Instructivo introductorio de LayerD para programadores .NET

Contenido

1 Introducción

El presente es una pequeña guía sobre como programar en LayerD utilizando como plataforma de ejecución la plataforma Microsoft .NET. Recuerde que LayerD independiza la implementación de software de su entorno de ejecución y librerías utilizadas pudiéndose desarrollar software para diferentes plataformas e incluso software multiplataforma.

Todos los conceptos vertidos en éste documento sirven para programar en cualquier plataforma disponible, sin embargo los ejemplos mostrados asumen que se utiliza Microsoft .NET como entorno de ejecución y librerías de base y el lenguaje Meta D++ es enseñado comparándolo con el lenguaje C#.

Para más información, ejemplos y compiladores actualizados visite periódicamente http://layerd.net y http://layerd.blogspot.com.

2 Conceptos Básicos

2.1 Estructura básica

Toda clase en Meta D++ debe declararse dentro de un espacio de nombres, ej:

```
namespace Test {
	public class App{
	public:
		static void Main(string^[] args){
		}
	}
}
Por lo tanto no puede hacerse lo siguiente:

public class App{
public:
	static void Main(string^[] args){
	}
}
```

Para utilizar tipos de datos del .NET framework debemos incluir la directiva import en uno de los fuentes de nuestro programa:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
```

Los parametros de la directiva import se interpretan como sigue:

"System"	"platform=DotNET"	"ns=DotNET"	"assembly=mscorlib"
El espacio de nombres a importar, se importaran todos los tipos declarados	La plataforma de tiempo de ejecución para la cual se procesara la directiva import.	El nombre del espacio de nombres donde se situaran los tipos importados.	El nombre del assembly desde donde vamos a importar los tipos. Es posible indicar el path completo del archivo usando la opcion "assemblyfilename=MYASSEMBLY_PATH.dll"

dentro de dicho espacio de nombres, en el assembly especificado.			en lugar de "assembly=…"

La directiva import solo requiere ser escrita una única vez, por lo tanto si nuestro programa posee más de un archivo fuente, sólo debemos declararla en uno de ellos y no repetirla, si se repite simplemente sera ignorada.

Las directivas import son equivalentes a los argumentos "/reference:" del compilador de C#. A diferencia de éste lenguaje .NET en Meta D++ la referencia forma parte directamente del fuente del programa no teniéndose que especificar necesariamente las referencias al ejecutar el compilador.

2.2 Directivas import y using por defecto requeridas para programar con .NET en LayerD

Siempre que querramos hacer un programa para .NET deberemos importar el assembly por defecto de .NET, es decir "mscorlib", esto se hace con la directiva:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
```

Luego normalmente agregaremos las directivas "using":

```
using DotNET::System;
using DotNET::System::Collections;
```

Las directivas using son equivalentes a la directiva using de C# (o "import" en Java). Por tanto, se encarga de situar en el alcance (scope) actual los tipos dentro de dicho espacio de nombres para no requerir el nombramiento completo.

2.3 Hola Mundo para .NET en LayerD

El clásico hola mundo:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace Test {
	public class App{
	public:
		static void Main(string^[] args){
			Console::WriteLine("Hola Mundo");
		}
	}
}
```

Como puede apreciarse, si se requiere utilizar un tipo importado desde .NET simplemente se lo utiliza de la misma forma en la que se lo haría desde cualquier otro lenguaje .NET teniendo en cuenta que los tipos importados se encuentran dentro del espacio de nombres "DotNET".

Adicionalmente se deberán tener en cuenta las diferencias sintácticas y semánticas del lenguaje Meta D++, las más importantes de las cuales se enumeran a continuación:

	En C#	**En Meta D++**
Acceso a	`using Ns1.Ns2.Ns3;`	`using Ns1::Ns2::Ns3;`

miembros estáticos o nombramiento de tipos completos, se requieren "::".	`Console.WriteLine( "Hello World" )`	`Console::WriteLine( "Hello World" )`

En declaración de variables, campos, etc., donde el tipo sea una clase, se requiere el carácter "^" para indicar a Meta D++ que se trata de una referencia y no de un valor.	`string miCadena;`
	
	`ArrayList lista = new ArrayList();`
		`string^ miCadena;`
		
		`ArrayList^ lista = new ArrayList();`

En la declaración de miembros de clase no utilizar el modificador de acceso en el miembro, utilizar "public:" por ejemplo.	`class App{`
	
	`public void test(){}`
	
	`}`
		`class App{`
		`public:`
		
		`void test(){}`
		
		`}`

En la declaracion de herencia e implementación de interfaces utilizar las palabras claves "inherits" e "implements".	`class App : Base, IInterface {`
	
	`}`
		`Class App inherits Base implements IInterface{`
		
		`}`

En la declaración de propiedades se requiere la palabra clave "property" entre el tipo y el nombre de la propiedad.	`string Nombre{`
	`  get{return nombre;}`
	`  set{nombre = value;}`
	`}`
		`string^ property Nombre{`
		`  get{return nombre;}`
		`  set{nombre = value;}`
		`}`

Los indexadores se declaran con la palabra reservada "indexer".	`int this[string arg]{`
	`  get{}`
	`  set{}`
	`}`
		`int indexer(string^ arg){`
		`  get{}`
		`  set{}`
		`}`

Declaración de matrices. Las matrices son por defecto	`int[] myArray = new int[2];` `string[] myArray = new string[] {"uno", "dos"};`	`int[] myArray = new int[2];` `string^[] myArray = new string^[] = {"uno", "dos"};`

"referencias a matrices" por tanto se declara igual que en C# teniendo en cuenta las diferencias al declarar el tipo de los elementos de la matriz. Al declarar inicializadores de matriz se debe utilizar un signo "=" antes de las llaves.		

Para atachar eventos de tipos importados de .NET se deben utilizar los metodos "add_.." y "remove_.." para desasociar el evento a un método y el operador unario "&".	this.CSalir.Click+=new EventHandler(this.CDestinoB_Click);
	
	this.CSalir.Click-=new EventHandler(this.CDestinoB_Click);
		this.CSalir.add_Click(new EventHandler(&this.CDestinoB_Click);
		
		this.CSalir.remove_Click(new EventHandler(&this.CDestinoB_Click);

Ejemplo:

```
import "Microsoft", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System.Windows.Forms", "platform=DotNET", "ns=DotNET",
"assembly=System.Windows.Forms";
using DotNET::System;
using DotNET::System::Collections;
using DotNET::System::Windows::Forms;
using Zoe;

namespace EjemploNET{
	class MiVentana inherits Form{
		//Uso todo igual a .NET
		Button^ CSalir;
	public:
		 MiVentana(){
			this.Text = "Una ventana Windows Forms con LayerD";
			CSalir = new Button();
			CSalir.AutoSize = true;
			CSalir.Text = "Salir";
			this.Controls.Add(CSalir);
			//Enlazo eventos
			this.CSalir.add_Click(new EventHandler(&this.CDestinoB_Click));
		}
		void CDestinoB_Click(object^ sender, EventArgs^ e){
			ArrayList^ lista = new ArrayList();
```

```
                lista.Add("Hola ");
                lista.Add("Mundo ");
                lista.Add("desde ");
                lista.Add("Meta D++");
                foreach(string^ item in lista){
                        MessageBox::Show(item);
                }
                this.Close();
        }
    }
    class Principal{
    public:
        static void Main(string^[] args){
                Application::EnableVisualStyles();
                Application::Run(new MiVentana());
        }
    }
}
```

3 Utilización de los compiladores

Para compilar un programa Meta D++ desde la línea de comandos debe realizar dos pasos:

- Compilar el fuente con el compilador de Meta D++ para obtener el código intermedio Zoe.
- Compilar el código intermedio con el compilador Zoe.

El compilador de Meta D++ es “metadppc.exe” y el compilador Zoe es “zoec.exe”. Por ejemplo, si posee el fuente “HelloWorld.dpp” se procede de la siguiente forma:

