# ✨ TadPrinter ✨

TADP - 2023 - 2C - TP Metaprogramación

# Descripción del dominio

Vamos a desarrollar un framework en Ruby para facilitar el renderizado (o serialización) de objetos. Para eso, primero vamos a extender la sintaxis del lenguaje para admitir un DSL (o Lenguaje de Dominio Específico) que permita describir un XML de forma declarativa (y generarlo utilizando un código base que se provee en el anexo de este enunciado). Después vamos a automatizar este proceso para que cualquier objeto pueda serializarse sin necesidad de detallar cómo y, finalmente, vamos a hacer esta automatización customizable utilizando metadata.

# Entrega Grupal

En esta entrega tenemos como objetivo desarrollar la lógica necesaria para implementar la funcionalidad que se describe a continuación. Además de cumplir con los objetivos descritos, es necesario hacer el mejor uso posible de las herramientas vistas en clase sin descuidar el diseño. Esto incluye:

- Evitar repetir lógica.
- Evitar generar construcciones innecesarias (mantenerlo lo más simple posible).
- Buscar un diseño robusto que pueda adaptarse a nuevos requerimientos.
- Mantener las interfaces lo más limpias posibles.
- Elegir adecuadamente dónde poner la lógica y qué abstracciones modelar, cuidando de no contaminar el scope global.
- Aprovechar las abstracciones provistas por el metamodelo de Ruby.
- Realizar un testeo integral de la aplicación cuidando también el diseño de los mismos.
- Respetar la sintaxis pedida en los requerimientos.

# Nota importante:

Tomá este emoticón: ✨. Lo van a necesitar…

## Punto 1: DSL e Impresión

Lo primero que vamos a hacer es escribir el código necesario para soportar la definición de un XML. Cada **Tag** (o nodo) de XML tiene un **label** (o nombre) y puede contener **atributos** que consisten en pares clave-valor donde la clave es un string y el valor puede ser un string, un número, un booleano o null (nil, en Ruby). Además, cada Tag puede contener cualquier cantidad de otros tags como **hijos**.
Todo **Documento** de XML tiene un único tag raíz, que puede o no contener otros tag hijos.

Para definir un documento, queremos utilizar una clase ***Document*** que al instanciarse reciba un bloque con la **definición del tag raíz.** Las instancias de *Document* tienen que entender un mensaje ***xml*** que retorne un string que contenga el XML generado:

```
documento = Document.new do
  {{ definición de un tag }}
end

documento.xml   # Esto debería retornar XML
```

Queremos utilizar la siguiente notación para declarar cada tag de XML:

```
{{nombre}} {{atributo1: valor1}}, ..., {{atributoN: valorN}} do
  {{definiciónDelContenido}}
end
```

Donde la definición del contenido del tag puede ser, o bien un único **valor** (un string, número, booleano o nil) o una sucesión de cero o más definiciones de tags hijos.

Por ejemplo, el siguiente fragmento de código define el XML que describe a un alumno:

**Ruby:**

```
documento = Document.new do
      alumno nombre: "Matias", legajo: "123456-7" do
            telefono { "1234567890" }
            estado es_regular: true do
              finales_rendidos { 3 }
              materias_aprobadas { 5 }
            end
      end
end
```

**XML:**

```
<alumno nombre="Matias" legajo= "123456-7">
  <telefono>1234567890</telefono>
  <estado es_regular=true>
     <finales_rendidos>3</finales_rendidos>
     <materias_aprobadas>5<materias_aprobadas>
  </estado>
</alumno>
```

**Nota 1:** Es **MUY** importante que respeten la sintaxis exactamente como la describimos. No vale agregar, quitar o modificar nada.

**Nota 2:** No es necesario preocuparse por el formateado del xml generado, lo ponemos así en los ejemplos para que sea más fácil de entender.

**Nota 3**: Recuerden que, en Ruby, las siguientes dos notaciones son equivalentes:

```ruby
self.msj(n1: p1,  n2: p2) do ... end  # Esto...

msj n1: p1, n2: p2 { ... }            # ...es igual a esto.
```

## Punto 2: Generación automática

Un mecanismo *manual* para definir un XML es conveniente para casos aislados, cuando buscamos serializar una pieza de información poco común que no se repite en el sistema, pero, en general, resulta tedioso tener que describir cada clase de objeto que necesitamos. Teniendo esto en cuenta queremos automatizar el mecanismo de serialización para que decida automáticamente una representación razonable para cualquier objeto.

