

# Diseño e implementación de un DSL Interno en C# (C Sharp)

Abel Ponce González  
Gabriel Alonso Coro  
Josué Rolando Naranjo Sieiro

*Estudiantes de Ciencias de la Computación  
Asignatura: Lenguaje de Programación*

3 de junio de 2025

## 1. DSL(Domain Specific Language)

Un **DSL (Domain Specific Language)** es un lenguaje específicamente diseñado para abordar problemas dentro de un dominio particular. Según Martin Fowler, existen dos tipos: externo e interno. En este trabajo se desarrolla un DSL interno usando **C#**.

El patrón de diseño *fluent interface* permite crear métodos encadenados para obtener una sintaxis clara y legible similar al lenguaje natural.

La implementación se realiza usando cuatro variantes sintácticas:

```
1 var p1 = Factory.New.Person;
2 p1.FirstName = "Louis";
3 p1.LastName = "Dejardin";
```

### 1.1. Acceso mediante diccionario (Indexador)

```
1 var p2 = Factory.New.Person;
2 p2["FirstName"] = "Louis";
3 p2["LastName"] = "Dejardin";
```

### 1.2. Fluent Interface

```
1 var p3 = Factory.New.Person
2     .FirstNameMethod("Louis")
3     .LastNameMethod("Dejardin");
```

### 1.3. Inicialización similar a JSON

```
1 var p4 = Factory.New.Person(FirstName: "Louis", LastName: "  
Dejardin");
```

### 1.4. Clase Person

```
1 public class Person  
2 {  
3     public string FirstName { get; set; }  
4     public string LastName { get; set; }  
5  
6     private readonly Dictionary<string, string> attributes  
7         = new Dictionary<string, string>();  
8  
9     public string this[string attributeName]  
10    {  
11        get => attributes.ContainsKey(attributeName) ? attributes  
12            [attributeName] : null;  
13        set  
14        {  
15            attributes[attributeName] = value;  
16            if (attributeName == "FirstName") FirstName = value;  
17            else if (attributeName == "LastName") LastName =  
18                value;  
19        }  
20    }  
21  
22    public Person FirstNameMethod(string firstName)  
23    {  
24        FirstName = firstName;  
25        return this;  
26    }  
27  
28    public Person LastNameMethod(string lastName)  
29    {  
30        LastName = lastName;  
31        return this;  
32    }  
33}
```

### 1.5. Clase Factory

```
1 public static class Factory  
2 {  
3     public static class New  
4     {  
5         public static Person Person => new Person();  
6     }
```

```
7     public static Person Person(string FirstName, string
8         LastName)
9     {
10        return new Person
11        {
12            FirstName = FirstName,
13            LastName = LastName
14        };
15    }
16 }
```

## Características aprovechadas de C#

- Propiedades (*Properties*)
- Indexadores (*Indexers*)
- Encadenamiento de métodos (*Method chaining*)
- Sobrecarga de métodos

Este enfoque garantiza claridad, flexibilidad y facilidad de mantenimiento, proporcionando una sintaxis natural y adaptativa a diversas necesidades del dominio particular abordado.

## 2. Extensión dinámica del DSL utilizando características de C# 4.0

La llegada de C# 4.0 introdujo el tipo *dynamic*, una característica que permite definir miembros en tiempo de ejecución. Esto proporciona una flexibilidad extraordinaria, especialmente útil para la creación de DSL internos altamente adaptables y expansivos, que puedan satisfacer necesidades adicionales no anticipadas durante el diseño inicial del objeto.

### 2.1. Utilizando el tipo *dynamic* y *DynamicObject*

La clave para lograr esta flexibilidad radica en extender la clase *DynamicObject*, sobrescribiendo los métodos *TrySetMember* y *TryGetMember*, los cuales permiten definir y acceder dinámicamente a miembros del objeto.

