Seminario de Lenguajes de Programación.

DSL C# 4.0

Dayrene Fundora González
Yamilé Reinoso Díaz
Camilo González Hurtado
Gelin Equinosa Rosique
Rafael Horrach Santiago

Equipo 8

27 de marzo de 2020

1. DSL

Lenguaje de Dominio Específico(DSL) es un lenguaje de programación de expresividad limitada orientado a un dominio particular. Es un lenguaje especializado usado en un propósito especifico, se especializa en modelar o resolver un conjunto determinado de problemas, conocido como dominio.

Lenguajes como Java, Visual Basic, C, C++, C# son Lenguajes de Propósito General(GLP), los cuales son usados para cualquier cantidad de propósitos para resolver varios tipos de problemas. Pero hay muchas situaciones en donde para los GLP no todos los problemas son igualmente sencillos de resolver. *En la programación como en la construcción, la herramienta correcta hace una gran diferencia.*

1.1. Tipos de DSL

Después de leer esto uno puede pensar que los DSL se encargan de resolver problemas "técnicos" de Lenguajes Generales, pero no es así; según su enfoque a la hora de implementarlos hay diferentes tipos de categorías.

1.1.1. DSL Externo

Los DSL externos son un lenguaje en si mismo, independiente de cualquier otro lenguaje, el cual tienen su propia sintaxis y reglas para escribir código en él. Ejemplos comunes de estos pueden ser SQL usado para el trabajo con bases de datos, HTML y CSS usados en conjunción para el trabajo con paginas webs.

1.1.2. DSL Interno

Los DSL internos se define a partir de un lenguaje de programación base mediante la utilización de sus características.

Este DSL no es un lenguaje construido desde cero, sino que se basa en otro lenguaje existente. Se apoya en las construcciones sintácticas del lenguaje huésped y debe cumplir con las reglas del mismo. LINQ es un DSL interno donde la sintaxis que usamos es una sintaxis C# valida, pero representa una sintaxis para un dominio extendido y mas especifico. Otro ejemplo es Rails , que es un DSL basado el lenguaje de proposito general Ruby.

1.2. Ventajas y Desventajas

Los DSL permiten expresar soluciones usando los términos y el nivel de abstracción apropiado para el dominio del problema. Pueden mejorar la correctitud del desarrollo de la aplicación y ser una herramienta muy potente para crear un código *self-documented*. Con los DSL se pueden combinar múltiples paradigmas de la programación, ademas de poder disminuir el ruido sintáctico y tener un alto nivel de abstacción.

A pesar del bajo costo final de todo el desarrollo, el costo inicial de diseñar, implementar y mantener un DSL y las herramientas para trabajar con él es siempre visto como una desventaja. Existe un costo de aprender un nuevo lenguaje para algo con una aplicación limitada por su dominio específico.

1.3. Código

Se quiere diseñar un DSL interno en C# 3.5 que permita la definición de personas con atributos: FirstName, LastName. Como muestra este código.

```
//Accediendo directamente a los atributos
var p1 = Factory.New.Person;
p1.FirstName = "Yami";
p1.LastName = "Reynoso";
```

Fueron creadas tres clases, *Factory*, *CreatePerson* y *Person*.La clase *Factory* tiene una propiedad *New*, como se puede observar en la figura, para así poder acceder a esta mediante el *dot notation*.

```
public class Factory
    {
        public static CreatePerson New
        {
            get { return new CreatePerson(); }

        }
    }
```

La propidad *New* de *Factory* es de tipo *CreatePerson*. Por lo que al crear esta clase se añadio la propiedad *Person* para de la misma forma acceder directamente a los atributos.

```
public class CreatePerson
    {
        public Person Person
        {
            get { return new Person(); }

        }
    }
```

La clase *Person*, es la contenedora de las propidades *FirstName* qy *LastName*.

```
public class Person
    {
        public Person()…
        public Person(string FNa, string LNa)…
        public string FirtName …
        public string LastName …
        public string this[string index]…

    }
```

2. Fluent-Programming

Es un patrón de diseño con el que se escribe código de forma más legible y parecida al lenguaje común. Permite la manipulación de objetos mediante el uso de *dot notation* donde cada llamado transmite el contexto al siguiente. Utiliza el *Method Chaining* herramienta en la cual todo método retorna una referencia a "this" cuando termina el *query*, lo que permite que todos los métodos sean modificados en un simple "statement".