```
c:\layerd\bin>metadppc.exe HelloWorld.dpp
c:\layerd\bin>zoec.exe HelloWorld.zoe
```

La secuencia de comandos anterior producirá “HelloWorld.exe”, note que el compilador de Meta D++ produce archivos de extensión .zoe por defecto. Para compilar un programa que posea más de un fuente primero compile los fuentes Meta D++ llamando al compilador metaddc y luego utilice el compilador zoec una vez, ejemplo:

```
c:\layerd\bin>metadppc.exe HelloWorld_Source1.dpp
c:\layerd\bin>metadppc.exe HelloWorld_Source2.dpp
c:\layerd\bin>zoec.exe HelloWorld_Source1.zoe HelloWorld_Source2.zoe –pn:HelloWorld
```

La opción “-pn:HelloWorld” indica que el nombre del programa Zoe el cual será utilizado como nombre de archivo de salida, por tanto se generará el archivo “HelloWorld.exe”.

3.1 Compilar librerías dinámicas

Para compilar una librería dinámica en lugar de un ejecutable utilice la opción “-lib” del compilador Zoe:

```
c:\layerd\bin>metadppc.exe HelloWorldLib.dpp
c:\layerd\bin>zoec.exe HelloWorldLib.zoe –lib
```

3.2 Compilar y utilizar Classfactorys

Si bien es posible declara y utilizar classfactorys en un mismo programa e incluso en un mismo fuente, aquí se explicara cómo utilizar classfactorys escribiéndolas en un archivo y programa separado al programa cliente.

Asumiendo que se poseen los archivos “MyClassfactory.dpp” y “MyClassfactoryClient.dpp” con la classfactory y el programa cliente respectivamente, se debe proceder como sigue con los compiladores:

```
c:\layerd\bin>metadppc.exe MyClassfactory.dpp
c:\layerd\bin>zoec.exe MyClassfactory.zoe –ae
Extension Added: MyClassfactory

c:\layerd\bin>metadppc.exe MyClassfactoryClient.dpp
c:\layerd\bin>zoec.exe MyClassfactoryClient.zoe
```

Primero debe compilarse la classfactory y agregarla a la librería de classfactorys del compilador Zoe utilizando el modificador “-ae” (agregar extensión) y luego compilar el programa cliente. Como muestra el ejemplo el compilador Zoe debe mostrar un mensaje indicando que la extensión se agrego al compilador.

Para ver las extensiones instaladas en el compilador Zoe y desinstalar extensiones utilice respectivamente las opciones “-le” y “-re:EXTENSION_NAME1,EXTENSION_NAME2,..” (O las opciones equivalentes “-listextensions” y “-removeextensions”). Ejemplo:

```
c:\layerd\bin>zoec.exe –le
Compilador ZOE - Interfaz de Usuario de Consola.
Extensiones Instaladas: 12

Nombre de Tipo, Archivo del Modulo, Interactive, Active
-------------------------------------------------------
Zoe.Utils, .\.\FactoriesLib\ZoeUtilsTemplates1.dll, False, True
Zoe.iUtils, .\.\FactoriesLib\ZoeUtilsTemplates1.dll, True, True
Zoe.Logger, .\.\FactoriesLib\ZoeUtilsTemplates1.dll, False, True
DataSample.MyType, .\.\FactoriesLib\DataSampleTemplates1.dll, False, True
DataSample.GUI, .\.\FactoriesLib\DataSampleTemplates1.dll, False, True
DataSample.iASPNET, .\.\FactoriesLib\DataSampleTemplates1.dll, True, True
DataSample.ASPNET, .\.\FactoriesLib\DataSampleTemplates1.dll, False, True
DataSample.Concurrent, .\.\FactoriesLib\DataSampleTemplates1.dll, False, True
DataSample.ClassGenerator, .\.\FactoriesLib\DataSampleTemplates1.dll, False, True
DataSample.ProgramChecks, .\.\FactoriesLib\DataSampleTemplates1.dll, False, True
DataModel2.Model, .\.\FactoriesLib\DataModelTemplatesPablo1.dll, True, True
DataModel.Model, .\.\FactoriesLib\DataModelTemplates1.dll, False, True

c:\layerd\bin>zoec.exe –re:DataSampleTemplates
Compilador ZOE - Interfaz de Usuario de Consola.
Extension DataSampleTemplates eliminada. Classfactorys eliminadas: 7
```

Para ver más opciones del compilador Zoe y el compilador Meta D++ ejecute los compiladores sin proporcionar argumentos para mostrar la ayuda en la línea de comandos.

3.3 Compilar para otras plataformas

Por defecto el compilador Zoe construido para .NET genera módulos para dicha plataforma utilizando el Generador de Código Zoe para .NET.

Si se desea compilar un programa multiplataforma para otra plataforma que no sea la por defecto se debe utilizar la opción “-p:PLATAFORMA” del compilador Zoe, por ejemplo para compilar utilizando el Generador de Código Zoe para Java:

```
c:\layerd\bin>metadppc.exe HelloWorld.dpp
c:\layerd\bin>zoec.exe HelloWorld.zoe –p:java
```

Si no se posee un Generador de Código adecuado para la plataforma indicada o no existen las Classfactorys adecuadas que soporten dicha plataforma se informaran los errores.

4 Tiempo de Compilación Vs. Tiempo de Ejecución

LayerD provee dos tiempos de ejecución diferenciados. El tiempo de ejecución es conocido por todos y refiere simplemente al momento de la ejecución del programa. Por otro lado LayerD nos permite realizar procesamiento durante el proceso de compilación de un programa.

Para programar el procesamiento deseado durante el tiempo de compilación se utilizan clases especiales denominadas “Classfactorys”.

4.1 Teoría rápida sobre Classfactorys

Las Classfactorys son simplemente clases destinadas a programar el procesamiento de tiempo de compilación de un programa LayerD. Durante la compilación no pueden evaluarse todos los valores, por otro lado, es posible acceder a construcciones que ya no existen en tiempo de ejecución.

Durante la compilación de un programa LayerD se puede acceder al código del programa que se está compilando (reflexión en tiempo de compilación) ya sea para analizarlo o modificarlo (siempre que la classfactory posea los permisos necesarios). Debido a esto las classfactorys permiten utilizar tipos de datos y objetos especiales.

Los tipos de datos especiales utilizables por las classfactorys son los siguientes:

Tipo	Descripción	Ejemplo

Expresión	Las variables de tipo expresión representan una expresión en el código fuente.
	Los miembros de classfactorys pueden tomar como argumento expresiones o sea de tipo expresión (como en el caso de campos o propiedades).
	Para declarar una variable como de tipo expresión debe utilizarse el modificador de declaración “exp” como en “exp int campo;” este modificador indica al compilador que lo que queremos guardar en dicha variable es una expresión y no su valor.
	Un tipo de dato expresión tiene asociado un tipo de expresión, por ejemplo “exp int varInt” tiene asociado el tipo int, por tanto sólo podrá almacenar expresiones de tipo int o de un tipo convertible a int.
	Las variables de tipo expresión son equivalentes al tipo “LayerD::CodeDOM::XplExpression^”.
		`factory class F{`
		
		`exp int fieldExp;`
		
		`exp void func(exp int arg1){`
		`}`
		
		`}`

Tipo	Las variables de tipo “type” (o sea tipo) representan un tipo de datos. Es posible tomar como argumento tipos o retornarlos desde funciones especialmente de “constructores de tipo”.
	El tipo intrínseco “type” es equivalente al tipo “LayerD::CodeDOM::XplType^”.
		`type varMyType;`
		
		`type Func(type arg1, type arg2){`
		`}`

Bloques	Las variables de tipo “block” nos permiten tomar como argumento de funciones bloques de código fuente para ser procesado. Cuando es utilizado como argumento de función, el tipo block debe ser el último argumento de la función y no puede poseerse más de	`block field;`
		`exp void func(block arg){`
		`field = arg;`
		`}`

	un argumento de tipo block. El tipo intrínseco “block” es equivalente al tipo “LayerD::CodeDOM::XplFunctionBody^”.	

Identificadores	Si todo lo que necesitamos tomar como argumento de una función es un identificador y no una expresión podemos utilizar el modificador de tipo “iname”, si declaramos un argumento “iname void argName” podremos tomar como argumento un identificador de cualquier tipo, incluyendo un identificador no definido, si declaramos el argumento “iname int argName” la función tomara un argumento que sea un identificador, pero el identificador deberá estar declarado y ser de tipo entero en este caso.
	El modificador de tipo intrínseco “iname” es equivalente al tipo “LayerD::CodeDOM::XplIName^”.
	Es posible utilizar la clase XplIName para crear identificadores que necesitemos utilizar en expresiones “writecode”, por ejemplo “XplIName id = new XplIName()” crea un iname de tipo “void” (sin tipo) con un nombre de identificador único, lo cual es útil para generar nombres de identificadores.
		`void func(iname void arg, iname int arg2){`
		
		`}`

Los objetos especiales a los cuales tienen acceso las classfactorys son los siguientes:

Objeto	**Descripción**	**Ejemplo**

context	El objeto “context” representa el contexto de llamada actual a un miembro de una classfactory.
	El objeto “context” posee miembros útiles como las propiedades “CurrentNamespace”, “CurrentClass”, etc., que representan las construcciones más inmediatas que engloban al contexto de llamada del miembro.
	El objeto “context” se utiliza para analizar el contexto de llamada o para utilizarlo como referencia al insertar un nodo.
		`void func (){`
		`…`
		`//representa la clase en la cual es llamada a la función func`
		`context.CurrentClass.Children().InsertAtEnd(member);`
		`…`
		`}`

currentDTE	El objeto “currentDTE” representa el código fuente que se está compilando actualmente y permite realizar análisis y modificación (siempre que se posean los permisos adecuados). El objeto “currentDTE” es de tipo “LayerD::CodeDOM::XplDocument^”, por tanto pueden utilizarse todos los miembros de esta clase.	`//Busca todas las funciones en el código` `currentDTE.FindNodes(“/@XplFunction”);`
compiler	El objeto “compiler” representa la instancia actual del compilador zoe que esta compilando el modulo. Es de tipo “LayerD::ZoeCompiler::ZoeCompilerCore^”. Se utiliza para agregar errores, advertencias o consultarlos, entre otras cosas.	`//Agrega un error al proceso de compilación` `compiler.Errors.Add(“error”, errorNode);`

4.2 Generar e Inyectar código en tiempo de compilación

Para generar código en tiempo de compilación se deben seguir dos pasos sencillos:

- Generar el código que se quiere inyectar en la memoria
- Insertar el código en memoria en el lugar deseado dentro del código del programa que se está compilando.

Para generar el código en memoria se lo puede realizar de tres formas:

- Usando las clases dentro del espacio de nombres LayerD::CodeDOM (o DotNET::LayerD::CodeDOM de acuerdo a la implementación del compilador Zoe utilizado).
- Usando la expresión “writecode”.
- Usando una combinación de las anteriores.

Normalmente se encontrará útil combinar las dos primeras formas de generar código, sobretodo utilizaremos la expresión “writecode” para generar el grueso del código y las clases del CodeDOM para complementar.

4.2.1 El CodeDOM

En LayerD se denomina “CodeDOM” al conjunto de clases declaradas dentro del espacio de nombres “LayerD::CodeDOM” (según la implementación del compilador Zoe puede estar contenido dentro de otro espacio de nombres como “DotNET::LayerD::CodeDOM”, “Java::LayerD::CodeDOM”, etc.).

Las clases dentro del CodeDOM representan porciones de código fuente Zoe en memoria, por ejemplo clases, funciones, expresiones, tipos, etc. Todas las clases que representan alguna porción de código fuente derivan de XplNode la cual es la clase base para todos los nodos que representan un fuente Zoe.

En memoria un fuente Zoe es representado con una estructura de árbol y las clases dentro del CodeDOM representan las ramas y hojas dentro del árbol.

Como clases más importantes dentro del CodeDOM pueden nombrarse las siguientes:

Clase	Descripción
`XplNode`	Es la clase base de todas las clases que forman parte de un documento Zoe en memoria.
`XplDocument`	Representa el nodo inicial de un Documento Zoe en memoria. Contiene un “DocumentData” que contiene meta-información e información de configuración y un “DocumentBody” donde se encuentra el código funcional del fuente.
`XplNamespace`	Representa un espacio de nombres Zoe, para establecer u obtener el nombre se utiliza “set_name” o “get_name”.
`XplClass`	Representa una clase, interface, enumeración u estructura Zoe. Por defecto representa una clase.

`XplFunction`	Representa una función miembro de una clase Zoe.
`XplProperty`	Representa una propiedad miembro de una clase Zoe.
`XplField`	Representa un campo miembro Zoe.
`XplType`	Representa un tipo de datos, es una estructura recursiva. Se utiliza para representar el tipo de datos “type”.
`XplFunctionBody`	Representa un cuerpo de función, contiene instrucciones. Se utiliza para representar el tipo de datos “block”.
`XplIName`	Representa un “iname” o identificador a los fines de ser utilizado por el compilador Zoe, pero no representa en si una porción de código fuente.
