1 Seminario DSL-Reflection-Dynamic

1.1 Lenguajes de Dominio Específico (DSL)

Supongamos la siguiente situación. Se quiere implementar un mecanismo para crear objetos de tipo *persona*. A estas *personas* debe ser posible modificarles su *nombre* y *apellido*. Una posible implementación para esto es la siguiente:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public StateMachine(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
```

Luego podría crearse una persona de la siguiente manera:

```
var person = new Person("Jorge", "Mederos");
```

Lo anterior resuelve nuestro problema, pero que tal si quisieramos una manera mas expresiva.

> Si el ejemplo siguiente es o no una manera mas expresiva, es algo subjetivo y que por supuesto para algo tan sencillo puede no marcar la diferencia

```
static class PersonExtensions
{
    public static Person FirstName(this Person person, string firstName)
    {
        person.FirstName = firstName;
        return person;
    }

    public static Person LastName(this Person person, string lastName)
    {
        person.LastName = lastName;
        return person;
```

```
    }
}
```

Esto nos permitiria hacer lo siguiente:

```
var person = new Person()
             .FirstName("Jorge")
             .LastName("Mederos")
```

Lo que hemos hecho anteriormente es utilizar un ***DSL*** para representar nuestro problema. A este tipo de interfaces o *apis* que permiten llamados a métodos de forma concatenada que mantienen estrecha relación entre sí, se les llama *Fluent API*, dado que permiten una manera fluida de escribir el código, dando la sensación de ser un lenguaje diferente.

> Un ***DSL*** es un lenguaje de programacion de expresividad limitada, enfocado a un dominio especifico.

Analicemos la definicion anterior:

- **Lenguaje de programacion:**
  Un ***DSL*** es utilizado para dar instrucciones a una computadora. Como cualquier otro lenguaje de programación, debe proveer una manera *legible* para los seres humanos de expresar lo que se desea, y aun asi premitir su ejecución en una computadora.
- **Naturaleza de lenguaje:**
  Como lenguaje de programación, debe proveer fluidez, la expresividad no debe solo venir de las expresiones individuales, sino de la posibilidad de componer las mismas.
- **Expresividad limitada:**
  Un lenguaje de programación de proposito general posee estructuras para controlar el flujo del programa y varias instrucciones complejas, estas pueden ser muy útiles, pero esto convierte al lenguaje en algo complejo de aprender, utilizar y entender. Un ***DSL*** por otro lado es facil de comprender, es legible, soporta solo lo necesario para ser utilizado en su dominio. No es posible contruir toda una aplicación con un ***DSL***, se construyen componentes de la misma. No es turing-completo en la mayoría de los casos.
- **Enfocado a un dominio específico:**
  Intuitivamente un lenguaje tan limitado solo puede ser útil si se conoce bien el dominio en el que será aplicado. Este enfoque en un dominio específico es lo que hace que un ***DSL*** tenga sentido ser utilizado.

¿Que tal si tuvieramos lo siguiente en un archivo externo a la aplicación?

```
person:
    first-name: "Jorge"
```

```
last-name: "Mederos"
```

Esta claro que es posible leer el contenido de este archivo y luego de un proceso de *parsing* convertirlo en una instancia de `Person`. En este caso tambien estaríamos en presencia de un ***DSL***. Solo que este estaria ubicado en un archivo externo a la aplicación principal.

Es posible dividir los ***DSL*** en tres grupos, de acuerdo a su naturaleza.

1. **DSLs externos:**
   Un **DSL Externo** es un lenguaje diferente al de la aplicacion principal, generalmente con una sintaxis propia, aunque en ocaciones pueden utilizarse algunas existentes como *xml*, *json*, etc. Un script en un *dsl externo* usualmente es parseado por la aplicacion principal.
2. **DSLs internos**:
   Un **DSL Interno** es una manera particular de utilizar un lenguaje de proposito general. Un script en un *dsl interno* es valido tambien en el lenguaje de proposito general, pero solamente utiliza un subconjunto del mismo en un estilo determinado para un aspecto del sistema general. El resultado se siente como un lenguaje diferente. **LISP** y **RUBY** son ejemplos importantes aqui.
3. **Language Workbenches:**
   Un **Language Workbench** es un *IDE* especializado para la creación de ***DSLs***. En particular un **Language Workbench** no solo es utilizado para determinar la estructura del ***DSL*** sino para editar scripts de este ***DSL***. EL *script* resultante íntimamente combina el *IDE* con el ***DSL***

El primer caso que analizamos es un ***DSL*** *interno* y el segundo uno *externo*.

1.1.1 ¿Como distinguir un DSL?

Es facil ver que identificar si estamos en presencia de un **DSL** o no, es algo que puede llegar a ser ambiguo, la frontera entre lo que es un **DSL** y lo que no lo es es algo borrosa en muchos casos. Tratemos de definir mejor esta frontera para cada tipo de **DSL**.

Para los **DSLs internos** esta frontera la define la expresividad limitada. La diferencia aqui se establece entre el **DSL interno** y una *command query api*. Una manera comun de documentar una *clase* en una *api clásica* es listar los metodos que esta contiene y documentarlos uno a uno, estos metodos tienen que tener sentido por si solos, ser *autocontenidos*; por otro lado en un **DSL interno** podria pensarse en los metodos como palabras de un lenguaje que no expresan una idea hasta que no son compuestas en oraciones. Por ejemplo metodos `transition(event)` o `to(dest)` - correspondientes a la creación de una maquina de estado - no tendrian sentido por si solo en una *api clasica*, pero pueden ser

utilizados en un **DSL interno** de la siguiente manera `.transition(event).to(dest)`. Es común tambien destinguir los **DSLs internos** por la no utilizacion de condicionales, ciclos, o cualquier otra estructura de control de flujo.

Para los **DSLs externos** la frontera se establece con los lenguajes de proposito general. Ser enfocados a un dominio no convierte a un lenguaje en un **DSL**, dado que a pesar de su enfoque especifico, puede ser utilizado para tareas de proposito general, por ejemplo lenguaes como **R**. Otro ejemplo de un **DSL externo** son las expresiones regulares, estas definen un lenguaje enfocado a hacer *text matching*, y ademas de tener este enfoque especifico, tienen un lenguaje lo suficientemente *limitado* como para lograr solo la expresividad necesaria para resolver problemas de este dominio. Es comun distinguir en los **DSL** que no sean *Turing completos*. Los archivos de configuracion (*.json*, *.xml*, *etc*) pueden considerarse **DSL**, pero solo en los casos en los que tienen el objetivo de ser no solo entendibles por un ser humano, sino de ser editables.

1.1.2 Modelo Semántico

Observese que en el caso del *DSL* externo es posible construir luego del proceso de ***parsing*** una instancia de `StateMachine` con el primer modelo, es decir:

```
class StateMachine
{
    private List<Transition> transitions;

    public StateMachine()
    {
        this.transitions = new List<Transition>();
    }

    public AddTransition(Transition transition)
    {
        this.transitions.Add(transition);
    }
}

class Transition
{
    public string Origin { get; private set; }
    public string Dest { get; private set; }

    public Transition(string origin, string dest)
```

```
    {
        this.Origin = origin;
        this.Dest = dest;
    }
}
```

En este caso al modelo anterior le llamaremos ***modelo semántico***, el modelo es agnóstico al ***DSL***, lo que nos brinda varias ventajas:

- Permite pensar en la semántica del modelo sin estar pensando simultáneamente en la sintaxis del ***DSL***, esto es significativo dado que normalmente implementamos un ***DSL*** para representar situaciones bastante complejas.
- Permite testear de manera separada el *modelo* y el ***DSL***.
- Es posible tener múltiples ***DSL*** que utilicen el mismo modelo (incluyendo externos e internos).
- Es posible añadir funcionalidad al ***DSL*** y al *modelo* de manera independiente

La existencia de un modelo semántico nos permite tambien tener varios ***DSL*** para un mismo modelo semántico, permitiéndonos asi pensar en los ***DSL*** como una mera interfaz al *modelo*. De cierta manera esta separación podría asemejarse la separación entre *lógica* y *visualización* en cualquier otra aplicación.

¿Es lo mismo Modelo Semantico que AST?

Si estamos usando un *DSL externo* y parseamos texto, obtendremos como sabemos un *AST* - En un *DSL interno* la jerarquia del *AST* la darian los llamados a funciones -, la idea es convertir este *AST* a nuestro ***Semantic Model***, dado que ejecutar sobre el ***AST***, a pesar de ser posible, complijazaría mucho la situación, debido a la atadura tan grande que se crearía entre la sintaxis del ***DSL*** y la capacidad de procesamiento del mismo

El *modelo semántico* no se restringe solo a los *DSLs externos*:

Es posible obtener las ventajas que nos brinda la existencia de un *modelo semántico* en un ***DSL*** *interno*. Para ello necesitamos separar la capa del lenguaje de la capa del modelo, es aquí donde entran a jugar un papel fundamental los *Expression Builders*, basicamente tendremos una capa en nuestro modelo que proveerá el lenguaje y construir de la manera debida la instancia del modelo que provee la funcionalidad.

Un ejemplo de *ExpressionBuilder* para `Person` quedaria de la siguiente manera:

```
class PersonBuilder
{
  private Person currentPerson = null;

  public PersonBuilder Build()
```

```
  {
    this.currentPerson = new Person();
    return this;
  }

  public PersonBuilder FirstName(string name)
  {
    this.FirstName = name;
    return this;
  }

  public PersonBuilder LastName(string name)
  {
    this.LastName = name;
    return this;
  }