Las reglas a seguir para convertir un objeto *X* en un documento son:

- El nombre del tag raíz debe ser **el nombre de la clase de X,** en minúsculas.
- Los atributos de X que no tengan definido un **getter** se ignoran.
- Los atributos de X con getter que referencian a **Strings, Booleanos, Números o nil** se deben serializar como atributos del tag raíz.
- Los atributos de X con getter que referencian a **Arrays** de objetos de cualquier tipo deben serializarse cómo tags hijos, conteniendo un nuevo tag hijo por cada elemento del array. Estos tags deben llamarse como la clase de los valores que representan.
- Los atributos de X con getter que referencian a cualquier otro tipo de objeto se deben serializar cómo tags hijos del tag raíz, cada uno con el nombre del atributo en cuestión.

Por ejemplo, dada una instancia de la clase *Alumno*, las siguientes dos formas de generar XML deben resultar equivalentes:

**Ruby:**

```ruby
class Alumno
  attr_reader :nombre, :legajo, :estado
  def initialize(nombre, legajo, telefono, estado)
    @nombre = nombre
    @legajo = legajo
    @telefono = telefono
    @estado = estado
  end
end
```

```ruby
class Estado
  attr_reader :finales_rendidos, :materias_aprobadas, :es_regular
  def initialize(finales_rendidos, materias_aprobadas, es_regular)
    @finales_rendidos = finales_rendidos
    @es_regular = es_regular
    @materias_aprobadas = materias_aprobadas
  end
end

unEstado = Estado.new(3, 5, true)
unAlumno = Alumno.new("Matias","123456-8", "1234567890", unEstado)

documento_manual = Document.new do
  alumno nombre: unAlumno.nombre, legajo: unAlumno.legajo do
    estado finales_rendidos: unAlumno.estado.finales_rendidos,
           materias_aprobadas: unAlumno.estado.materias_aprobadas,
           es_regular: unAlumno.estado.es_regular
  end
end

documento_automatico = Document.serialize(unAlumno)

documento_manual.xml == documento_automatico.xml  # Esto debe cumplirse
```

**XML Generado:**

```xml
<alumno nombre="Matias" legajo="123456-7">
  <estado es_regular=true finales_rendidos=3 materias_aprobadas=5/>
</alumno>
```

**Nota 1:** Es **MUY** importante no repetir lógica entre el punto 1 y el punto 2. Refactoricen y reutilicen a criterio.

**Nota 2:** No hace falta preocuparse por las referencias cruzadas. Se asume que el usuario no debe usar esta funcionalidad en objetos cuyo estado contenga ciclos.

## Punto 3: Personalización y Metadata

Si bien la serialización automática es sumamente conveniente, resulta un poco rígida. ¿Qué pasa si, para cumplir con una interfaz particular, necesitamos ajustar un poco la forma en que ciertos objetos se serializan? Necesitamos una forma de introducir *"pistas"* en el código para ayudar al serializador a producir el XML que queremos.

Para esto vamos a modelar **Annotations**.

Podemos entender una *annotation* como una construcción sintáctica especial que permite agregar información (o *metadata*) a las definiciones a las que están asociadas. Nuestras annotations van a declararse con chispitas:

```
✨AnnotationAplicadaADocente✨
✨OtraAnnotationAplicadaADocente✨
class Docente

  ✨EstaAnnotationAplicaANombreYLegajo✨
  attr_accessor :nombre, :legajo
  ✨EstaAplicaACursoYTieneParametros✨(42, "foo")
  attr_accesor :cursos
  attr_accesor :criterio

  ✨YEstaOtraAlMetodEvaluarYRecibeUnBloque✨ { ... }
  def evaluar
    self.cursos.map{|curso| curso.evaluar(self.criterio) }
  end

  ...

end
```

Sí.

Podríamos haberles dejado elegir el emoticón, pero somos unos viejos rancios y exigimos que le pongan fantasía.

Cualquier palabra se considera una **Annotation** siempre y cuando empiece y termine con (✨). Cuando se usa una annotation en el código el lenguaje debe buscar una clase que se llame igual (quitando las chispitas) e intentar instanciarla con los parámetros y bloque que recibió la annotation. En caso de que la clase no exista o no pueda ser instanciada con los parámetros dados debe lanzarse una excepción acorde.

Una vez instanciada la annotation se “asocia” a la siguiente declaración que ocurra en el contexto. Se debe poder asociar annotations a definiciones de **clases**, **métodos** y llamados a ***attr_reader*** *y* ***attr_accessor***.
Una vez implementada esta extensión, extender el framework de renderizado para soportar las siguientes anotaciones:

## ✨Label✨

La anotación **Label** recibe un nombre por parámetro. Las clases anotadas con *label* utilizan el nombre dado (en lugar del propio) como nombre de Tag. Así mismo, los getters anotados utilizan el nombre dado, ya sea que se serialicen a tags o atributos.