Permite retransmitir un contexto a través de llamadas subsecuentes. La esencia de *fluent programming* es que la salida de una operación (o método) es la entrada de la siguiente, creando una cadena de operaciones que conforman una sentencia de código que puede resultar muy legible. Con una correcta elección de los nombres de los métodos es posible programar de una forma muy similar al lenguaje natural.

Este patrón de diseño es mas fácil de entender, pero su uso suele ser menos atractivo para personas acostumbradas a programar por el hecho de que hacer un *debuggin* se vuelve mas complicado.

Suele usarse en lenguajes poco flexibles. En C# con LINQ se convierte un pequeño grupo de operaciones en un mismo contexto.

2.1. Código

Se quiere en lugar de acceder a los atributos directamente, como en el código anterior, poder inicializar mediante una *"fluent interface"*.

```
//Inicializando mediante una "fluent interface"
var p3 = Factory.New.Person.FirstName("Yami").LastName("Reynoso") ;
```

Los métodos extensores permiten *"agregar"* métodos a los tipos existentes sin necesidad de crear un nuevo tipo derivado y volver a compilar, o sin necesidad de modificar el tipo original. Su primer parámetro especifica la clase que extiende, y este está precedido por el modificador "this".

```
static class ExtendedPerson
   {
       public static Person FirstName(this Person p, string FirstName)
       {
           p.FirstName = FirstName;
           return p;
       }
       public static Person LastName(this Person p, string LastName)
       {
           p.LastName = LastName;
           return p;
       }
   }
```

3. Named-Parameters

Los Parametros Nombrados son una sintaxis alternativa de C# 4.0 para pasar parámetros a una función.

Para utilizarlo se escribe el nombre formal del parámetro seguido de *":"* y luego el valor a asignar.

```
Factory.New.Person(FirstName: "Louis", LastName: "Dejardin");
```

Usa variables locales temporales para la asignación de los parámetros, anteriormente hacia el llamado a funciones más lento que de la forma clásica, pero en las nuevas versiones ya no hay esta penalización.

Nos brindan una mayor legibilidad del código y nos permite cambiar el orden en que pasamos los argumentos.

```
PrintOrderDetails(productName:"Gift Shop", orderNum: 31, sellerName:"Red Mug");
```

Los argumentos con nombre liberan de la necesidad de recordar o buscar el orden de los parámetros de la lista de parámetros de los métodos llamados.

Permite reordenar el paso de parámetros a funciones como desee el programador, especificando de manera explícita a que parámetro hace referencia cada valor de la función.

```
PrintOrderDetails(orderNum: 31, sellerName:"Red Mug", productName:"Gift Shop");
```

3.1. Código

Se quiere hacer algo similar a los incisos anteriores, pero se quiere acceder a los atributos de persona de manera similar a JSON, para lo que usamos *"parametros nombrados"*

```
//Con notación similar a JSON
var p3 = Factory.New.Person.(FirstName:"Yami" ,LastName:"Reynoso") ;
```

Para hacer esta notación utilizamos nuevamente métodos extensores para usar las propiedades de persona como método y de esta forma pasar los argumentos que serán nombrados.

```
 static class CreateExtendedPerson
    {
        public static Person Person(this CreatePerson cp , string FirstName,
                                    string LastName)
        {
           return new Person(FirstName, LastName) ;
        }
}
```

4. Dynamic

Una de las novedades más relevantes de .NET Framework 4.0 es el soporte para el tipado dinámico con la inclusión del Dynamic Language Runtime (DLR) y el especificador dynamic en C# 4.0.

Dynamic se trata de un tipo estático, pero un objeto de tipo *dynamic* omite la comprobación en tiempo de compilación realizada a los tipos estáticos. En la mayoría de los casos, funciona como si se tuviera el tipo *object*.

En tiempo de compilación, se supone que un elemento con tipo *dynamic* admite cualquier operación. Por consiguiente, no tendrá que preocuparse de si el objeto obtiene su valor de una API de COM, de un lenguaje dinámico como IronPython, del Document Object Model (DOM) HTML, de la reflexión o de otro lugar en el programa. Pero si el código no es válido, los errores se detectan en tiempo de ejecución.