A continuación se presenta la implementación detallada del enfoque dinámico:

```
1  using System.Dynamic;
2  using System.Collections.Generic;
3
4  public class DynamicPerson : DynamicObject
5  {
6      private readonly Dictionary<string, object> attributes = new
7          Dictionary<string, object>();
8
9      // Establece un valor dinámicamente.
10     public override bool TrySetMember(SetMemberBinder binder,
11         object value)
12     {
13         attributes[binder.Name] = value;
14         return true;
15     }
16
17     // Obtiene un valor dinámicamente.
18     public override bool TryGetMember(GetMemberBinder binder, out
19         object result)
20     {
21         return attributes.TryGetValue(binder.Name, out result);
22     }
23
24     // Opcional: acceso mediante índice de cadena
25     public object this[string attributeName]
26     {
27         get => attributes.ContainsKey(attributeName) ? attributes
28             [attributeName] : null;
29         set => attributes[attributeName] = value;
30     }
31
32     public static class DynamicFactory
33     {
```

```

31     public static dynamic NewPerson(params (string key, object
32                                     value)[] properties)
33     {
34         dynamic person = new DynamicPerson();
35         foreach (var (key, value) in properties)
36         {
37             ((DynamicPerson)person)[key] = value;
38         }
39     }
40 }
```

## 2.2. Ejemplo práctico de uso dinámico

Este enfoque permite definir objetos con propiedades no predefinidas, como en el siguiente ejemplo, donde la propiedad `Manager` se introduce dinámicamente:

```

1 var person = DynamicFactory.NewPerson(
2     ("FirstName", "Louis"),
3     ("LastName", "Dejardin"),
4     ("Manager", DynamicFactory.NewPerson(
5         ("FirstName", "Bertrand"),
6         ("LastName", "Le Roy")
7     ))
8 );
```

## 2.3. Explicación de la implementación

- **Clase `DynamicPerson`:** Esta clase extiende `DynamicObject`, permitiendo añadir atributos dinámicamente mediante un diccionario interno que almacena clave-valor.
- **Método `TrySetMember`:** Se invoca automáticamente cuando se intenta establecer una propiedad no definida previamente.
- **Método `TryGetMember`:** Se invoca automáticamente al acceder a una propiedad, permitiendo recuperarla desde el diccionario interno.
- **`DynamicFactory`:** Clase de fábrica que simplifica la creación dinámica de objetos proporcionando una interfaz intuitiva para definir propiedades arbitrarias.

## 2.4. Ventajas del enfoque dinámico

- **Flexibilidad:** Permite definir propiedades y relaciones no anticipadas.
- **Adaptabilidad:** Facilidad para modificar estructuras en tiempo de ejecución.
- **Extensibilidad:** Capacidad de expandir objetos sin alterar su definición inicial.

Este enfoque dinámico asegura que el DSL interno pueda adaptarse fácilmente a cambios en los requerimientos del dominio del problema, ofreciendo una solución poderosa y elegante a la limitación típica de los objetos rígidamente tipados.

### 3. Implementación de la clase Factory para instantiación dinámica de objetos

Una característica poderosa en C# es su capacidad para instanciar objetos de tipos conocidos en tiempo de ejecución mediante técnicas de **Reflexión**. Reflexión es el mecanismo que permite inspeccionar y manipular objetos, tipos, métodos y propiedades dinámicamente durante la ejecución del programa.

#### 3.1. Conceptos fundamentales involucrados

La implementación dinámica en C# se basa en los siguientes conceptos fundamentales:

- **Reflexión (Reflection)**: Permite obtener información sobre los tipos en tiempo de ejecución, como constructores disponibles, propiedades y métodos.
- **System.Type**: Representa metadatos sobre tipos específicos en .NET, y es clave para usar reflexión.
- **Activator**: Clase que proporciona métodos para crear instancias de objetos de forma dinámica a partir de sus metadatos.

#### 3.2. Implementación detallada usando reflexión

A continuación se presenta una implementación profesional y sencilla de la clase **Factory** que permite instanciar dinámicamente cualquier tipo conocido durante la ejecución:

```
1  using System;
2  using System.Reflection;
3
4  public static class Factory
5  {
6      // Método genérico para crear instancias por tipo
7      public static object CreateInstance(string typeName, params
8          object[] args)
9      {
10         // Obtiene el tipo basado en su nombre
11         Type type = Type.GetType(typeName);
12         if (type == null)
13             throw new ArgumentException($"El tipo '{typeName}' no
14                 fue encontrado.");
15
16         // Crea una instancia del tipo especificado con
17         // argumentos dados
18         return Activator.CreateInstance(type, args);
19     }
20
21     // Método genérico con inferencia de tipos
22     public static T CreateInstance<T>(params object[] args)
23     {
24         return (T)Activator.CreateInstance(typeof(T), args);
25     }
26 }
```

```
22     }
23 }
```