`XplClassMembersList`	No se utiliza como parte de un programa Zoe, sino para agrupar miembros de clase en memoria.
`XplExpression`	Representa un contenedor de algún tipo de expresión. Por ejemplo, puede contener una expresión literal, unaria, binaria, etc. Es el tipo de datos utilizado para representar los tipos de datos expresión “exp”.
`XplNodeList`	Es una lista de nodos de tipo que se utiliza para almacenar cualquier tipo de nodo derivado de XplNode.

Como XplNode es la clase base de todos los nodos en el CodeDOM su funcionalidad se comparte con la mayor parte de los tipos dentro del CodeDOM. Puede pensar a una instancia de XplNode como la representación de un nodo Xml del fuente Zoe, luego cada clase derivada representa una porción de código fuente Zoe diferente en memoria. La funcionalidad principal de XplNode se detalla en la siguiente tabla:

Miembro	**Descripción**
`get_ElementName () y set_ElementName(string^)`	Obtiene o estable el nombre del elemento XML.
`Children()`	Devuelve un XplNodeList con los nodos hijos. Sólo para elementos que puedan contener una lista variable de nodos hijos como clases, bloques, etc. Si el elemento no puede contener una lista variable de hijos, como una expresión, devuelve nulo.
`get_Parent()`	Devuelve el padre del nodo, nulo si no tiene padre.
`get_Content()`	Devuelve el contenido del nodo para nodos de tipo “contenedor” como nodos de tipo XplExpression devuelve el nodo de expresión específico. Nulo si no contiene ningún elemento.
`set_Value(string^)`	Establece el contenido del nodo al string pasado como argumento (sólo sirve para nodos simples de tipo XplNode).
`get_StringValue()`	Obtiene el valor string almacenado en el nodo. Sólo se sirve con nodos de tipo XplNode que contengan un valor de tipo string (los nodos simples pueden contener otros tipos de valores como enteros,

	flotantes o fechas).
`FindNodes("/n")`	Devuelve una lista de nodos (un XplNodeList) que coincidan con la cadena de búsqueda pasada como argumento. Vea más abajo para el formato de la cadena de búsqueda.
`FindNode("/n(id)")`	Devuelve el primer nodo encontrado que cumpla con el patrón pasado como argumento o nulo si no encuentra un nodo.
`CurrentNamespace`	Devuelve el espacio de nombre más inmediato que contiene al nodo actual o nulo si el nodo actual no se encuentra dentro de un espacio de nombres.
`CurrentClass`	Devuelve la clase más inmediata que contiene al nodo actual o nulo si no se encuentra dentro de una clase.
`CurrentFunction`	Devuelve la función más inmediata que contiene al nodo actual o nulo si no se encuentra dentro de una función.
`CurrentProperty`	Devuelve la propiedad más inmediata que contiene al nodo actual o nulo si no se encuentra dentro de una propiedad.
`CurrentBlock`	Devuelve el bloque más inmediato que contiene al nodo actual o nulo si no se encuentra dentro de un bloque.
`CurrentExpression`	Devuelve la expresión más inmediata que contiene al nodo actual o nulo si no se encuentra dentro de una expresión.
`ChildNodes()`	Devuelve una colección con todos los nodos hijos. Si el elemento no posee nodos hijos devuelve una colección vacia.
`string^[] Attributes()`	Devuelve una matriz con todos los atributos XML del tipo del elemento.
`string^ AttributeValue(string^)`	Devuelve el valor de el atributo indicado en el parámetro convertido a tipo string como es representado en XML cuando se almacena un documento Zoe.
`set_doc(string^) y get_doc()`	Establece y retorna un comentario simple asociado al elemento.
`string^ get_ldsrc()`	Devuelve un string indicando las líneas/columnas de inicio y/o fin en el fuente original. Puede devolver una cadena vacía en cuyo caso indica que el nodo no posee información de fuente de origen, en dicho caso deberá examinarse los nodos padres más cercanos.
`CodeDOMTypes get_TypeName()`	Devuelve un valor de enumeración con el nombre del tipo del nodo, por ejemplo “XplNode”, “XplClass”, etc. Puede utilizarlo opcionalmente a la identificación de tipos soportada por el lenguaje de alto nivel.
`XplNode^ Clone()`	Obtiene una copia profunda de un nodo.
`Write(XplWriter^)`	Guarda el nodo y todos los hijos usando la instancia

	de XplWriter recibida como argumento.
`XplNode^ Read(XplReader^)`	Carga en memoria el nodo a partir de la instancia de XplReader recibida como argumento.

La clase XplNodeList es utilizada por las clases del CodeDOM donde se requiere una lista de nodos de cualquier longitud y donde en general dicha lista puede poseer varios tipos de nodos del CodeDOM.

Los miembros más importantes de XplNodeList son:

Miembro	Descripción
`InsertAtEnd(XplNode^)`	Inserta un nodo al final de la lista. También existe una sobrecarga que acepta una XplNodeList como argumento, en cuyo caso inserta todos los nodos contenidos por la lista recibida.
`InsertAtBegin(XplNode^)`	Inserta un nodo al principio de la lista. Existe otra sobrecarga que acepta un XplNodeList.
`InsertBefore(XplNode^ reference, XplNode^ newNode)`	Inserta newNode antes del nodo reference. Si el nodo reference no existe no inserta el newNode. Existe otra sobrecarga que acepta un XplNodeList.
`InsertAfter(XplNode^ reference, XplNode^ newNode)`	Inserta newNode después del nodo reference. Si el nodo reference no existe no inserta el newNode. Existe otra sobrecarga que acepta un XplNodeList.
`FirstNode()`	Retorna el primer nodo de la lista e inicializa el iterador interno de la lista.
`NextNode()`	Obtiene el nodo siguiente en la iteración actual, si no existen más nodos (esta al final de la lista) retorna nulo.
`GetLength()`	Retorna la cantidad de nodos en la lista.
`XplNodeList::CopyNodesAtEnd(XplNodeList^ source, XplNodeList^ target)`	Copia los nodos dentro de la lista source al final de la lista target.

Utilizando instancias de las clases del CodeDOM se puede crear en memoria porciones de código fuente e incluso programas completos. Es preciso aclarar que todo código Zoe representado en memoria como un documento es implementado con las clases del CodeDOM, por ello es muy útil para todo programador de classfactorys conocer el CodeDOM.

4.2.2 La expresión "writecode"

En lugar de crear código fuente en memoria utilizando las clases del CodeDOM se puede utilizar la expresión "writecode". La expresión writecode toma como argumento una expresión o un bloque de código dependiendo de la forma utilizada, y genera en memoria el código pasado como argumento utilizando las clases del CodeDOM, es decir la expresión writecode realiza automáticamente el trabajo que deberíamos de hacer a "mano" utilizando las clases del CodeDOM. Adicionalmente la expresión writecode realiza reemplazos en su bloque argumento reemplazando identificadores marcados con "$" por el contenido de la variable nombrada.

La expresión writecode posee tres formas sintácticas diferentes:

Ejemplo	Descripción

`//Devuelve un objeto XplExpression` `writecode( $argumentExp + 5 * var )`	Forma de Expresión En la forma de expresión la instrucción writecode siempre devuelve un objeto de tipo "XplExpression^".

`//Devuelve un objeto XplClass`
`writecode{`
`  class $className{`
`  void func () { }`
`  }`
`}`
 	
Ó
 	
`//Devuelve un objeto XplDocumentBody`
`writecode{`
`  using Zoe;`
`  namespace test{`
 	
`  }`
`}`
 	
Ó
 	
`//Devuelve un objeto XplFunctionBody`
`writecode{`
`  int localvar = 0;`
`  for(int n=0; n<$arg.Count; n++){`
`  localvar++;`
`  }`
`}`
	Forma de Bloque simple
	En la forma de bloque simple la expresión writecode devuelve un objeto del tipo "XplClass^", "XplFunctionBody^" o "XplDocumentBody^" dependiendo del contenido del bloque argumento.

`//Devuelve un objeto XplClassMembersList`
`//Notar que se requiere "{%" y "%}" para`
`//el caso de generar miembros de clase`
`writecode{%`
`  int field1;`
`  float field2;`
`  void func($t1 arg1, int arg2) { }`
`%}`
	Forma de Bloque de miembros de Clase
	En esta forma sintáctica la expresión writecode devuelve un objeto del tipo "XplClassMembersList^" el cual contiene una lista de miembros de clase.

Para indicarle a la expresión writecode que deseamos reemplazar un identificador por el contenido de una variable debemos marcar el identificador con el signo de dólar al comienzo, por ejemplo "$miVariable". A continuación se muestra un ejemplo:

```
members =    writecode
{%
private:
       $fieldType $internalFieldName;
public:
       $fieldType property $fieldName{
              get{
                return $internalFieldName;
              }
              set{
                $internalFieldName = value;
              }
       }
%};
```

En el ejemplo superior "fieldType", "internalFieldName" y "fieldName" son variables en el alcance de la expresión writecode. Los tipos que se pueden reemplazar en una expresión writecode son: type o XplType^, block o XplFunctionBody^, XplExpression^ o tipos "exp", XplIName^ o tipos "iname", constantes.

4.2.3 Retornar expresiones y tipos desde una función miembro de classfactory

Para retornar expresiones desde un miembro de una classfactory se puede utilizar la instrucción “return” ordinaria como en cualquier otra función.

Ejemplos:

```
factory class Example{
public:
	static exp string^ Add(exp int arg1, exp int arg2){
		return writecode( “Result: ” + ($arg1+$arg2).ToString() );
	}
}
```

Para utilizar la función superior en un programa cliente lo hacemos como si se tratara de cualquier otra función:

```
Console::WriteLine( Example::Add(5, 1024) );
```

También es posible retornar expresiones utilizando variables del tipo XplExpression^. Por ejemplo:

```
factory class Example{
public:
	exp string^ Add(exp int arg1, exp int arg2){
		XplExpression^ resExp = null;
		resExp = writecode( “Result: ” + ($arg1+$arg2).ToString() );
		return resExp;
	}
}
```

El ejemplo superior muestra como los tipos declarados con el modificador “exp” son equivalentes a una referencia al tipo XplExpression.

Para retornar tipos a código cliente podemos utilizar los “Constructores de Tipos”. Los constructores de tipos se escriben igual que los constructores ordinarios pero se especifica que retornan un tipo, como se muestra a continuación:

```
factory class MyString {
public:
	type MyString(){
		return gettype(string^)
	}
}
```

Al utilizar el tipo MyString en un programa cliente el resultado será que cada ocurrencia de una declaración del tipo “MyString” se reemplazará por el tipo “string^”, ejemplo:

```
MyString myVar = “Hola Mundo”;
MyString[] myArray = new MyString[] = { “Zoe”, “Meta D++” };
```

El ejemplo superior es equivalente a escribir las siguientes líneas:

```
string^ myVar = “Hola Mundo”;
string^[] myArray = new string^[] = { “Zoe”, “Meta D++” };
```

Los constructores de tipos se utilizan para desarrollar classfactorys que encapsulen la diferencia de tipos entre diversas plataformas o entornos de ejecución. También existe una variante de los constructores de tipos que permite tomar argumentos, para más información consulte los manuales de Meta D++, Zoe o visite el sitio web http://layerd.net.

El tipo “type” es equivalente al tipo “XpType^”, como se muestra a continuación:

```
factory class MyType{
	static XplType^ refType;
public:
	static exp void SetType(type argType){
		refType = argType;
		return null;
	}
	type MyType(){
		if(refType==null)
			return gettype(int);
		else
			return refType;
	}
}
```

El ejemplo superior retorna entero si no se estableció un tipo llamando a SetType, de lo contrario retorna el tipo establecido en SetType.

4.2.4 El objeto “context”

El objeto “context” representa el contexto de llamada actual en un miembro de una classfactory. El objeto context es de tipo XplNode por tanto pueden accederse a todos los miembros de dicha clase.

Ejemplo:

La classfactory:

```
namespace Zoe::Examples{

factory class ContextSensitive{
public:
	exp void WhereAmI(){
		if(context.CurrentNamespace!=null && context.CurrentClass==null)
			Console::WriteLine( “You are on a namespace body.” );
		if(context.CurrentClass!=null && context.CurrentFunction==null
			&& context.CurrentProperty==null)
			Console::WriteLine( “You are on a class body.” );
		if(context.CurrentBlock!=null)
			Console::WriteLine( “You are on a block.” );
		return null;
	}
}

}
```

El cliente:

```
using Zoe::Examples;

namespace N{
	ContextSensitive::WhereAmI();
	class A{
		ContextSensitive::WhereAmI();
		void Func(){
			ContextSensitive::WhereAmI();
		}
	}
}
```

4.2.5 El objeto "currentDTE"

El objeto "currentDTE" representa el fuente actual y refiere al "Documento de Tiempo de Ejecución" del tiempo de compilación en curso. El contenido del objeto currentDTE es la suma de todos los fuentes que componen el programa que se está compilando actualmente, por tanto al realizarse análisis o búsquedas en dicho objeto estamos realizando dichas acciones en el código de todo el programa, sin embargo esto no incluye los documentos importados por los generadores de código Zoe (por ejemplo los tipos importados usando la directiva import).

4.2.6 El objeto "compiler"

El objeto "compiler" representa la instancia actual del compilador Zoe. Con el objeto compiler es posible consultar los errores y advertencias del proceso de compilación actual, también es posible agregar nuevos errores o advertencias y consultar la información de tipos recopilada durante el análisis semántico. Ejemplo:

```
// add persistent compilation error
Error^ newError = new Error("Error processing xml file. " + error.Message,context);
newError.set_PersistentError(true);
compiler.Errors.AddError(newError);
```

5 Ejemplos de programación con Classfactorys

5.1 Classfactorys de Menor a Mayor

La siguiente guía mostrara una serie de ejemplos partiendo desde lo más sencillo a temas más complejos. En todos los casos se tendrán dos archivos. Uno para la classfactory y otro para el cliente de manera de mostrar el consumo y prueba.

Para probar los ejemplos primero se deberá compilar la classfactory y agregarla a la librería de classfactorys usando el modificador "-ae" del compilador Zoe, y luego compilar el cliente.

5.1.1 Hola Mundo con una Classfactory

El código de la classfactory es la siguiente:

```
using Zoe;
using DotNET::LayerD::ZoeCompiler;
using DotNET::LayerD::CodeDOM;

namespace Test{
	public factory class HelloWorld{
	public:
		static exp string^ SayHello(){
			return writecode( "Hello World" );
		}
	}
}
```

El código del cliente:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace Test {
	public class App{
	public:
		static void Main(string^[] args){
			Console::WriteLine( HelloWorld::SayHello() );
		}
	}
```

```
}
```

En lo subsiguiente se omitirán las instrucciones Using, Import y la declaración del espacio de nombres, pero deberá incluirse en todos los ejemplos para poder compilarlos, aquí se los omitirá por comodidad.

Para compilar el ejemplo anterior guarde el código de la classfactory en un archivo (aquí "TestTemplates.dpp") y el cliente en otro archivo ("Test.dpp") luego compile con los siguientes comandos:

```
c:\layerd\bin>metadppc.exe TestTemplates.dpp
c:\layerd\bin>zoec.exe TestTemplates.zoe -ae

c:\layerd\bin>metadppc.exe Test.dpp
c:\layerd\bin>zoec.exe Test.zoe
```

De la classfactory destacamos:

```
static exp string^ SayHello(){
       return writecode( "Hello World" );
}
```

Las palabras reservada "exp" antes de la declaración de tipos indicamos que en realidad queremos especificar una expresión de determinado tipo, en este caso indicamos que queremos devolver una expresión de tipo "string^", es decir una referencia a string o lo que es lo mismo un equivalente al tipo de datos string en C# o Java – lenguajes en los cuales siempre son por referencia las cadenas – luego la palabra reservada "writecode" crea en memoria una porción de código – en este caso una expresión – que finalmente es retornada por la instrucción return.

El resultado es que al ejecutarse la llamada a SayHello durante el tiempo de compilación del programa cliente se retornara la expresión `"Hello World"` y se insertará en el lugar de la llamada a función.

5.1.2 Classfactorys y Compilación Interactiva de programas LayerD

Para aprender a programar Classfactorys y utilizar las características de tiempo de compilación de LayerD es útil conocer una versión especial de classfactorys denominadas "interactivas".