Dynamic proporciona una clase base para especificar el comportamiento dinámico en tiempo de ejecución, nombrada *DynamicObject*. No se puede crear directamente una instancia de dicha clase. Para implementar el comportamiento dinámico, puede que desee heredar de la clase *DynamicObject* e invalidar los métodos necesarios. Por ejemplo, si solo necesita operaciones para establecer y obtener propiedades, puede invalidar solo los métodos TrySetMember y TryGetMember. Lo cual explicaremos a profundidad más adelante.

La clase DynamicObject permite definir qué operaciones se pueden realizar en objetos dinámicos y cómo realizar esas operaciones. Por ejemplo, puede definir lo que ocurre cuando se intenta obtener o establecer una propiedad de objeto, llamar a un método o realizar operaciones matemáticas estándar, como suma y multiplicación. Esta clase puede ser útil si desea crear un protocolo más cómodo para una biblioteca. Si los usuarios de la biblioteca tienen que usar una sintaxis como *Scriptobj.SetProperty("Count", 1)*, puede proporcionar la capacidad de usar una sintaxis mucho más sencilla, como scriptobj.Count = 1, siendo scriptobj de tipo dynamic.

4.1. Diferencias entre var, dynamic y object

La palabra clave *var* declara una variable cuyo tipo es inferido a partir de la expresión que se le asigna por lo que no podemos declarar una variable *var* sin asignarle expresión (lógico puesto que el compilador no sabría el tipo de dicha variable) y que tampoco se puede declarar asignándole *null* (ya que null no tiene tipo). Entonces *var*, no es (sólo) para complacer a los perezosos sino para dar soporte a los ***tipos anónimos***.

```
var i = 10;          // Ok
int i2 = i + 1;      // Ok
i = "20";            // error CS0029: Cannot implicitly convert type 'string' to 'int'
string s = i;        // error CS0029: Cannot implicitly convert type 'int' to 'string'
var j;               // error CS0818: Implicitly-typed local variables must be initialized
var k = null;        // error CS0815: Cannot assign <null> to an implicitly-typed local variable
```

En el siguiente código la variable *i* es de tipo dinámico y es por ello que le podemos asignar un *int* (como en la primera línea) o bien una cadena (como en la tercera) y del mismo modo podemos asignar la variable *i* a una cadena. ***¡Ojo!*** Que podamos asignar la variable i a una cadena no significa que sea válido hacerlo: en tiempo de ejecución se realiza la transformación y puede ser que nos dé un error.

```
dynamic i = 10;     // Ok
int i2 = i + 1;     // Ok
i = "20";           // Ok
string s = i;       // Ok
dynamic j;          // Ok
dynamic k = null;   // Ok
```

Por ejemplo el siguiente código compila pero da error en ejecución.

```
dynamic i = 10;
Guid guid = i;
```

Si lo probamos vemos que sí, que compila, pero en ejecución nos lanza la excepción *RuntimeBinderException* con el mensaje "Cannot implicitly convert type *'int'* to *'System.Guid'*".

```
dynamic i = 10;
Console.WriteLine(i.GetType().FullName);      //System.Int32
i = "20";
Console.WriteLine(i.GetType().FullName);      //System.String
```

Aunque la variable *i* se haya declarado como *dynamic*, cuando se ejecuta el método *GetType()* se ejecuta sobre el objeto real contenido por *i*.

Alguien pudiera decir que si en el código anterior cambiamos *dynamic* por *object* el resultado es idéntico. ¿Entonces, dónde está la diferencia? ¿Cuándo debo usar dynamic?

Bien, simplificando podemos asumir lo siguiente: En tiempo de ejecución las variables *dynamic* se traducen a *object* (el CLR no entiende de dynamic). Pero cuando usamos *dynamic* el compilador desactiva toda comprobación de tipos, cosa que no ocurre cuando usamos *object*. Compara los dos códigos:

```
// Compila
dynamic d = "exímenos";
string str = d.ToUpper();

// NO compila
object d2 = "eiximenis";          // Ok
string str2 = d2.ToUpper();       // error CS1061: 'object' does not contain a definition for
                                     'ToUpper' and no extension method 'ToUpper' accepting a
                                     first argument of type 'object' could be found (are you
                                     missing a using directive or an assembly reference?)
```

El primer código compila mientras que el segundo no, puesto que, aunque la variable d2 contiene un objeto de tipo *string*, la referencia es de tipo *object* y *object* no contiene ningún método *ToUpper*. Mientras que en el caso de *dynamic* el compilador asume que sabemos lo que estamos haciendo, así que desactiva la

comprobación de tipos y listo. Por supuesto si en tiempo de ejecución el objeto referido por la variable dinámica no contiene el método especificado, lanza una excepción.