### 3.3. Ejemplo de uso práctico

La clase `Factory` puede instanciar cualquier clase previamente definida o que sea accesible en tiempo de ejecución. Ejemplo práctico:

```
1 // Usando el nombre completo del tipo como cadena
2 var persona = (Person)Factory.CreateInstance("MiNamespace.Person")
3
4 // Con argumentos al constructor
5 var fecha = (DateTime)Factory.CreateInstance("System.DateTime",
6     2025, 6, 3);
7
8 // Usando todo genérico (sin necesidad de casting explícito)
9 var personaGenerica = Factory.CreateInstance<Person>();
```

### 3.4. Características de un lenguaje que favorecen DSL embebidos

Para que un lenguaje de programación (LP) sea apropiado para crear DSL embebidos, debe cumplir ciertas características clave:

- **Reflexión y dinamismo:** Permiten instanciar y manipular objetos en tiempo de ejecución, facilitando crear estructuras dinámicas.
- **Métodos encadenados y sobrecarga de operadores:** Facilitan una sintaxis clara y fluida.
- **Inferencia de tipos y parámetros opcionales o nombrados:** Mejoran la legibilidad y simplicidad de los DSL internos.
- **Extensibilidad sintáctica:** Ofrece facilidad para expresar claramente conceptos del dominio en términos naturales dentro del lenguaje huésped.

C#, al contar con reflexión avanzada, tipos dinámicos, métodos de extensión, inferencia de tipos y características como parámetros nombrados, es un excelente candidato para la creación de DSL internos sofisticados y expresivos.

## 4. Definición y diferenciación de DLR y CLR en .NET

En el marco del desarrollo en .NET, los términos **DLR (Dynamic Language Runtime)** y **CLR (Common Language Runtime)** desempeñan roles fundamentales, aunque con funciones claramente diferenciadas. Comprender ambos conceptos es esencial para entender cómo .NET maneja lenguajes dinámicos y estáticos dentro de su plataforma.

## 4.1. Common Language Runtime (CLR)

El **Common Language Runtime (CLR)** es el componente principal del entorno de ejecución .NET, responsable de la ejecución y gestión de aplicaciones escritas en diversos lenguajes compatibles con .NET (por ejemplo, C#, VB.NET, F#). El CLR ofrece servicios fundamentales tales como:

- **Gestión de memoria (Garbage Collection):** Automatiza la asignación y liberación de memoria.
- **Manejo de excepciones:** Proporciona estructuras para tratar errores de ejecución.
- **Verificación de tipos y seguridad:** Asegura la ejecución segura y robusta de aplicaciones.
- **Compilación Just-in-Time (JIT):** Convierte código intermedio (IL) en código nativo en tiempo de ejecución.

En resumen, el CLR es el corazón de .NET, ofreciendo un ambiente estable y consistente para la ejecución de aplicaciones.

## 4.2. Dynamic Language Runtime (DLR)

El **Dynamic Language Runtime (DLR)** es una capa construida sobre el CLR, diseñada para soportar lenguajes dinámicos (por ejemplo, IronPython, IronRuby y JavaScript dentro del ecosistema .NET). El DLR facilita la ejecución eficiente y la interoperabilidad entre estos lenguajes dinámicos y lenguajes estáticos como C#.

Sus principales funciones son:

- **Resolución dinámica de operaciones:** Permite determinar métodos y propiedades en tiempo de ejecución.
- **Interoperabilidad entre lenguajes:** Facilita la integración y la comunicación entre diferentes lenguajes dinámicos y estáticos.
- **Generación dinámica de código:** Optimiza operaciones dinámicas mediante caché y técnicas avanzadas de compilación.

El DLR permite así una gran flexibilidad en la programación y una integración fluida de múltiples paradigmas y lenguajes dentro del entorno .NET.

## 4.3. Relación y diferencias entre CLR y DLR

El CLR y el DLR están estrechamente relacionados, pero no se encuentran al mismo nivel jerárquico en la arquitectura .NET:

- **CLR:** Representa la base del entorno .NET. Todos los lenguajes compatibles con .NET, incluyendo los dinámicos, dependen del CLR para la ejecución de código.
- **DLR:** Es una capa adicional construida sobre el CLR. Se especializa en proporcionar mecanismos para ejecutar código dinámico y mejorar la interoperabilidad de lenguajes dinámicos con el entorno estático.