Las llamadas classfactorys interactivas son ejecutadas cuando compilamos un programa cliente en "modo interactivo", el modo de compilación interactivo de un programa LayerD funciona de la siguiente forma:

- El compilador Zoe compila el programa normalmente y ejecuta en tiempo de compilación todas las llamadas a Classfactorys Interactivas.
- Al resultado del primer tiempo de compilación lo convierte al código fuente LayerD de alto nivel sobre-escribiendo el fuente original.

La utilidad y aplicación de las classfactorys interactivas es muy variada. Ya que permite programar comportamiento muy complejo que normalmente se encuentran en los IDEs de los lenguajes como refactoring, análisis de código, utilización de snippets de código, ejemplos, asistentes, diseñadores visuales, etc. Pero lo más importante es que nos permite programar todas estas cosas de manera independiente al IDE, en nuestro propio lenguaje de alto nivel y si queremos independiente de la plataforma utilizando las características de LayerD. También cabe recordar que las classfactorys interactivas pueden ser consumidas por cualquier lenguaje LayerD cliente que soporte generación inversa (de código Zoe al código del meta lenguaje) como por ejemplo Meta D++.

Una classfactory interactiva es prácticamente idéntica a una classfactory ordinaria, simplemente debe cambiarse en la declaración el modificador “factory” por el modificador “interactive”, ej:

```
using Zoe;
using DotNET::LayerD::ZoeCompiler;
using DotNET::LayerD::CodeDOM;

namespace Test{
	public interactive class HelloWorld{
	public:
		static exp string^ SayHello(){
			return writecode( “Hello World” );
		}
	}
}
```

Luego se compila la extensión como cualquier otra classfactory, por ejemplo:

```
c:\layerd\bin>metadppc.exe TestTemplates.dpp
c:\layerd\bin>zoec.exe TestTemplates.zoe –ae
```

En los programas cliente se las utilizan de forma idéntica a una classfactory ordinaria:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace Test {
	public class App{
	public:
		static void Main(string^[] args){
			Console::WriteLine( HelloWorld::SayHello() );
		}
	}
}
```

Luego sólo debemos ejecutar el compilador Zoe en “modo interactivo” – utilizando la opción `“-i”` del compilador – al compilar el programa cliente:

```
c:\layerd\bin>metadppc.exe Test.dpp
c:\layerd\bin>zoec.exe Test.zoe –i
```

Como resultado de la ejecución de la classfactory interactiva anterior nuestro programa cliente será modificado como sigue:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace Test {
	public class App{
	public:
		static void Main(string^[] args){
			Console::WriteLine( “Hello World” );
		}
	}
}
```

Si asociamos el proceso de compilación interactiva a un macro de nuestro IDE preferido será muy sencillo llamarlo usando un atajo de teclado.

En los ejemplos siguientes resultara muy útil para el estudiante utilizar la compilación interactiva para ir verificando el funcionamiento de las classfactory. También es siempre útil utilizar este modo de compilación para depurar classfactorys complejas que estemos desarrollando (téngase en cuenta que es posible depurar

paso por paso la ejecución de classfactorys si es necesario utilizando depuradores estándar para la plataforma en la cual se ejecute el compilador Zoe).

A modo de recordatorio es importante comprender que no es una buena práctica de programación abusar de classfactorys interactivas para inyectar código que después modificaremos "a mano", precisamente la función de las classfactorys (ordinarias, no las interactivas) es encapsular código y permitir al programador de alto nivel abstraerse de las complejidades de la implementación. Por tanto, sólo deben usarse classfactorys interactivas para desarrollar componentes tipo RAD útiles, como ejemplos de utilización de librerías, asistentes para clases, refactoring de código, diseñadores visuales y similares utilidades que normalmente estarían enlazadas y dependientes de un IDE en particular. Consulte otros documentos en http://layerd.net para más discusiones e información sobre classfactorys interactivas.

5.1.3 Hola Mundo con argumento constante

El código de la classfactory es la siguiente:

```
public factory class HelloWorld{
public:
	static exp string^ SayHello(string^ constantArgument){
		return writecode( "Hello World" + $constantArgument );
	}
}
```

El código del cliente:

```
public class App{
public:
	static void Main(string^[] args){
		Console::WriteLine( HelloWorld::SayHello( " from Argentina." ) );
	}
}
```

Compile el ejemplo de la misma manera y corra el programa cliente.

El cambio desde el ejemplo anterior es muy simple y se limita a agregar un argumento a la función `SayHello` de tipo `string^` y utilizarlo en la expresión `writecode` con el prefijo `"$"`. De esa forma es posible pasar un valor constante a la función de la classfactory cuando es llamada. En las expresiones `writecode` al utilizar identificadores precedidos por `"$"` estamos indicando que queremos reemplazar el identificador por el contenido de una variable, si dicha variable no existen en el alcance o no es de un tipo indicado se generará un error al intentar compilar la classfactory.

El resultado es que al ejecutarse la llamada a SayHello durante el tiempo de compilación del programa cliente se retornara la expresión `"Hello World from Argentina."` y se insertará en el lugar de la llamada a función.

Un concepto importante que se desprende del ejemplo se aprende al intentar en el programa cliente lo siguiente:

```
public class App{
public:
	static void Main(string^[] args){
		Console::WriteLine( HelloWorld::SayHello( " from Argentina." + args[0] ) );
	}
}
```

Al intentar compilar el programa cliente superior se producirá un error indicando que no es posible evaluar la expresión `" from Argentina." + args[0]`, lo cual es correcto ya que debe recordarse que las llamadas a miembros de classfactorys se procesan en tiempo de compilación – esto es mientras se está compilando el programa – y por tanto no es posible evaluar el valor de expresiones que posean valores "de tiempo de ejecución" como en este caso la variable "args". Nótese que sencillamente no se posee un valor para args ya que el programa todavía no se ejecuto y por tanto no es posible determinar un valor para el argumento de la función `SayHello`.

La solución al problema planteado por el programa cliente se muestra en el siguiente ejemplo.

5.1.4 Hola Mundo con una expresión como argumento

El código de la classfactory es la siguiente:

```
public factory class HelloWorld{
public:
	static exp string^ SayHello(exp string^ argument){
		return writecode( "Hello World" + $argument );
	}
}
```

El código del cliente:

```
public class App{
public:
	static void Main(string^[] args){
		Console::WriteLine( HelloWorld::SayHello( " from Argentina." ) );
	}
}
```

La única diferencia con la classfactory anterior es la palabra reservada "exp" en la declaración del parámetro de la función SayHello:

```
	static exp string^ SayHello(exp string^ argument){
		return writecode( "Hello World" + $argument );
	}
```

Dicho cambio en la declaración del parámetro nos permite tomar como argumento expresiones sin evaluar completamente, luego podemos utilizarlas como objetos del tipo XplExpression o en expresiones writecode para generar código en memoria. Ahora el programa cliente anterior será correcto:

```
public class App{
public:
	static void Main(string^[] args){
		Console::WriteLine( HelloWorld::SayHello( " from Argentina." + args[0] ) );
	}
}
```

El resultado sería igual a escribir el siguiente código:

```
public class App{
public:
	static void Main(string^[] args){
		Console::WriteLine( "Hello World" + " from Argentina." + args[0] ) );
	}
}
```

Debe observarse que se retorna una expresión completa y no un valor. Luego, al ejecutar el programa cliente una vez compilado la expresión resultante podrá evaluarse normalmente. Puede comprobarse el resultado cambiando la classfactory a interactiva y compilando en modo interactivo con el compilador Zoe.

5.1.5 Pasando bloques como argumento

Las funciones miembros de classfactorys pueden recibir como argumento bloques de instrucciones desde el código cliente. Para esto basta con declarar el último parámetro de una función con el tipo "block". Luego la variable de tipo "block" en la función puede ser tratada como una variable de tipo XplFunctionBody^.