O sea que *dynamic* es un truco que nos proporciona el compilador: el CLR no sabe nada de *dynamic*, es el compilador de C# quien hace toda la magia. Veamos este código:

```
List<dynamic> lst = new List<dynamic>();
List<object> lst2 = new List<object>();
bool b = lst.GetType() == lst2.GetType();        // b es true
```

La variable *b* es *true* porque en tiempo de ejecución, tanto *lst* como *lst2* son un listas de objetos, dado que *dynamic* se traduce en tiempo de ejecución por *object*.

Hemos dicho que cuando usamos *dynamic*, el compilador lo que hace es básicamente declarar la variable como *object* y suspender su comprobación de tipo, ¿Pero, que más hace? Es decir, como traduce:

```
d.foo();
```

Siendo *d* una variable declarada como *dynamic*. Lo que podría hacer el compilador es simplemente "no traducirlo por nada", es decir generar el mismo código (IL) como si d fuese una variable tradicional. P.ej. dado el siguiente código C#:

```
int i = 0;
i.ToString();
```

El compilador lo traduce en el siguiente código IL (se puede ver con ildasm):

```
// int i=0;
ldc.i4.0        // Cargamos el valor 0 a la pila
stloc.0         // Sacamos el top de la pila y lo guardamos en la var #0 (i)
// i.ToString();
ldloca.s   i    // Ponemos la dirección de la variable #0 (i) en la pila
// Llamamos al método ToString. El valor the 'this' se obtiene del top de la pila call instance
string [mscorlib]System.Int32::ToString()
```

Una opción que tendría el compilador si *i* estuviese declarada como *dynamic* en lugar de *int* seria generar el mismo IL, es decir una llamada tradicional a *call*. Si en tiempo de ejecución el método indicado no se encuentra en la clase,

el *CLR* da un error. Un ejemplo de porque esto no funciona pudiera ser tener una estructura un poco más compleja como un *struct*, que como sabemos puede almacenar cualquier tipo de información como las clases, pero los *struct* son tipos por valor, o sea cada vez que se usa un *struct* se crea una copia de la estructura.

```
public struct Point
{
   public int X;
   public int Y;
}
var p= Point();
p.ToString();
```

Como vimos el código *IL* primero guarda el objeto, una referencia al *this*, luego al intentar sacar el objeto de la pila, como el *struct* puede tener cualquier tamaño a diferencia del *int*, con el *struct* el compilador no sabe de qué tamaño es el objeto y esto proporciona un error si *dynamic* pudiera guardar un *struct* de cualquier dimensión.
Otra opción que tiene el compilador es usar *Reflection*, es decir traducir la llamada *d.foo()*; a un código parecido a:

```
// código original es d.foo();
var mi = d.GetType().GetMethods().FirstOrDefault(x => x.Name.Equals("foo"));
object retval = mi.Invoke(d, null);
```

El compilador no usa ninguna de esas dos opciones. En su lugar utiliza llamadas al *DLR*. ¿Y qué es el *DLR*? Pues un conjunto de servicios (construidos encima del *CLR*) para añadir soporte a lenguajes dinámicos en *.NET*.

Te puedes preguntar porque necesitamos el *DLR* y no podemos usar simplemente *Reflection*. Aunque con *Reflection* podemos simular llamadas dinámicas, los lenguajes dinámicos permiten más cosas, como p.ej. añadir en tiempo de ejecución métodos a clases u objetos ya existentes. Hacer esto con *Reflection* es imposible, puesto que *Reflection* nos permite invocar cualquier miembro de una clase, pero dicho miembro debe estar definido en la clase cuando esta se crea (no se pueden añadir miembros en tiempo de ejecución). A continuación se encuentra un ejemplo del porque *Reflection* no es nuestra solución.