  public Person End()
  {
    return this.currentPerson;
  }
}
```

1.1.3 Fluent Interface

Las interfaces fluidas o encadenamiento de metodos son muy utilizadas en la creación de *DSLs*, dobre todo en lenguajes con sintaxis poco flexible, se utilizan para dar una sensacion de estar escribiendo en otro lenguaje, con mas fluidez, evitando el uso de variables y ganando en expresividad.

Hay varias maneras de poner en practica este concepto. Los metodos que seran encadenados al llamarlos pueden ser implementados a partir de metodos extensores en un lenguaje como *C#*, esto permite la separación una vez mas del lenguaje y el modelo.

Los *ExpressionBuilders* vistos anteriormente son un lugar muy común donde implementar esta interfaz, dado que como se explica anteriormente estos son los encargados de la capa del lenguaje. La implementacion de este tipo de interfaces fluidas en los *ExpressionBuilders* trae algunos problemas, como por ejemplo el problema de la terminación. Este problema consiste en que dado que cada metodo retorna la propia instancia del *ExpressionBuilder* entonces puede ser complicado decidir cuando se ha terminado la cadena de metodos y en este caso retornar la instancia que se quiere construir. Este problema se puede

resolver añadiendo un metodo `End()` al builder, que al ser llamado retornara la instancia construida hasta el momento; este enfoque puede parecer un poco *verbose*. Otra forma puede ser en lenguajes como *C#* establecer un casteo implicito entre un builder y el tipo de objetos que este construye.

Otro enfoque para la implementación de interfaces fluidas son las *progressive interfaces*. Esta idea consiste en que cada metodo en la fluent interface retorna que implementa un interface de solo un metodo, de manera tal que el orden en que pueden encadenarse los metodos es determinado por el creador del *DSL*.

1.1.4 Ventajas y Desventajas de los DSL:

Un *DSL* tiene, entre otras, las siguientes ventajas:

- **Aumento de la productividad:**
  Menos bugs, más fácil de encontrarlos. Menos duplicación de código, gracias a modelo semántico.
- **Comunicación con expertos del dominio**
- **Mayor expresividad, el codigo tiene mayor semántica**
- **Modelo Computacional Alternativo:**
  En ocaciones una máquina de estado o expresiones regulares, etc, pueden ser una mejor manera de lograr lo que se desea que la clásica programación imperativa, donde tenemos que tener en cuenta cada detalle para obtener el funcionamiento correcto de nuestro programa. En este caso el *modelo semántico* representa esta forma computacional alternativa, mientras que un **DSL** permite instanciar y *construir* este modelo de manera cómoda, con todas las ventajas que hemos visto hasta ahora.

Por otro lado, tambien tiene inconvenientes:

- **Cacofonía del lenguaje:**
  El problema más común con los ***DSL*** es la gran cantidad de ***DSL*** que pueden llegar a existir, lo cual hace difícil que la gente los aprenda; y como al traer gente nueva a un proyecto, puede tomar tiempo que los nuevos trabajadores se adapten a los ***DSL*** que en este son usados.

1.1.5 Algunos ejemplos y exponentes:

1.1.6 LISP

Lisp (históricamente LISP) es una familia de lenguajes de programación con una larga historia y una notación de prefijo distintiva, completamente entre paréntesis. Originalmente

especificado en 1958, Lisp es el segundo lenguaje de programación de alto nivel más antiguo en uso generalizado en la actualidad. Solo Fortran es mayor, por un año.

Lisp fue creado originalmente como una práctica notación matemática para los programas de ordenador. Como uno de los primeros lenguajes de programación, Lisp fue pionera en muchas ideas en ciencias de la computación, incluyendo estructuras de árbol de datos, gestión de almacenamiento automático, tipado dinámico, condicionales, funciones de orden superior, la recursividad, la autoalojamiento compilador, y el ciclo de lectura – evaluación – impresión .

Las listas enlazadas son una de las principales estructuras de datos de Lisp, y el código fuente de Lisp está hecho de listas. Por lo tanto, los programas Lisp pueden manipular el código fuente como una estructura de datos, dando lugar a los macro sistemas que permiten a los programadores crear nuevas sintaxis o nuevos lenguajes específicos de dominio integrados en Lisp.

1.1.6.1 Sintaxis de Lisp

Lisp es un lenguaje orientado a expresiones y por tanto no se hace distinción entre "expresiones" y "declaraciones"; todo el código y los datos se escriben como expresiones. Cuando se evalúa una expresión, produce un valor, que luego se puede incrustar en otras expresiones. Cada valor puede ser cualquier tipo de datos.

El uso de paréntesis es la diferencia más obvia de Lisp de otras familias de lenguaje de programación y un rasgo que en ocaciones resulta incómodo. Pero, la sintaxis de las expresiones *S* también es responsable de gran parte del poder de Lisp: la sintaxis es extremadamente regular, lo que facilita la manipulación por computadora.

Las funciones de Lisp se escriben como listas, se pueden procesar exactamente como los datos. Esto permite escribir fácilmente programas que manipulan otros programas (metaprogramación). Muchos dialectos de Lisp explotan esta característica utilizando sistemas macro, que permiten la extensión del lenguaje casi sin límite.

1.1.6.2 Expresiones *S*:

Una *expresión s*, también conocida como *sexpr* o *sexp* , es una forma de representar una lista anidada de datos . Significa "*expresión simbólica*".Por ejemplo, la expresión matemática simple `5*(7+3)` se puede escribir como una *expresión s* con notación de prefijo de la forma `(* 5 (+ 7 3))`.

1.1.6.3 Listas

Una lista de Lisp se escribe con sus elementos separados por espacios en blanco y entre paréntesis. La lista vacía `()` también se representa como el átomo especial **nil**. Esta es la única entidad en Lisp que es tanto un átomo como una lista.

1.1.6.4 Expresiones

Las expresiones se escriben como listas, usando la notación de prefijo . El primer elemento en la lista es el nombre de una función, el nombre de una macro, una expresión lambda o el nombre de un "*operador especial*". El resto de la lista son los argumentos. Por ejemplo, la función `list` devuelve sus argumentos como una lista:

```
(list 1 2 (quote foo))
```

Cualquier expresión sin comillas se evalúa de forma recursiva antes de evaluar la expresión de cierre.

```
(list 1 2 (list 3 4))
```

AL evaluar esta expresión tenga en cuenta que el tercer argumento es una lista, o sea (1 2 (3 4))

1.1.6.5 Operadores

Los operadores aritméticos se tratan de manera similar a las expresiones:

```
(+ 1 2 3 4)
```

Lisp no tiene noción de operadores, los operadores aritméticos en Lisp son funciones variadas (o n-arias), capaces de tomar cualquier número de argumentos. Un operador de incremento '++' de estilo *C* a veces se implementa bajo el nombre **incf** dando sintaxis

```
(incf x)
```

lo cual equivale a:

```
(setq x (+ x 1))
```

Los "*operadores especiales*" (a veces llamados "*formas especiales*") proporcionan la estructura de control de Lisp. Por ejemplo, el operador especial **if** toma tres argumentos. Si el primer argumento no es **nil** , se evalúa como el segundo argumento; de lo contrario, se evalúa al tercer argumento. Así, la expresión

```
(if nil
  (list 1 2 "foo")
  (list 3 4 "bar"))
```

Lisp también proporciona operadores lógicos. Los operadores **and** y **or** realizan una evaluación de cortocircuito y devolverán su primer argumento nil y non-nil respectivamente.

```
(or (and "zero" nil "never") "James" 'task 'time)
```

1.1.6.6 Expresiones lambda y definición de funciones

Otro operador especial **lambda**, se utiliza para vincular variables a valores que luego se evalúan dentro de una expresión. Este operador también se utiliza para crear funciones: los argumentos lambda son una lista de argumentos y la expresión o expresiones a las que se evalúa la función.

