente lo permita.

Cabe mencionar que las capacidades de meta-programación y extensión del lenguaje ZOE deben adicionalmente considerarse dentro del framework LayerD completo, a diferencia de los sistemas nombrados, las características de extensión y meta-programación del lenguaje ZOE se circuns-

criben dentro de un diseño global más grande como los múltiples meta-lenguajes como entrada a la plataforma, los generadores de código modulares, la unificación semántica, artefactos que sumados proporcionan ventajas que no pueden ser comparadas con los restantes sistemas extensibles y de meta-programación aquí nombrados, por abarcar estos últimos un dominio de problema y una finalidad considerablemente menor.
Al analizar el framework completo es posible decir que proporciona herramientas para el desarrollo multiplataforma de sistemas orientados a objetos con la capacidad de incluir técnicas de meta-programación y paradigmas más abstractos como la Programación Orientada al Lenguaje [4] o la Programación Intencional [5], sin embargo a diferencia de los sistemas que se están desarrollando para estos paradigmas [3, 5] el framework propuesto no promueve el desarrollo de herramientas completamente novedosas, sino herramientas que resulten familiares a los desarrolladores, pero que permitan explotar gradualmente las ventajas de enfoques de desarrollo más productivos que la orientación a objetos por si sola.

**Beneficios Esperados y Aplicaciones:**

- No se pierde la inversión de lo desarrollado para otras plataformas como .NET o Java. La reutilización de la infraestructura existente es completa y transparente para el programador de alto nivel.
- No se requiere aprender nuevas librerías de clases, se usa lo existente en cada plataforma. Los Meta-Lenguajes iniciales serán muy similares a los lenguajes orientados a objetos populares en la actualidad, con lo cual se requiere muy poco tiempo de capacitación inicial. Adicionalmente se planea la construcción de compiladores de meta-lenguajes clones de Java y C#.
- Se puede aprovechar las ventajas (extensiones del compilador) gradualmente. Inicialmente se puede programar sólo utilizando las librerías existentes en la plataforma de implementación y orientación a objetos tradicional, gradualmente se puede incorporar técnicas de meta-programación, orientación a aspectos, programación intencional, librerías activas, etc.
- Utilización de un único lenguaje para implementar en más de una plataforma (por ej. Java, NET y C++).
- Los Plug-Ins pueden ser desarrollados por programadores con experiencia y utilizados por principiantes sin riesgos.
- Si en el futuro se quiere dejar de utilizar LayerD, no se pierde la inversión realizada. El software desarrollado con LayerD puede reutilizarse nativamente en la plataforma de implementación.
- Los Directores de Proyecto pueden forzar reglas semánticas a los programadores como la no utilización de campos públicos, la documentación del código, respetar el diseño en capas, etc.
- Al no permitir el cambio en la estructura léxica o sintáctica de los lenguajes de alto nivel, los fuentes de los programas de alto nivel siempre conservan una estructura básica similar sin importar cuantas extensiones utilice un programa. Esto facilita la lectura de los programas y garantiza que no se perderá legibilidad. Adicionalmente, cada Meta-Lenguaje utiliza su propia sintaxis y estructura léxica, por tanto los desarrolladores utilizan las extensiones de forma simple.
- Es posible y simple desarrollar DSLs (lenguajes de domino especifico) para el lenguaje intermedio ZOE, que pueden ser utilizados por cualquier meta-lenguaje cliente usando su propia sintaxis y estructura léxica. Por ejemplo se podrían desarrollar DSLs para el procesamiento de imágenes, consulta de bases de datos, manipulación y renderización de imágenes 3D, análisis y síntesis de sonido, etc.

**Conclusiones y Temas Pendientes**

El framework propuesto en el presente trabajo pretende acercarse lo más posible a la realización de una visión para las “herramientas de desarrollo de próxima generación” según como lo ven muchos expertos en la actualidad al futuro de las herramientas de desarrollo. Como premisa en el diseño se ha considerado primordial la reutili-

zación de la infraestructura existente y el permitir un paso gradual a la explotación de técnicas más abstractas y potencialmente más productivas que la orientación a objetos.

Aún no se ha podido demostrar la prueba del concepto por encontrarse las etapas de extensión del compilador ZOE bajo desarrollo, pero ya es posible utilizar un único meta-lenguaje para programar para más de una plataforma y se espera tener muy pronto las herramientas necesarias para probar el concepto completo detrás del framework propuesto.