Se desea obtener un diccionario de propiedades y sus valores de un objeto declarado con la palabra clave *dynamic*. Hay varios escenarios a considerar para darle solución a este problema. Primeramente debe verificar el tipo de su objeto, simplemente puede llamar a *GetType()* para esto. Si el tipo devuelto por el método *GetType()* no implementa la interfaz *IDynamicMetaObjectProvider*, entonces puede usar *Reflection* como para cualquier otro objeto, algo como:

```
var propertyInfo = test.GetType().GetProperties();
```

Sin embargo, para las implementaciones de *IDynamicMetaObjectProvider*, reflexión simple no funciona, dado que necesitas saber más del objeto. Si es *ExpandoObject* (que implementa *IDynamicMetaObjectProvider*), implemente *IDictionary <string,object>* para sus propiedades, por lo que la solución es tan simple como:

```
dynamic s = new ExpandoObject ();
IDictionary<string, object> propValues = (IDictionary<string, object> , object >)s;
```

Si es *DynamicObject* (que implementa *IDynamicMetaObjectProvider*), entonces se necesita usar cualquier método que este exponga. *DynamicObject* no necesita "almacenar" su lista de propiedades en ningún lugar. Por ejemplo, podría hacer algo como esto:

```
public class SampleObject: DynamicObject
{
   public override bool TryGetMember(GetMemberBinder binder, out object result)
   {
        result = binder.Name;
        return true;
   }
}
```

En este caso, siempre que se intente acceder a una propiedad(con cualquier nombre), el objeto simplemente devuelve el nombre de la propiedad como un string*string*.

```
dynamic a = new SampleObject();
Console.WriteLine(a.Hello);
//Prints "Hello"
```

Por lo tanto, no se tiene que hacer *Reflection*, este objeto no tiene ninguna propiedad y, al mismo tiempo, todos los nombres de propiedad validos funcionaran.

Así pues, dado que tenemos al *DLR* que nos ofrece soporte para tipos dinámicos, el compilador de C# usa llamadas al *DLR* cuando debe resolver llamadas a miembros de objetos contenidas en variables declaradas como *dynamic*. Así pues, podemos ver que una referencia *dynamic* se traduce en tiempo de ejecución (gracias al compilador) en una referencia *object* pero que usará el *DLR* para acceder a sus miembros.
Un ejemplo sencillo y rápido es la clase *ExpandoObject*. Dicha clase le permite agregar y eliminar miembros de sus instancias de forma dinámica, en tiempo de

ejecución, así como establecer y obtener valores de estos miembros. Esta clase admite el enlace dinámico, que le permite utilizar una sintaxis estándar como *sampleObject.sampleMember* en lugar de una sintaxis más compleja como *sampleObject.GetAttribute("sampleMember")*.
La clase *ExpandoObject* implementa la interfaz estándar *Dynamic Language Runtime(DLR) IDynamicMetaProvider*, que le permite compartir instancias de la clase *ExpandoObject* entre lenguajes que admiten el modelo de interoperatividad *DLR*. Por ejemplo, puede crear una instancia en C# y luego pasarla a una función de *IronPython*.

```
static void Main(string[] args)
{
    dynamic eo = new ExpandoObject();
    eo.MiPropiedad = 10;
    eo.MiOtraPropiedad = "Cadena";
    Dump(eo);
    Console.ReadLine();
}

static void Dump(dynamic d)
{
    Console.WriteLine("MiPropiedad:" + d.MiPropiedad);
    Console.WriteLine("MiOtraPropiedad:" + d.MiOtraPropiedad);
}
```

Creamos un *ExpandoObject* y luego creamos las propiedades *MiPropiedad* y *MiOtraPropiedad*. Crear una propiedad en un *ExpandoObject* es tan simple como asignarle un valor (*¡Ojo!* la propiedad sólo se crea cuando se asigna un valor a ella, no cuando se consulta). Luego en el método *Dump* consultamos dichas propiedades y obtenemos sus valores.

Aquí el uso de *dynamic* es obligatorio: No podemos declarar la variable *eo* como *ExpandoObject* ya que entonces no podemos "añadir propiedades". Al declarar la variable como *dynamic*, hacemos que el código compile (el compilador no comprueba que existan las propiedades) y que se use el *DLR* para llamar a las propiedades *MiPropiedad* y *MiOtraPropiedad*. La clase *ExpandoObject* se integra con el *DLR* (a través de la interfaz *IDynamicMetaObjectProvider*) y eso es lo que permite que se añadan esas propiedades al objeto en cuestión.