```
(lambda (arg) (+ arg 1))
```

Las expresiones lambda no se tratan de manera diferente a las funciones nombradas; se invocan de la misma manera. Por lo tanto, la expresión

```
((lambda (arg) (+ arg 1)) 5)
```

evalúa a 6. A este caso se le conoce como **función anónima**.Las funciones con nombre se crean almacenando una expresión lambda en un símbolo utilizando la macro **defun** .

```
(defun foo (a b c d) (+ a b c d))
```

De esta manera se define una nueva función nombrada foo el entorno global. Otra forma de lograrlo es

```
(setf (fdefinition 'f) #'(lambda (a) (block f b...)))
```

1.1.6.7 Estructura de lista del código del programa, explotación por macros y compiladores

Una distinción fundamental entre Lisp y otros lenguajes es que en Lisp, la representación textual de un programa es simplemente una descripción legible por humanos de las mismas estructuras de datos internas (listas enlazadas, símbolos, números, caracteres, etc.) que usarían el sistema subyacente Lisp.

Lisp usa esto para implementar un sistema macro muy poderoso. Al igual que otros lenguajes de macros como *C* , una macro devuelve código que luego se puede compilar. Sin embargo, a diferencia de las macros C, las macros son funciones de Lisp y, por lo tanto, pueden explotar todo el poder de Lisp.

Además, debido a que el código Lisp tiene la misma estructura que las listas, las macros se pueden construir con cualquiera de las funciones de procesamiento de listas en el lenguaje. En resumen, cualquier cosa que Lisp pueda hacer a una estructura de datos, las macros

de Lisp pueden hacer al código. En contraste, en la mayoría de los otros idiomas, la salida del analizador es puramente interna a la implementación del lenguaje y el programador no puede manipularla.

En implementaciones simplistas de Lisp, esta estructura de lista se interpreta directamente para ejecutar el programa. Una función es literalmente una parte de la estructura de la lista que el intérprete recorre al ejecutarla. Sin embargo, los sistemas Lisp más importantes también incluyen un compilador. El compilador traduce la estructura de la lista a *código de máquina* o *bytecode* para su ejecución. Este código puede ejecutarse tan rápido como el código compilado en lenguajes convencionales como *C*.

1.1.6.8 Macros de Lisp y DSL

Las macros se utilizan para definir extensiones de sintaxis de lenguaje para *Common Lisp* o *Lenguajes de Dominio Específico (DSL)*. Estos idiomas están integrados directamente en el código Lisp existente. Ahora, los DSL pueden tener una sintaxis similar a Lisp o completamente diferente.

Aquí hay un ejemplo más concreto: *Python* tiene *list comprehensions* integradas en el lenguaje. Esto proporciona una sintaxis simple para un caso común. La línea

```
even = [x for x in range(10) if x % 2 == 0]
```

produce una lista que contiene todos los números pares entre 0 y 9. En Lisp podríamos definir un *DSL* para hacer *list comprehensions*.

Lisp define un par de formas de sintaxis especiales.

- `'`: indica que el siguiente token es literal.
- `` ` ``: indica que el siguiente token es un literal con excepciones. Las excepciones estan precedidas por el operador de coma. El literal `'(1 2 3)` es el equivalente de *Python* `[1, 2, 3]`. Puede asignarlo a otra variable o usarlo en su lugar. Puede pensar `(1 2 ,x)` como el equivalente de *Python* `[1, 2, x]`donde `x` es una variable previamente definida. Esta notación de la lista es parte de la magia que entra en las macros. La segunda parte es el lector Lisp que sustituye inteligentemente las macros por el código, pero que se ilustra mejor a continuación:

Entonces podemos definir una macro llamada `lcomp`(abreviatura para *list comprehension*). Su sintaxis será exactamente como la de python que utilizamos en el ejemplo `[x for x in range(10) if x % 2 == 0]`, `(lcomp x for x in (range 10) if (= (% x 2) 0))`

```
(defmacro lcomp (expression for var in list conditional conditional-test)
  (let ((result (gensym))) ;; crear un nombre de variable unico para el resultad
    ;; the arguments are really code so we can substitute them
```

```
;; store nil in the unique variable name generated above
` (let ((,result nil))
   ;; var es un nombre de variable
   ;; list es la lista literal por la que iteraremos
   (loop for ,var in ,list
        ;; conditional es "if" o "unless"
        ;; conditional-test es (= (mod x 2) 0) en nuestro ejemplo
        ,conditional ,conditional-test
        ;; y esta es la acción del ejemplo de lisp anterior
        ;; result = result + [x] en python
        do (setq ,result (append ,result (list ,expression))))
       ;; retorna result
   ,result)))
```

Ahora podemos ejecutar en la línea de comando:

```
(let ((,result nil)(lcomp x for x in (range 10) if (= (mod x 2) 0))
```

De esta manera es posible lograr casi cualquier sintaxis que se pueda desear.

1.1.7 Ruby

Ruby es un lenguaje orientado a objetos y sigue la noción habitual de cualquier otro lenguaje orientado a objetos para definir una clase, *Ruby* tiene su propio modelo de objetos el cual tiene artefactos como clases, objetos, métodos de instancia, métodos de clase y métodos singleton; posee soporte para expresiones regulares, lo cual es muy útil en la coincidencia de patrones y procesamiento de texto y se pueden implementar bloques, los cuales se utilizan para implementar lambdas y cierres en *Ruby*.

Sin embargo, quizás la característica más apreciada de *Ruby* es la metaprogramación, con la cual se puede manipular el lenguaje para satisfacer las necesidades, en lugar de adaptarse al lenguaje cómo es, es por ello que la metaprogramación y los DSLs tienen una estrecha relación en el mundo *Ruby*. *Ruby* es un lenguaje altamente dinámico; puede insertar nuevos métodos en las clases en tiempo de ejecución (incluso una clase principal como Array), crear alias para los métodos existentes, e incluso definir métodos en objetos individuales (métodos Singleton). Además, tiene una rica API para la *reflection*. Un programa de *Ruby* puede configurar dinámicamente nombres de variables, invocar nombres de métodos e incluso definir nuevas clases y nuevos métodos.

1.1.7.1 Caracteristicas distintivas del lenguaje Ruby

La sintaxis de *Ruby* esta orientada a la expresividad y su unidad básica es la expresión, el intérprete de *Ruby* evalúa las expresiones y produce valores. Las expresiones más simples son las expresiones primarias, las cuales representan directamente los valores, ejemplo de esto son los números y las cadenas. El lenguaje tiene muchas similitudes con *Python*, aunque con diferencia en pequeños detalles. Mientras que *Python* intenta a la vez ser lo mas legible y sencillo posible, *Ruby* intenta ser lo mas explesivo posible dentro de la sencillez y el tipado dinámico. Luego siguiendo esa idea los *scope* en *Ruby* estan delimitados por las palabras predefinidas del lenguage (`if`, `def`, etc.) y la palabra clave `end`. A continuacion se enumeraran algunas diferencias.

1. En *Ruby* Las estructuras de control son expresiones y por tanto, se puede escribir código como el siguiente:

   ```
   min = if x < t then x else t end
   ```

2. En *Ruby* tienes una referencia a la clase (`self`) que ya está en el cuerpo de la clase. En Python, no tiene una referencia a la clase hasta que finaliza la construcción de la clase.

```
class RubyClass
  puts self
end
```

`self` en este caso es la clase, y este código imprimirá “RubyClass”. No hay forma de imprimir el nombre de la clase o de otra manera acceder a la clase desde el cuerpo de definición de clase en Python (definiciones de método externas).

3. En Ruby para referirse a una propiedad de una clase se realiza a partir del `@` en lugar de `self.property`.

```
class MyRubyClass
  def initialize(text)
    @text = text
  end
  def welcome
    @text
  end
end
```

4. Todas las clases son mutables en Ruby. Esto le permite desarrollar extensiones para las clases principales:

   ```
   class String
     def starts_with?(other)
       head = self[0, other.length]
   ```

```
        head == other
      end
    end
```

5. Ruby tiene bloques Con la instrucción `do`, puede crear una función anónima de varias líneas en Ruby, que se pasará como un argumento al método frente a `do`, y se llamará desde allí. En Python, en su lugar, haría esto ya sea pasando un método o con generadores.

```
def themethod
  yield 5
end

themethod do |foo|
  puts foo
end
```

6. En Ruby, cuando se importa un archivo con **require**, todas las cosas definidas en ese archivo terminarán en su espacio de nombres global. Esto causa la contaminación del espacio de nombres. La solución a eso son los módulos Rubys. Pero si crea un espacio de nombres con un módulo, debe usar ese espacio de nombres para acceder a las clases contenidas.

7. Ruby no tiene herencia multipli, sino que reutiliza el concepto de módulo como un tipo de clases abstractas.

8. Ruby simula las *list comprehensions* de *Python* de la siguiente manera:

```
res = (0..9).map { |x| x * x }
```

1.1.7.2 Ruby y los DSL

En los lenguajes compilados como *C++* los métodos y variables tienen valor en un espacio de memoria solo en tiempo de compilación, una vez finaliza el período de compilación los espacios de memoria se liberan y se pierde la información relacionada con el programa, por esa razón no se pueden solicitar a una clase sus métodos de instancia, ya que, en el momento de hacer la solicitud, la clase se ha desvanecido. Por otra parte, en los lenguajes interpretados como Ruby la metaprogramación es posible, ya que en tiempo de ejecución la mayoría de las construcciones del lenguaje todavía están ahí, de esa forma, se pueden consultar valores y construcciones del programa en ejecución.

Para la metaprogramacion *Ruby* se vale de las clases `Kernel`, `Object` y `Module` que definen métodos con el mismo objetivo que `System.Reflection` en *C#*, trabajar con los metadatos del programa en ejecución. Pero gracias al tipado dinámico de Ruby y que sus

tipos principales no son inmutables, este trabajo con los metadatos del programa aporta una gran flexibilidad al lenguaje. A continuación algunos ejemplos:

```
    def array_second
      self[1]
    end
    a = [1,2,3]
    a.instance_eval(array_second)
    a.second #returns: 2
```

```
  str = "ruby.devcoop.fr"
  str.instance_eval do
    def /(delimiter)
      split(delimiter)
    end
  end
  str / "." # return: ["ruby" , "devcoop", "fr"]
```

Otra forma de realizar esto es mediante los metodos `add_method`, `remove_method` y `undef_method`. Otro método útil para manipular métodos es el `alias_method` o simplemente `alias`, mediante estos métodos es posible darle sinónimos a un método existente, funciona como un diccionario en el cual un método puede tener diferentes nombres. Por otra parte, si se llama a un método no existente, se puede usar `method_missing` para capturar y manejar invocaciones arbitrarias en un objeto.

1.1.8 Ejemplo C++

En *C++* es posible lograr una sintaxis de tipo *DSL* a partir de la utilización de macros, el siguiente ejemplo permite una sintaxis estilo *bash* en *C++*:

```
#include <iostream>

#define LPAREN (
#define echo ECHO_MACRO LPAREN
#define done )

#define ECHO_MACRO(X) std::cout << (X) << "\n"