El código de la classfactory es la siguiente:

```
public factory class MyTools{
public:
	static XplNode^ Repeat(exp int^ repeatCount, block argumentBlock){
		return writecode{
			for( int n=0 ; n < $repeatCount ; n++ )
				$argumentBlock;
		};
	}
}
```

El código del cliente:

```
public class App{
public:
	static void Main(string^[] args){
		MyTools::Repeat(10){
			Console::WriteLine( "Hola Mundo." );
		};
	}
}
```

Tanto el código de la classfactory como el código cliente poseen particularidades que es preciso describir:

```
public factory class MyTools{
public:
	static XplNode^ Repeat(exp int^ repeatCount, block argumentBlock){
		return writecode{
			for( int n=0 ; n < $repeatCount ; n++ )
				$argumentBlock;
		};
	}
}
```

El parámetro `"argumentBlock"` se declara en último lugar, esto es preciso hacerlo así o de lo contrario es un error, tampoco puede usarse un argumento de tipo `"XplFunctionBody^"` ya que aunque dicho tipo sea internamente equivalente en dicho caso se interpretará que se desea tomar como argumento una referencia a un valor de dicho tipo y no un bloque de instrucciones per se. La función `Repeat` retorna en este caso una referencia a `XplNode` - la clase base de todos los nodos dentro del `LayerD::CodeDOM` – esto es porque en este caso no queremos retornar una expresión sino algún otro tipo de código, en éste caso un bloque de instrucciones. Finalmente las variables de tipo `"block"` y `"XplFunctionBody^"` pueden utilizarse dentro de expresiones `writecode` como argumentos del template utilizando `"$"` al igual que las variables de tipo expresión.

En el cliente remarcamos como se invoca a una función que toma como último argumento un bloque de instrucciones:

```
public class App{
```

```
public:
	static void Main(string^[] args){
		MyTools::Repeat(10){
			Console::WriteLine( "Hola Mundo." );
		};
	}
}
```

Para pasar un bloque como argumento desde el código cliente simplemente debemos escribir las instrucciones entre llaves. Note el punto y coma al final de la llave, ello indica que `"Repeat"` es una función de una classfactory y no una instrucción nativa del lenguaje Meta D++. Es recomendable probar éste ejemplo en modo interactivo para ver el resultado más claramente.

5.1.6 Mejora de la función Repeat

La función `Repeat` del ejemplo anterior funciona, sin embargo posee una limitación que es preciso aclarar y en la práctica corregir.

Supongamos el siguiente programa cliente:

```
public class App{
public:
	static void Main(string^[] args){
		MyTools::Repeat(10){
			Console::WriteLine( "Hola Mundo." );
			MyTools::Repeat(5){
				Console::WriteLine( "Inner Loop." );
			};
		};
	}
}
```

El programa anterior – por el funcionamiento de la classfactory `MyTools` – es equivalente al siguiente programa:

```
public class App{
public:
	static void Main(string^[] args){
		for( int n = 0 ; n < 10 ; n++ ){
			Console::WriteLine( "Hola Mundo." );
			for( int n = 0 ; n < 5 ; n++ ){
				Console::WriteLine( "Inner Loop." );
			}
		}
	}
}
```

En este caso el error es evidente, la classfactory siempre retorna el ciclo for usando la variable de iteración `"n"` lo cual causará un error de compilación. Por tanto muchas veces necesitamos generar nombres de identificadores aleatorios y únicos que podamos usar en expresiones `writecode` para la transformación/generación de código.

Para obtener identificadores únicos podemos usar dos métodos:

- El tipo `XplIName`
- El prefijo `$$` en identificadores

Utilizando el tipo `XplIName` el problema del ejemplo anterior se soluciona de la siguiente forma (se resaltaron los cambios):

```
public factory class MyTools{
public:
	static XplNode^ Repeat(exp int^ repeatCount, block argumentBlock){
		XplIName^ myID = new XplIName();
		return writecode{
			for( int $myID=0 ; $myID < $repeatCount ; $myID++ )
				$argumentBlock;
		};
	}
}
```

Por tanto es preciso crear una instancia de `XplIName` usando el constructor sin argumentos para generar un nuevo identificador único. Usando el prefijo `$$` en identificadores se soluciona como sigue:

NOTA : El modificador $$ en identificadores no está implementado en la distribución del compilador Zoe que acompaña él presente documento. Por tanto no podrá utilizarse dicha funcionalidad.

```
public factory class MyTools{
public:
	static XplNode^ Repeat(exp int^ repeatCount, block argumentBlock){
		return writecode{
			for( int $$myID=0 ; $$myID < $repeatCount ; $$myID++ )
				$argumentBlock;
		};
	}
}
```

Ambos métodos dan exactamente el mismo resultado, el prefijo `$$` reutiliza el identificador generado cuando se repite el identificador que sigue al los signos `$$`. Ejemplo:

```
public factory class MyTools{
public:
	static XplNode^ Repeat(exp int^ repeatCount, block argumentBlock){
		return writecode{
			for( int $$myID=0 ; $$myID < $repeatCount ; $$myID++ )
				for( int $$otherID=0 ; $$otherID < $repeatCount ; $$otherID++ )
					$argumentBlock;
		};
	}
}
```

En cualquiera de las dos soluciones posibles mostradas anteriormente se utilizará un nuevo identificador en cada llamada a la función `Repeat`. Experimente con el modo interactivo para comprender como funciona internamente éste mecanismo.

5.1.7 Retornar una clase desde una classfactory

El código de la classfactory es la siguiente:

```
public factory class MyTools{
public:
	static XplNode^ MakeClass(){
		return writecode{
			public class TestClass{
			public:
				int DoSomething(){
					return 5;
				}
			}
		};
	}
}
```

El código del cliente:

```
namespace Test {
	MyTools::MakeClass();
	public class App{
	public:
		static void Main(string^[] args){
			TestClass^ var = new TestClass();
			Console::WriteLine( var.DoSomething().ToString() );
		}
	}
}
```

Para remarcar en el código de la classfactory tenemos los siguientes:

```
public factory class MyTools{
public:
	static XplNode^ MakeClass(){
		return writecode{
			public class TestClass{
			public:
				int DoSomething(){
					return 5;
				}
			}
		};
	}
}
```

En este caso la expresión `writecode` genera el código de una clase completa y lo retorna. Al valor retornado por `writecode` también podríamos guardarlo en una variable de tipo `XplClass^` en lugar de directamente retornarlo. La función retorna un `XplNode^` pero también podría retornar directamente un `XplClass^`.

El código cliente muestra una particularidad importante de las llamadas a classfactorys:

```
namespace Test {
	MyTools::MakeClass();
	public class App{
	public:
		static void Main(string^[] args){
			TestClass^ var = new TestClass();
			Console::WriteLine( var.DoSomething().ToString() );
		}
	}
}
```

Las llamadas a classfactorys en programas clientes pueden encontrarse fuera del cuerpo de funciones, esto incluye cuerpo de clases, cuerpo de espacio de nombres y directamente en el cuerpo del programa.