4.2. DynamicObject

Como habíamos dicho anteriormente existe una clase llamada *DynamicObject()*, la cual es la clase base que facilita la especificación de comportamiento dinámico en tiempo de ejecución para los tipos derivados de esta. Se tiene que heredar de esta clase y sobrescribir varios de sus miembros que inician con el prefijo *Try*, en la siguiente figura se muestran algunos de estos métodos:

Method	Description
`TryBinaryOperation`	Called for binary operations such as addition and multiplication
`TryConvert`	Called for operations that convert from one type to another
`TryCreateInstance`	Called when the type is instantiated
`TryGetIndex`	Called when a value is requested via an array-style index
`TryGetMember`	Called when a value is requested via a property
`TryInvokeMember`	Called when a method is invoked
`TrySetIndex`	Called when a value is set via an array-style index
`TrySetMember`	Called when a property value is set

Figura 1. Métodos virtual de la clase `DynamicObject`.

1. El Método ***TryGetMember*** se invoca implícitamente cuando intentamos obtener un valor por medio de una propiedad.

```
bool TryGetMember(GetMemberBinder binder, out object result)
```

   Parámetros y resultado:

   - Binder: Es el generador de la información del objeto que invoca el comportamiento.
   - Result: Resultado de la operación de obtención del valor (Debe establecer un valor independiente de si se ha encontrado un valor o no, esto debido a q su especificación incluye el modificador *out*).
   - Retorno: true si la operación de establecimiento de valor fue satisfactoria, false en caso contrario. (En este último se suelen lanzar excepciones)

2. El Método ***TrySetMember*** se invoca implícitamente cuando intentamos actualizar o crear una propiedad

```
bool TrySetMember(SetMemberBinder binder, object value)
```

Parámetros y resultado:

- Binder: Es el generador de la información del objeto que invoca el comportamiento.
- Value: Instancia del objeto con la información a establecer sobre el miembro
- Retorno: *true* si la operación de establecimiento de valor fue satisfactoria, *false* en caso contrario.

3. El Método ***TryGetIndex*** se invoca implícitamente cuando intentamos obtener un valor de una propiedad mediante corchetes o índices

```
bool TryGetIndex(GetIndexBinder binder, object[] indexes, out object result)
```

Parámetros y resultado:

- Binder: Es el generador de la información del objeto que invoca el comportamiento.
- Indexes: índices que se usan en la operación.
- Result: Resultado de la operación de obtención del valor (Debe establecer un valor independiente de si se ha encontrado un valor o no, esto debido a q su especificación incluye el modificador out).
- Retorno: *true* si la operación de establecimiento de valor fue satisfactoria, *false* en caso contrario.

4. El Método ***TrySetIndex*** se invoca implícitamente cuando intentamos establecer un valor de una propiedad mediante corchetes indices

```
bool TrySetIndex(SetIndexBinder binder, object[] index, object value)
```

Parámetros y resultado:

- Binder: Es el generador de la información del objeto que invoca el comportamiento.
- Indexes: índices que se usan en la operación.
- Value: Valor que se establece en el objeto q tiene el índice especificado.
- Retorno: *true* si la operación de establecimiento de valor fue satisfactoria, *false* en caso contrario.

5. El Método ***TryInvoke*** proporciona la implementación para las operaciones que invocan un objeto

```
bool TryInvoke(InvokeBinder binder, object[] args, out object result)
```

Parámetros y resultado:

- Binder: Es el generador de la información del objeto que invoca el comportamiento.
- args: Argumentos que se pasan al objeto durante la operación de invocación.
- result: Resultado de la invocación de objeto.
- Retorno: *true* si la operación de establecimiento de valor fue satisfactoria, *false* en caso contrario.

```
bool TryInvokeMember(InvokeMemberBinder binder, object[] args, out object result)
```

Parámetros y resultado:

- Binder: Es el generador de la información del objeto que invoca el comportamiento.
- args: Argumentos que se pasan al objeto durante la operación de invocación.
- result: Resultado de la invocación de objeto.
- Retorno: *true* si la operación de establecimiento de valor fue satisfactoria, *false* en caso contrario.