Gráficamente, la arquitectura podría describirse como:

**Lenguajes Dinámicos → DLR → CLR**

En otras palabras, el DLR no es independiente, sino una extensión que potencia al CLR con capacidades específicas para lenguajes dinámicos.

#### **4.4. Importancia práctica del DLR**

La introducción del DLR permite al entorno .NET una mayor flexibilidad y expresividad, facilitando a los desarrolladores trabajar con lenguajes dinámicos y métodos dinámicos dentro de sus aplicaciones. Esta combinación de CLR y DLR hace posible crear aplicaciones robustas, adaptables y eficientes que aprovechen las fortalezas tanto del dinamismo como de la tipificación estática.

### **5. Nivel arquitectónico y relación entre DLR y CLR en .NET**

En la arquitectura de la plataforma .NET, es importante clarificar cómo interactúan y en qué niveles operan el **Common Language Runtime (CLR)** y el **Dynamic Language Runtime (DLR)**. Aunque ambos runtimes están estrechamente relacionados y colaboran en la ejecución del código, operan en diferentes niveles dentro de la pila tecnológica de .NET.

#### **5.1. Ubicación del CLR dentro de la arquitectura .NET**

El **CLR** constituye el núcleo fundamental de la plataforma .NET. Todos los lenguajes compatibles con .NET (estáticos o dinámicos) son compilados hacia un lenguaje intermedio conocido como **Intermediate Language (IL)**, y posteriormente este código intermedio es ejecutado por el CLR, el cual:

- Ejecuta y optimiza código mediante compilación Just-in-Time (JIT).
- Gestiona memoria mediante recolección de basura (garbage collector).
- Proporciona seguridad en la ejecución y validación de tipos.
- Proporciona interoperabilidad de múltiples lenguajes compilados a IL.

El CLR es, por tanto, el componente central sobre el cual se apoya toda la infraestructura de ejecución en .NET.

#### **5.2. Ubicación del DLR dentro de la arquitectura .NET**

El **DLR** no es un runtime independiente al nivel del CLR, sino que es una capa complementaria que reside directamente sobre el CLR. El DLR añade soporte para lenguajes dinámicos dentro del ecosistema .NET, facilitando la implementación de características específicas como:

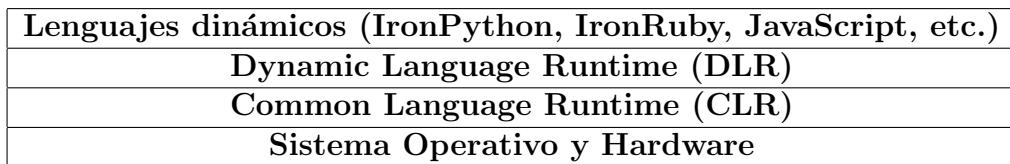
- Ejecución dinámica y resolución de tipos en tiempo de ejecución.
- Interoperabilidad mejorada entre lenguajes dinámicos y estáticos.
- Gestión eficiente de código dinámico mediante técnicas avanzadas como *call sites* y *binders*.

En otras palabras, el DLR utiliza directamente las capacidades del CLR para ejecutar código, añadiendo funcionalidades específicas para soportar lenguajes dinámicos y mecanismos de resolución dinámica.

### 5.3. ¿Están al mismo nivel DLR y CLR?

La respuesta corta es: **no, no están al mismo nivel**. La relación entre DLR y CLR es jerárquica, con el CLR situado en la base de la plataforma, y el DLR actuando como una capa adicional sobre este último, ofreciendo soporte especializado para lenguajes dinámicos.

Una representación esquemática simplificada sería la siguiente:



Esta estructura evidencia claramente que el DLR actúa como una capa de extensión sobre el CLR, y no como un reemplazo o equivalente.

### 5.4. Importancia de comprender la jerarquía CLR-DLR

Entender esta distinción es crucial para:

- Comprender el rendimiento y los costos asociados al dinamismo en tiempo de ejecución.
- Optimizar la interoperabilidad entre lenguajes dinámicos y estáticos en aplicaciones híbridas.
- Aprovechar adecuadamente las ventajas que ofrece cada runtime según el contexto específico de la aplicación desarrollada.