#define DSL(X) X

int main() {
    DSL(
```

```
        echo "Look ma, no brains!" done;
    )
    return 0;
}
```

1.2 Dynamic

¿Qué tal si queremos un mayor dinamismo en nuestro *DSL*?

> Vease el codigo adjunto:
> Implementacion en CSharp -> Carpeta DynamicDSl -> Clase Person

A continuación se explica como es posible lograr esto.

1.2.1 Palabra clave `dynamic`

La palabra clave `dynamic` es utilizada para indicar que una instancia esta involucrada en un ***enlace tardío*** *(Late Binding)* y que el ***DLR*** o *Dynamic Language Runtime* se encargue del manejo de este objeto. El comportamiento de este objeto durante el *enlace tardío* puede ser controlado y sobreescrito a traves de la implementación de la interfaz `IDynamicMetaObjectProvider`, el ***DLR*** se encargará de llamar a los métodos provenientes de `IDynamicMetaObjectProvider`, los cuales describen el comportamiento de la clase en el momento de enlace.

1.2.2 ¿Qué es el DLR?

Para entender que es el ***DLR*** debemos entender primero que es el ***CLR***.

1.2.2.1 Common Language Rutime (CLR)

El *Common Language Runtime* o *CLR* es un entorno de ejecución para los códigos de los programas que corren sobre la plataforma *Microsoft .NET*. El *CLR* es el encargado de compilar una forma de código intermedio (el conocido *IL*) a código de maquina nativo, mediante un compilador en tiempo de ejecución. El *CLR* también permite otros servicios importantes, incluyendo los siguientes:

- Administración de la memoria
- Administración de hilos
- Manejo de excepciones
- Recolección de basura

- Seguridad

1.2.2.2 Dynamic Language Rutime (DLR)

El *Dynamic Language Rutime* o *DLR* agrega un conjunto de servicios al *CLR* para un mejor soporte de lenguajes dinámicos. Estos servicios incluyen lo siguiente:

- **Árboles de expresión:**
  El *DLR* utiliza árboles de expresión para representar la semántica del lenguaje. Para este propósito, el *DLR* ha ampliado los árboles de expresión *LINQ* para incluir el control de flujo, la asignación y otros nodos para modelar el lenguaje.
- **Interaccion y almacenamiento en caché:**
  Mediante el dynamic call site, en un lugar en el código donde realiza una operación como `a + b` o `a.B()` en objetos dinámicos. El DLR almacena en caché las características `a` y `b` (generalmente los tipos de estos objetos) e información sobre la operación. Si dicha operación se ha realizado previamente, el DLR recupera toda la información necesaria de la memoria caché para un envío rápido.
- **Interoperabilidad dinámica de objetos**:
  El DLR proporciona un conjunto de clases e interfaces que representan objetos y operaciones dinámicos y pueden ser utilizados por implementadores de lenguaje y autores de bibliotecas dinámicas. Estas clases e interfaces incluyen `IDynamicMetaObjectProvider`, `DynamicMetaObject`, `DynamicObject` y `ExpandoObject`.

1.2.3 Enlace Tardío (Late binding)

Enlace: se le denomina a la asociación de una función con su objeto correspondiente al momento de llamado de la misma.

Enlace de tiempo de compilación, estático o temprano: es el de una función miembro, que se llama dentro de un objeto, dicho enlace se resuelve en tiempo de compilación. Todos los métodos que pertenecen a un objeto o nombre de una clase (estáticos) son a los que se pueden realizar enlaze de tiempo de compilación.

Enlace tardío o dinámico: es cuando solo se puede saber a que objeto pertenece una función, en tiempo de ejecución. Uno de los ejemplos más comunes de este tipo enlace son los metodos virtuales.

1.2.3.1 Enlace tardío con métodos dinámicos vs. métodos virtuales en CSharp

Los métodos virtuales todavía están "*vinculados*" en tiempo de compilación. El compilador verifica la existencia real del método y su tipo de retorno, y el fallará en compilar si el método no existe o existe alguna inconsistencia de tipos.

El método virtual permite el polimorfismo y una forma de enlace tardío, ya que el método se enlaza al tipo adecuado en tiempo de ejecución, a través de la tabla de métodos virtuales.

Por otro lado, con `dynamic`, no hay absolutamente ningún enlace en el momento de compilación. El método puede o no existir en el objeto destino, y eso se determinará en tiempo de ejecución.

Todo es una cuestión de terminología, `dynamic` es realmente un enlace tardío (búsqueda del método de tiempo de ejecución), mientras que `virtual` proporciona el envío del método de tiempo de ejecución a través de una búsqueda virtual, pero todavía tiene algún "*enlace temprano*" en su interior.

1.2.4 Como se logra el comportamiento dinámico en CSharp

Este comportamiento es concecuencia directa del desarrolo del *DLR* el cual fue concebido para admitir las implementaciones *"Iron"* de los lenguajes de programación *Python* y *Ruby* en *.NET*.

En en centro del entorno de ejecución *DLR* se posiciona la clase llamada DynamicMetaObject. Dicha clase implemeta los siguientes métodos para dar respuesta a como actuar en todos los posibles esenarios en los que se puede encontrar una instancia de un objecto en un momento dado:

- `BindCreateInstance`: crea o activa un objeto.
- `BindInvokeMember`: llamar a un método encapsulado.
- `BindInvoke`: ejecuta el objeto (como una función).
- `BindGetMember`: obtenga un valor de propiedad.
- `BindSetMember`: establece un valor de propiedad.
- `BindDeleteMember`: eliminar un miembro.
- `BindGetIndex`: obtener el valor en un índice específico.
- `BindSetIndex`: establece el valor en un índice específico.
- `BindDeleteIndex`: elimina el valor en un índice específico.
- `BindConvert`: convierte un objeto a otro tipo.
- `BindBinaryOperation`: invoque un operador binario en dos operandos suministrados.
- `BindUnaryOperation`: invoque un operador unario en un operando suministrado.

De manera general las clases definidas de manera ordinaria (estática) saben como reaccionar

en dichos esenarios. Pero las clases *dinámicas* no tienen estas reacciones predefinidas por lo cual es necesario predefinir para estas clases su propio `DynamicMetaObject`, el cual en tiempo de ejecución sepa que tiene que ejecutar en cada esenario. Para definir una clase *dinámica*, `System.Dynamic` proveé la interfaz `IDynamicMetaObjectProvider`, la cual contiene el metodo:

```
DynamicMetaObject GetMetaObject(Expression parameter)
```

El cual debe encargarse de retornar el `DynamicMetaObject` que describa el comportamiento de la clase *dinamica* que implemeta la interfaz `IDynamicMetaObjectProvider` según el árbol de expresiones que dicho método recibe como parámetro

1.2.5 `System.Dynamic.DynamicObject`

Como se puede observar, en principio lograr un comportamieto dinámico en **C#** pasa por crear estos `DynamicMetaObject` y tener conocimientos para trabajar sobre el árbol de expresiones de **C#**. Para evitar todo este proceso `System.Dynamic` provée la clase `DynamicObject`, pensada para poder definir comportamietos dinámicos abstraídos de todo el proceso anteriormente descrito pues ya cuenta con una implemetacion del metodo `GetMetaObject` de `IDynamicMetaObjectProvider`. Dicha imlpemetación relaciona los siguentes métodos a sus respectivos esenarios:

- ```
  public virtual bool TryBinaryOperation(BinaryOperationBinder binder, object ar
  ```

  Proporciona implementación para operaciones binarias. Las clases derivadas de la clase `DynamicObject` pueden sobreescribir este método para especificar el comportamiento dinámico para operaciones como la suma, multiplicación, etc. La clase BinaryOperationBinder contiene una `ExpressionType` con información de la operación que se realiza en el momento de llamado de esta función. Este método considera que la instancia de la clase derivada de `DynamicObject` es el operador de la derecha y *arg* es el de la izq.

- ```
  public virtual bool TryConvert(ConvertBinder binder, out object result);
  ```

  Proporciona implementación para operaciones de conversión de tipos. Las clases derivadas de la clase `DynamicObject` pueden sobreescribir este método para especificar el comportamiento dinámico de las operaciones que convierten un objeto de un tipo a otro. La clase `ConvertBinder` contiene información sobre del tipo al cual se esta tratando de hacer la conversión e incluso contiene información sobre si el proceso es explícito o implícito.

- ```
  public virtual bool TryGetIndex(GetIndexBinder binder, object[] indexes, out o
  public virtual bool TrySetIndex(SetIndexBinder binder, object[] indexes, objec
  ```
  ```

Proporciona la implementación para operaciones que obtienen (establecen) valores de miembros por índices. Las clases derivadas de la clase `DynamicObject` pueden sobreescribir este método para especificar el comportamiento dinámico para operaciones tales como obtener (establecer) un valor para una propiedad mediante unos índices específico.

- ```
  public virtual bool TryGetMember(GetMemberBinder binder, out object result);
  public virtual bool TrySetMember(SetMemberBinder binder, object value);
  ```

  Proporciona la implementación para operaciones que obtienen (establecen) valores de miembros. Las clases derivadas de la clase `DynamicObject` pueden sobreescribir este método para especificar el comportamiento dinámico para operaciones tales como obtener (establecer) un valor de una propiedad. La clase `GetMemberBinder` (`SetMemberBinder`) contiene información sobre el nombre de la propiedad en cuestión.

- ```
  public virtual bool TryInvokeMember(InvokeMemberBinder binder, object[] args,
  ```

  Proporciona la implementación para operaciones que invocan a un miembro. Las clases derivadas de la clase `DynamicObject` pueden sobreescribir este método para especificar el comportamiento dinámico para operaciones como llamar a un método. La clase `InvokeMemberBinder` contiene información sobre el nombre del miembro en cuestión y `args` es un array con los parámetros.

- ```
  public virtual bool TryInvoke(InvokeBinder binder, object[] args, out object r
  ```

  Proporciona la implementación para operaciones que invocan un objeto. Las clases derivadas de la clase `DynamicObject` pueden sobreescribir este método para especificar el comportamiento dinámico para operaciones como invocar un objeto o un delegado. La clase `InvokeBinder` contiene informacion sobre la cantidad de argumentos y sus nombres mediante una propiedad del tipo `CallInfo` y `args` son los parámetros.