Cabe destacar que la clase *ExpandoObject* es una implementación del concepto de objeto dinámico que permite obtener, configurar e invocar miembros. Si desea definir tipos que tienen su propia semántica de manejo dinámico, use la clase *DynamicObject*. Si desea definir como lo objetos dinámicos participan en el protocolo de interoperabilidad y gestionan el almacenamiento en cache dinámico rápido del *DLR*, cree su propia implementación de la interfaz *IDynamicMetaObjectProvider*

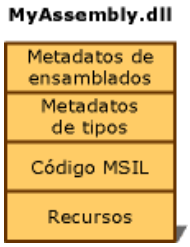
```
//Accediendo directamente a los atributos
var p1 = Factory.New.Person; //Se invoca a TrySetMember
p1.FirstName = "Yami";
p1.LastName = "Reynoso";
Console.WriteLine("Nombre:"+p1.FirstName+"| Apellido:"+p1.LastName); //Se invoca a TryGetMember
```

```
//Accediendo directamente a los atributos
var p2 = Factory.New.Person; //Se invoca a TrySetIndex
p2[“FirstName"] = "Yami";
p2[“LastName"]  = "Reynoso";
Console.WriteLine("Nombre:"+p2[“FirstName"]+“| Apellido:"+p2[“LastName"]); //Se invoca a TryGetIndex
```

5. Reflection

Reflection es el proceso por el cual se posibilita observar y modificar la estructura y comportamiento de un objeto. El *Framework .NET* posee un espacio de nombres dedicado a esta filosofía: ***System.Reflection*** que permite obtener los datos de los ensamblados cargados, y los elementos en ellos como clases, métodos y tipos.

Los ensamblados son los bloques de creación de las aplicaciones *.NET Framework*; constituyen la unidad fundamental de implementación, control de versiones, reutilización, ámbitos de activación y permisos de seguridad. Un ensamblado es una colección de tipos y recursos compilados para funcionar en conjunto y formar una unidad lógica de funcionalidad. Los ensamblados proporcionan a *Common Language Runtime* la información necesaria para conocer las implementaciones de tipos. Para la ejecución, un tipo no existe fuera del contexto de un ensamblado. O sea, es la unidad mínima que el CLR puede ejecutar. Contiene los módulos, las clases, los tipos y lo más importante, el manifiesto, que es donde se registran todos los metadatos. Cuando una aplicación arranca, el CLR consulta los metadatos del ensamblado para conocer el punto de entrada a este. Mediante reflexión, podemos obtener esos metadatos de los ensamblados cargados en la aplicación, pudiendo saber que versión de fichero estamos ejecutando, obtener una lista detallada de todas las clases y métodos que están disponibles en nuestra aplicación, o incluso cargar nuevos ensamblados en nuestra aplicación y permitir que estén disponibles de manera dinámica.



System.Type es la raíz de la funcionalidad de *System.Reflection* y constituye el modo principal de obtener acceso a los metadatos. Hay que utilizar los miembros de *Type* para obtener información sobre una declaración de tipos, como los constructores, métodos, campos, propiedades y eventos de una clase, así como el módulo y el ensamblado en que se implementa la clase. Es una clase base abstracta que permite diversas implementaciones y hereda de

System.Reflection.MemberInfo

System.Activator contiene métodos para crear tipos de objetos local o remotamente, u obtener referencias a objetos remotos existentes.

El método *System.Activator.CreateInstance()* crea una instancia de un tipo definido en un *Assembly* invocando al constructor que mejor coincide con los argumentos especificados. Si no se especifican argumentos se invoca el constructor que no toma parámetros, es decir, el constructor por defecto. Debe tener permiso suficiente para buscar y llamar a un constructor, de lo contrario se generará una excepción. Por defecto solo se consideran los constructores públicos durante la búsqueda de un constructor. De no existir constructor por defecto o no encontrar el constructor que coincida con los parámetros de búsqueda, también se lanzara una excepción.

Se puede crear una instancia de un tipo en un sitio local o remoto. Si el tipo se crea de forma remota, un parámetro de atributo de activación especifica el URL del sitio remoto. La llamada para crear la instancia podría pasar a través de sitios intermedios antes de llegar al sitio remoto. Otros atributos de activación pueden modificar el entorno o contexto en el que opera la llamada en los sitios remotos e intermedios.

Si la instancia se crea localmente, se devuelve una referencia a ese objeto. Si la instancia se crea de forma remota se devuelve una referencia a un proxy. El objeto remoto se manipula a través del proxy como si fuera un objeto local.