5.1.8 Mejora de la inyección de una clase:

Se mejorará el ejemplo anterior. El código de la classfactory es la siguiente:

```
public factory class MyTools{
public:
	static XplNode^ MakeClass(){
		if(context.CurrentBlock!=null ||
		  context.CurrentNamespace==null){
			compiler.Errors.Add(
				new Error ( "No puede utilizar MakeClass fuera del cuerpo de
una clase o espacio de nombres." , context )
				);
			return null;
		}
		return writecode{
```

```
                public class TestClass{
                public:
                        int DoSomething(){
                                return 5;
                        }
                };
        }
}
```

El código del cliente quedará igual que al anterior por ahora. Los cambios que necesitan explicación en la classfactory son los siguientes:

```
public factory class MyTools{
public:
        static XplNode^ MakeClass(){
                if(context.CurrentBlock!=null ||
                  context.CurrentNamespace==null){
                        compiler.Errors.Add(
                                new Error ( "No puede utilizar MakeClass fuera del cuerpo de
una clase o espacio de nombres." , context )
                                );
                        return null;
                }
                return writecode{
                        public class TestClass{
                        public:
                                int DoSomething(){
                                        return 5;
                                }
                        }
                };
        }
}
```

El bloque `if` lo que hace es emitir un error cuando la función `MakeClass` es llamada fuera de un cuerpo de una clase o un espacio de nombres, esto es porque no es posible insertar una clase dentro de una función o dentro del cuerpo del programa fuera de un espacio de nombres. Por tanto, el bloque `if` garantiza que la función `MakeClass` sea utilizada en el contexto adecuado.

La llamada a `"compiler.Errors.Add"` utiliza el objeto especial `"compiler"` – disponible en classfactorys – para agregar un error de compilación al programa cliente actual cuando la función `MakeClass` no es utilizada de forma adecuada. El primer argumento del constructor de `Error` es un `string` con el mensaje de error, el segundo argumento debe ser una instancia de `XplNode` o una clase derivada que indicará el nodo del programa dentro del cual se produjo el error, en este caso pasamos el nodo `context` indicando que el error está en la llamada a función `MakeClass` en el cliente. El objeto `compiler` permite examinar los errores actuales, agregar nuevos errores como en el ejemplo, advertencias o notificaciones, también podemos consultar los tipos evaluados en el último análisis semántico ejecutado antes del tiempo de compilación actual.

5.1.9 Tomar tipos como argumentos

En general no querremos inyectar en el código cliente clases determinadas de forma estática sino que su estructura variara de acuerdo a ciertos parámetros.

A continuación se muestra como tomar como argumento un tipo de datos en una classfactory. El código de la classfactory es el siguiente:

```
public factory class MyTools{
```

```
public:
	static XplNode^ MakeClass(type typeArgument){
		return writecode{
			public class TestClass{
				$typeArgument back;
			public:
				$typeArgument DoSomething($typeArgument argument){
					back += argument;
					return back;
				}
			}
		};
	}
}
```

El código del cliente:

```
namespace Test {
	MyTools::MakeClass( gettype(int) );
	public class App{
	public:
		static void Main(string^[] args){
			TestClass^ var = new TestClass();
			Console::WriteLine( var.DoSomething(15).ToString() );
			Console::WriteLine( var.DoSomething(12).ToString() );
		}
	}
}
```

En la classfactory sobresalen los cambios siguientes:

```
public factory class MyTools{
public:
	static XplNode^ MakeClass(type typeArgument){
		return writecode{
			public class TestClass{
				$typeArgument back;
			public:
				$typeArgument DoSomething($typeArgument argument){
					back += argument;
					return back;
				}
			}
		};
	}
}
```

El tipo especial "`type`" indica que la función toma como argumento un tipo, luego lo podemos usar como un tipo en expresiones `writecode` utilizando el prefijo "`$`" al igual que con identificadores, expresiones y bloques. Los parámetros de tipo "`type`" en tiempo de ejecución de la classfactory son equivalentes al tipo `XplType^`.

En el cliente utilizamos la expresión "`gettype`" para pasar como argumento cualquier tipo:

```
	MyTools::MakeClass( gettype(int) );
```

Con los parámetros tipo `type` es posible implementar tipos y funciones genéricas en LayerD de forma independiente al entorno de ejecución y la plataforma que usemos para el despliegue de nuestro software.

5.1.10 Retornar miembros de clase

El ejemplo muestra como retornar un miembro de clase desde una classfactory. El código de la classfactory es el siguiente:

```
public factory class MyTools{
public:
	static XplNode^ MakeFunction(type typeArgument){
		return writecode{%
			$typeArgument DoSomething($typeArgument argument){
				return argument + argument;
			}
		%}.Children().FirstNode();
	}
}
```

El código del cliente:

```
namespace Test {
	public class App{
		MyTools::MakeFunction( gettype(int) );
	public:
		static void Main(string^[] args){
			TestClass^ var = new TestClass();
			Console::WriteLine( var.DoSomething(15).ToString() );
		}
	}
}
```

5.2 Una Classfactory simple

Supongamos que debemos escribir multiples clases y declarar sus atributos usando campos privados y propiedades, todos con una funcionalidad similar. Podemos trabajar utilizando orientación a objetos tradicional y escribirlo de la siguiente forma:

```
public class Customer{
  string^ name, lastName;
  int age;
public:
  string^ property Name{
	get{
		return name;
	}
	set{
		name = value;
	}
  }
  string^ property LastName{
	get{
		return lastName;
	}
	set{
		lastName = value;
	}
  }
  int property Age{
	get{
		return age;
	}
	set{
		if(value<0){
		  Console::WriteLine("The value must be greater than 0.");
		}
		else{
		  age = value;
		}
```

```
        }
    }
}
public class Employee{
    string^ name, lastName;
    int employeeId;
public:
    string^ property Name{
        get{
            return name;
        }
        set{
            name = value;
        }
    }
    string^ property LastName{
        get{
            return lastName;
        }
        set{
            lastName = value;
        }
    }
    int property EmployeeId{
        get{
            return employeeId;
        }
        set{
            if(value<0){
              Console::WriteLine("The value must be greater than 0.");
            }
            else{
              employeeId = value;
            }
        }
    }
}
```

El ejemplo muestra un caso tipico en el cual necesitamos definir multiples clases siguiendo un patrón para los campos y su exposición al exterior usando propiedades. En LayerD podemos encapsular el patrón en una classfactory y evitar tener que repetir código de forma manual o requerir de la ayuda de generadores de código.

A continuación, se muestra el código de una classfactory que encapsula el patrón del ejemplo.

```
using Zoe;
// Creo los imports por defectos necesarios
Utils::DefaultNetImports();

namespace Test {
  public factory class Model{
  public:
    static exp void Field(iname void fieldName, type fieldType){
      XplIName^ internalFieldName = new XplIName();
       XplClassMembersList^ miembros = null;
       // Creo en memoria el código que quier injectar
       if(fieldType.get_typename()==NativeTypes::Integer){
              miembros =   writecode
              {%
              private:
                     $fieldType $internalFieldName;
              public:
                     $fieldType property $fieldName{
                           get{
                              return $internalFieldName;
```

```
                        }
                        set{
                           if(value<0)
                           Console::WriteLine("Insert a value > 0.");
                           $internalFieldName = value;
                         }
                    }
            %};
        }
        else{ // Si no es entero
            miembros =   writecode
            {%
            private:
                    $fieldType $internalFieldName;
            public:
                    $fieldType property $fieldName{
                           get{
                             return $internalFieldName;
                           }
                           set{
                             $internalFieldName = value;
                           }
                    }
            %};
        }
        //Lo inyecto al codigo donde yo quiero.
        context.CurrentClass.Children().InsertAtEnd( miembros.Children() );
        return null;
    }
  }
}
```

Para programar una Classfactory (procesamiento de tiempo de compilación) se debe declarar una clase con el modificador “factory” (también puede utilizarse el modificador “interactive” para “Classfactorys Interactivas”).

Completado del presente documento en curso. Visite periódicamente los sitios del proyecto LayerD para estar al tanto de versiones actualizadas.