En resumen, aunque estrechamente interrelacionados, CLR y DLR desempeñan roles diferenciados en diferentes niveles arquitectónicos dentro de .NET, siendo el CLR el soporte fundamental y el DLR la capa superior especializada en soporte dinámico.

Aquí tienes la respuesta detallada y profesional a la \*\*pregunta 6\*\*, preparada como una sección LaTeX para agregar a tu informe actual:

““latex

## 6. Conceptos clave: Call site, Receiver y Binder en DLR

En la ejecución dinámica dentro del **Dynamic Language Runtime (DLR)**, tres conceptos clave resultan esenciales para entender cómo el DLR realiza la resolución y optimización de operaciones dinámicas: **call site**, **receiver** y **binder**. A continuación, se define cada término y se explica su rol dentro del proceso dinámico.

### 6.1. Call site

Un **call site** (sitio de llamada) es un punto específico en el código fuente donde ocurre una operación dinámica. Es el lugar en el código en donde el DLR necesita resolver dinámicamente el método o la operación específica en tiempo de ejecución.

Características destacadas del call site:

- Específico a cada operación dinámica.
- Actúa como punto de interacción con el DLR, almacenando información que permite optimizar futuras invocaciones.
- Usa mecanismos internos para cachear resoluciones previas, aumentando el rendimiento en operaciones repetitivas.

### 6.2. Receiver

El **receiver** (receptor) es el objeto sobre el cual se ejecuta la operación dinámica especificada en el call site. En otras palabras, es el objeto cuyo método, propiedad o acción se intenta invocar o acceder dinámicamente.

El receiver tiene las siguientes características clave:

- Representa el contexto concreto sobre el cual se aplican operaciones dinámicas.
- Puede variar durante la ejecución, permitiendo resoluciones diferentes en tiempo de ejecución según su tipo o estructura.
- Su naturaleza dinámica permite comportamientos flexibles, típicos de lenguajes como Python, Ruby, o JavaScript, dentro de .NET.

### 6.3. Binder

El **binder** (enlazador) es el componente del DLR encargado de determinar cómo se resuelve una operación dinámica particular en tiempo de ejecución. El binder utiliza la información del call site y del receiver para determinar qué método específico, propiedad o acción ejecutar.

Sus características principales son:

- Es responsable de resolver dinámicamente referencias, accesos o invocaciones.
- Realiza análisis en tiempo de ejecución basados en el contexto actual.
- Aplica reglas de resolución que permiten reutilizar resultados previos (caché) optimizando el rendimiento.

## 6.4. Interacción entre call site, receiver y binder

La interacción entre estos tres elementos ocurre como sigue:

1. **Call site** indica al DLR que una operación dinámica necesita resolución.
2. El **binder** analiza el call site y el **receiver** (objeto objetivo) para resolver la operación concreta a ejecutar.
3. Una vez resuelta, la operación dinámica se ejecuta sobre el receiver y el resultado puede cachearse en el call site para futuras invocaciones más rápidas.

## 6.5. Ejemplo ilustrativo

Considera el siguiente ejemplo simplificado:

```
1 // Operación dinámica en C#
2 dynamic persona = GetPersona();
3 persona.Saludar(); // Call site dinámico
```

En este ejemplo:

- **Call site:** Es `persona.Saludar()`.
- **Receiver:** Es el objeto `persona`.
- **Binder:** Determina en tiempo de ejecución cuál método exacto de `Saludar()` se ejecutará, dependiendo del tipo real del objeto en ese momento.

## 6.6. Importancia de estos conceptos

La comprensión profunda de estos conceptos permite:

- Mejorar la eficiencia en programación dinámica al aprovechar mecanismos de optimización del DLR.
- Diagnosticar problemas relacionados con desempeño en aplicaciones dinámicas.
- Escribir código claro y efectivo aprovechando al máximo las características dinámicas de .NET.

En conclusión, los conceptos **call site**, **receiver** y **binder** son fundamentales para comprender y utilizar adecuadamente las capacidades dinámicas proporcionadas por el DLR en la plataforma .NET. “