Como temas pendientes a tratar han quedado los siguientes (entre otros):

- Seguridad en programas clientes: como limitar el acceso al código cliente por parte de las extensiones.
- Compilación Interactiva: capacidades RAD del lenguaje ZOE y los meta-lenguajes que permiten implementar características propias de entornos de desarrollo directamente en las herramientas básicas del framework.
- Escritura de Classfactorys: sobre la definición y escritura de extensiones para el compilador ZOE.
- Implementación de DSLs (Lenguajes de Dominio Específico) con Classfactorys.
- Clasificación de Generadores de Código: los generadores de código reciben una clasificación de acuerdo a las características semánticas estáticas y dinámicas que son capaces de implementar.

Para más información sobre estos temas puede ver [17, 18, 19].

En cuanto a los trabajos pendientes para implementar el framework LayerD sobresale la terminación del compilador ZOE y la depuración de los generadores de código y compiladores de meta-lenguajes disponibles.

## Referencias:


[1] *Dynamic Languages Research Group* at "MIT Computer Science And Artificial Intelligence Laboratory" (http://www.ai.mit.edu/projects/dynlangs/research.htm, http://www.csail.mit.edu/index.php).

[2] MetaL: An XML based Meta-Programming language (http://www.meta-language.net/).

[3] Sergey Dmitriev, "Language Oriented Programming: The Next Programming Paradigm" (http://www.onboard.jetbrains.com/is1/articles/04/10/lop/index.html).

[4] M. Ward. Language Oriented Programming. Software - Concepts and Tools, 15, 147-161. 2003. (http://www.dur.ac.uk/martin.ward/martin/papers/middle-out-t.pdf).

[5] Charles Simonyi. *The Death of Computer Languages, The Birth of Intentional Programming*. 1995. ftp://ftp.research.microsoft.com/pub/tr/tr-95-52.doc also ftp://ftp.research.microsoft.com/pub/tr/tr-95-52.ps.

[6] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers, "Polyglot: An Extensible Compiler Framework for Java". Proceedings of the 12th International Conference on Compiler Construction, Warsaw, Poland, April 2003. LNCS 2622, pages 138–152.

[7] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann, "Cetus, An Extensible Compiler Infrastructure for Source-to-Source Transformation". In Proceedings of the 16th Annual Workshop on Languages and Compilers for Parallel Computing LCPC '2003.

[8] Martin Fowler, 12-06-2005, "Language Workbenches: The Killer-App for Domain Specific Languages?".http://www.martinfowler.com/articles/languageWorkbench.html.

[9] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. "OpenJava: A Class-based Macro System for Java" *Lecture Notes in Computer Science 1826, Reflection and Software Engineering,* , Walter Cazzola, Robert J. Stroud, Francesco Tisato (Eds.), Springer-Verlag, pp.117-133, 2000.

[10] Adam Granicz and Jason Hickey, "Phobos: A front-end approach to extensible compilers". Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03).

[11] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, Todd Veldhuizen. "Generative Programming and Active Libraries" In Proceedings of Generic Programming '98, Lecture Notes in Computer Science 1766, Springer-Verlag Telos. (http://www.osl.iu.edu/~tveldhui/papers/dagstuhl1998/dagstuhl.html).

[12] S. Chiba. "A Metaobject Protocol for C++". In OOPSLA'95, pages 285-299, 1995.

[13] Gregory V. Wilson. "Extensible Programming for the 21st Century" ACM Queue vol. 2, no. 9 - Dec/Jan 2004-2005. (http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=247&page=1).

[14] Bachrach, J., and K. Playford. 2001. "The Java Syntactic Extender"; see (http://www.ai.mit.edu/~jrb/jse/jse.pdf).

[15] Knuth, D. E. 1992. Literate Programming. Cambridge University Press.
[16] Mozart: vea http://mozart-dev.sourceforge.net/ ; SuperX++: vea http://xplusplus.sourceforge.net/ ; o:XML: vea http://www.o-xml.org .
[17] Ferreyra, Alexis. 2005. “El Lenguaje Meta D++ (Borrador)”, Capitulo 14 – Tipos Factory, vea http://layerd.net/documentation/especificacion_lddpp_borrador.zip páginas 185-225.
[18] Ferreyra, Alexis. 2005. “El Lenguaje Meta D++ (Borrador)”, Capitulo 13 –Modelo de Ejecución, vea http://layerd.net/documentation/especificacion_lddpp_borrador.zip páginas 155-184.
[19] Vea: http://layerd.net.
[20] 2003, Microsoft Corporation – “C# Language Specification v1.2”
[21] 2005, Microsoft Corporation – “C# Language Specification v2.0”
[22] 2000, Sun Microsystems, Inc. – “Java Language Specification, Second Edition”
[23] 2005, Sun Microsystems, Inc. – “Java Language Specification, Third Edition”
[24] 1995, Intermetrics Inc. “INTERNATIONAL STANDARD ISO/IEC 8652:1995, ADA - Programming Language”