1.2.6 `System.Dynamic.ExpandoObject`

Aunque la clase DynamicObject es una gran abstración del proceso base, para su utilización es necesario implementar una clase que herede de esta y realice los override necesarios, lo cual es demasiado verboso en los casos más sencillos. Suponiendo que solo se necesita de un objeto dinámico que permita un control dinámico de propiedades, mediante `DynamicObject` necesitamos implementar los metodos `TryGetMember` y `TrySetMember`. Para evitar esto `System.Dynamic` proveé la clase `ExpandoObject`, la misma es una clase `sealed` y por tanto no se puede extender.
  ```

`ExpandoObject` implementa las interfaces `IDictionary<KeyValuePair<string, object>>` y `IDynamicMetaObjectProvider` entre otras. Mediante las dos interfaces antes mencionadas dicha clase logra el manejo dinámico de las propiedades. El proceso es realmente sencillo puesto que en su interior contiene algún tipo de implementación de diccionario, al momento de asignar una propiedad guarda el nombre de la propiedad como llave y el objeto que se le esta asignando como valor. Y en el momento en que se hace referencia a una propiedad busca si el nombre de dicha propiedad se encuentra entre las llaves, en caso afirmativo se devuelve el valor correspondiente, de lo contrario se lanza un excepción.

1.3 Reflection

Reflection es la capacidad de un proceso de examinar, introspectar y modificar su propia estructura y comportamiento. Reflection ayuda a los programadores a crear bibliotecas de software genéricas para mostrar datos, procesar diferentes formatos de datos, realizar la serialización o deserialización de datos para la comunicación, o agrupar y desagrupar datos para contenedores o ráfagas de comunicación. *Reflection* hace que un lenguaje sea más adecuado para el código orientado a la red.

Tambien se puede utilizar para observar y modificar la ejecución del programa en tiempo de ejecución. Esto generalmente se logra mediante la asignación dinámica de código de programa en tiempo de ejecución. En lenguajes de programación orientados a objetos, esta técnica permite la inspección de clases, interfaces, campos y métodos en tiempo de ejecución sin conocer los nombres de las interfaces, campos y métodos en tiempo de compilación. También permite la creación de instancias de nuevos objetos y la invocación de métodos. Por lo antes dicho es claro que es también una estrategia clave para la metaprogramación .

1.3.1 Reflection en C

Reflection en C# tiene como clase principal `System.Type`; clase abstracta que representa un tipo en el *Common Type System (CTS)* lo cual representa una de las principales componentes de *CLR*. Cuando utiliza esta clase, puede encontrar los tipos utilizados en un módulo y un espacio de nombres y también determinar si un tipo dado es una referencia o un tipo de valor. Puede analizar las tablas de metadatos (Campos, Propiedades, Métodos, Eventos) correspondientes para ver estos elementos

El enlace tardío también se pueden lograr a través de *Reflection*. Un ejemplo claro: es posible que no se sepa qué assembly cargar durante el tiempo de compilación. En este caso, puede pedirle al usuario que ingrese el nombre y el tipo del ensamblado durante

el tiempo de ejecución para que la aplicación pueda cargar el assembly apropiado. Con el tipo `System.Reflection.Assembly`, tiene algunos metodos estáticos que le permiten cargar un assembly directamente: `LoadFrom`, `LoadWithPartialName`

Dentro del archivo PE (ejecutable portátil) hay principalmente metadatos, que contienen una variedad de tablas diferentes, como:

- Tabla de definición archivada
- Tabla de definición de tipo
- Tabla de definición de método

1.3.2 Api se `System.Reflection`

Para realizar la manipulación de una instacia de un objeto en *C#* mediante *Reflection* se necesita comenzar obteniendo la clase `System.Type` que describe dicha instancia, `System.Type` es la puerta principal para el uso de *Reflection* en *C#*. Para abrir esta puerta *C#* nos brindan tres formas principales:

- `typeof`

```
Type t = typeof(Person);
```

- `object.GetType()`

```
var person = new Person();
Type t = person.GetType();
```

- `Type.GetType()`

```
Type t = Type.GetType("CSharp.DynamicReflectionDSL.Person");
```

Aunque las formas antes descritas dependen de conocer la tipo del objeto que queremos modificar de antemano. Para superar esta limitacion dentro de `System.Reflection` tenemos también la clase `Assambly` la cual contiene el metodo estático `GetExecutingAssembly()` que retorna una instancia del ensamblado que se encuentra en ejecución. El cual contiene todos lo tipos en el interior del mismo y por tanto se puede realizar una busqueda entre estos tipos:

```
Type[] classes = Assembly.GetExecutingAssembly().GetTypes();
foreach (var type in classes)
  if (type.Name == "Nombre del objeto") {....}
```

Una vez ya se tiene la instancia de `System.Type` necesaria, la misma contienen una serie de metodos para interactuar con las clases de `System.Reflection` entre los cuales resaltamos:

- `GetEvent` y `GetEvents` los cuales retornan los `EventInfo` de la instancia.
- `GetField` y `GetFields` los cuales retornan los `FieldInfo` de la instancia.
- `GetMember` y `GetMembers` los cuales retornan los `MemberInfo` de la instancia.
- `GetProperty` y `GetPropertys` los cuales retornan los `PropertyInfo` de la instancia.
- `GetMethod` y `GetMethods` los cuales retornan los `MethodInfo` de la instancia.
- `GetEvent` y `GetEvents` los cuales retornan los `EventInfo` de la instancia.
- `GetConstructorInfo` y `GetConstructorInfos` los cuales retornan los `ConstructorInfo` de la instancia.

Una vez se alcanzan estas clases de `System.Reflection` solo necesitamos conocer sobre cada uno de ellos para poder usar sus distintos métodos para indagar y modificar sus metadatos. A continuación se expone la descripción de algunos de ellos:

- `EventInfo` : clase abstracta. Contiene información sobre un evento específico.
- `FieldInfo` : clase abstracta. Puede contener información sobre los miembros de datos especificados de la clase.
- `MemberInfo`: clase abstracta. Contiene información de comportamiento general para clase EventInfo , FieldInfo , MethodInfo y PropertyInfo .
- `MethodInfo` : clase abstracta. Contiene información sobre el método especificado.
- `ParameterInfo` : clase abstracta. Contiene información sobre un parámetro dado en un método dado.
- `PropertyInfo` : clase abstracta. Contiene información sobre la propiedad especificada.
- `ConstructorInfo`: clase abstracta. Contiene datos sobre los parámetros, modificadores de acceso y detalles de implementación de un constructor.

Cada uno de ellos tienen distintos métodos para acceder a los metadatos de manera diferente y de acuerdo con sus respectivos significados. En particular ejemplificamos el caso de `PropertyInfo`:

```
...
var person = new Person();
person.LastName = "Name"
PropertyInfo info = .GetProperty("LastName");
info.GetType(); //System.Reflection.PropertyInfo
a.GetValue(person); //Name
a.SetValue(person,"NewName");//person.LastName = "NewName"
a.Name;//LastName
a.PropertyType; //string
...
```