Cuadro 1: Metodos mas usados

Assembly	Describe un ensamblado construyendo bloques de CLR
AssemblyName	Identifica un ensamblado con un nombre único.
ConstructorInfo	Describe un constructor de la clase y da acceso a los metadatos.
MethodInfo	Describe un método de la clase y da acceso a los metadatos.
ParameterInfo	Describe un parámetro de un método y da acceso a los metadatos.
EventInfo	Describe un event info y da acceso a los metadatos
PropertyInfo	Descubre los atributos de una propiedad y da acceso a los metadatos.
MemberInfo	Obtiene información de los atributos de un miembro y da acceso a sus metadatos.

Una importante característica de *.NET Framework* es su capacidad para descubrir información de tipo en tiempo de ejecución. En concreto, puede usar el espacio de nombres *reflection* para ver la información de tipo que contienen los ensamblados que, más tarde, puede enlazar a objetos e incluso puede usar este espacio de nombres para generar código en tiempo de ejecución.

Como programador, seguramente necesites usar a menudo un objeto sin comprender del todo lo que hace ese objeto. La reflexión permite tomar un objeto y examinar sus propiedades, métodos, eventos, campos y constructores.

Como la reflexión gira entorno a *System.Type*, puede examinar un ensamblado y usar métodos, como *GetMethods()* y *GetProperties()*, para devolver información de miembro desde el ensamblado.

Usando *Reflection* podemos acceder a miembros privados que necesitan ser accedidos o modificados para comprobar el sistema. Aunque no cumple los principios de encapsulado, es generalmente aceptado para usarlo con propósitos de

comprobación. A continuación veremos un ejemplo de como acceder a distintos métodos de una clase.

```
class PrivateMethodClass
{
        private void PrivateMethod(){…}
}
```

Esta clase *PrivateMethodClass* tiene un método privado (*PrivateMethod*),el cual no se puede acceder desde fuera de la clase (ese es el proposito al hacerlos privados); pero en varias ocasiones es necesario acceder o modificar estos métodos aunque sea inaccesible, esto se logra usando *Reflection*

```
class Program
{
        static void Main(strings[] args)
        {
                typeof(MethodsClass).GetMethod("PrivateMethod",
                BindingFlags.NonPublic | BindingFlags.Instance.
                Invoke(new PrivateMethodClass(), null);
        }
}
```

GetMethod toma dos parámetros, el nombre del método y los *BindingFlags* que usa (*BindingFlags.NonPublic* incluye los metodos privados en la busqueda y *BindingFlags.Instance* incluye miembros de instancia).

```
public class Create: DynamicObject
    {
        //Se obtienen clases, metodos y propiedades del Ensablado
        Assembly myAss = Assembly.GetExecutingAssembly();

        //Usando Dynamic
        public override bool TryGetMember(GetMemberBinder binder, out object result)
        {
            //Se obtiene el acceso a los metadatos
            Type[] myclasstype = myAss.GetTypes();
            foreach (var item in myclasstype)
            {
                //Si el typo se encuentra en el Assembly entonces guarda instancia
                // en el parametro de salida y devuelve true
                if(item.Name == binder.Name)
                {
                    Type[] newtype = { };
                    //crea instancia del tipo item q esta en el Ensamblado
                    result = Activator.CreateInstance(item);
                    return true;
                }

            }
            result = null;
            return false;
        }

    }
```

6. Características que favorecen la concepción de DSL internos

6.1. Sintaxis permisiva

Mientras más dinamismo nos permita el lenguaje base más maleable será su sintaxis, lo que nos permitirá lograr el comportamiento deseado de nuestro DSL.

6.2. Homoiconicidad

Metaprogramming es una técnica de programación en la cual los programas de computadora tienen la capacidad de tratar a otros programas como sus datos. Significa que un programa puede diseñarse para leer, generar, analizar o transformar otros programas e incluso modificarse a sí mismo mientras se ejecuta. Tener el lenguaje de programación en sí mismo como un tipo de datos de primera clase como en Lisp o Prologl también es muy útil; esto se conoce como homoiconicidad.

La homoiconicidad permite extender el lenguaje con nuevos conceptos de forma más sencilla ya que los datos representando el código puede ser pasado entre las capa base y meta del programa.