**Ruby:**

```ruby
✨Label✨("estudiante")
class Alumno
  attr_reader :nombre, :legajo, :estado
  def initialize(nombre, legajo, telefono, estado)
    @nombre = nombre
    @legajo = legajo
    @telefono = telefono
    @estado = estado
  end

  ✨Label✨("celular")
  def telefono
    @telefono
  end
end

✨Label✨("situacion")
class Estado
  attr_reader :finales_rendidos, :materias_aprobadas, :es_regular
  def initialize(finales_rendidos, materias_aprobadas, es_regular)
    @finales_rendidos = finales_rendidos
    @es_regular = es_regular
    @materias_aprobadas = materias_aprobadas
  end
end
```

**XML Generado:**

```xml
<estudiante nombre="Matias" legajo="123456-7" celular="1234567890">
  <situacion es_regular=true finales_rendidos=3 materias_aprobadas=5 />
</estudiante>
```

Si por utilizar un label algún XML quedara con dos atributos con el mismo nombre, el serializador debe fallar adecuadamente. No hay problema con tener múltiples hijos con el mismo nombre.

## ✨Ignore✨

La anotación **Ignore** no recibe parámetros. Los mensajes y clases marcados con ignore no son utilizados por el serializador a la hora de renderizar atributos e hijos. Serializar una clase anotada con *ignore* produce un string vacío.

**Ruby:**

```ruby
class Alumno
  ✨Ignore✨
  attr_reader :nombre, :telefono

  attr_reader :legajo, :estado

  def initialize(nombre, legajo, telefono, estado, dni)
    @nombre = nombre
    @legajo = legajo
    @telefono = telefono
    @estado = estado
    @dni = dni
  end

  ✨Ignore✨
  def dni
    @dni
  end
end

✨Ignore✨
class Estado
  attr_reader :finales_rendidos, :materias_aprobadas, :es_regular
  def initialize(finales_rendidos, materias_aprobadas, es_regular)
    @finales_rendidos = finales_rendidos
    @es_regular = es_regular
    @materias_aprobadas = materias_aprobadas
  end
end
```

**XML Generado:**

```xml
<alumno legajo="123456-7" />
```

## ✨Inline✨

La anotación **Inline** se utiliza para indicar que un campo debe ser serializada como **Atributo**, incluso si es de un tipo que normalmente se serializaría como Tag. Para esto, recibe un bloque que espera un parámetro, el cual se utiliza para convertir el valor del campo en algo que pueda ser mostrado como atributo.
No hay problema con anotar con *Inline* campos que iban a serializarse como atributos de todos modos, pero no se debe permitir utilizar esta annotation en clases. Si, luego de aplicar el bloque el campo no tiene un tipo representable con un atributo, el serializador debe fallar.

**Ruby:**

```ruby
class Alumno

  ✨Inline✨ {|campo| campo.upcase }
  attr_reader :nombre, :legajo
  def initialize(nombre, legajo, telefono, estado)
    @nombre = nombre
    @legajo = legajo
    @telefono = telefono
    @estado = estado
  end

  ✨Inline✨ {|estado| estado.es_regular }
  def estado
    @estado
  end
end

class Estado
  attr_reader :finales_rendidos, :materias_aprobadas, :es_regular
  def initialize(finales_rendidos, materias_aprobadas, es_regular)
    @finales_rendidos = finales_rendidos
    @es_regular = es_regular
    @materias_aprobadas = materias_aprobadas
  end
end
```

**XML Generado:**

```xml
<alumno nombre="MATIAS" legajo="123456-7" estado=true />
```

## ✨Custom✨

La anotación **Custom** permite declarar una forma personalizada de serializar una clase utilizando el DSL definido en el punto 1, que recibe como bloque. Cuando una clase anotada con *Custom* se serializa, el serializador ignora la estrategia habitual y utiliza el formato pedido en su lugar.

Cabe aclarar que, al serializar un campo, las anotaciones aplicadas a él tienen precedencia por encima de las aplicadas a la clase de su contenido, es decir que un campo *ignorado*, *etiquetado* o *inline* que contiene una clase con serialización *custom* priorizan la aplicación de las otras anotaciones.

**Ruby:**

```ruby
class Alumno

  attr_reader :nombre, :legajo, :telefono

  ✨Label✨("situacion")
  attr_reader :estado

  def initialize(nombre, legajo, telefono, estado)
    @nombre = nombre
    @legajo = legajo
    @telefono = telefono
    @estado = estado
  end

end

✨Custom✨ do |estado|
  regular { estado.es_regular }
  pendientes { estado.materias_aprobadas - estado.finales_rendidos }
end
class Estado
  attr_reader :finales_rendidos, :materias_aprobadas, :es_regular
  def initialize(finales_rendidos, materias_aprobadas, es_regular)
    @finales_rendidos = finales_rendidos
    @es_regular = es_regular
    @materias_aprobadas = materias_aprobadas
  end
end
```

**XML Generado:**

```xml
<alumno nombre="Matias" legajo="123456-7" telefono="1234567890">
  <situacion>
    <regular>true</regular>
    <pendientes>2</pendientes>
  </situacion>
</alumno>
```

**Nota:** Todas las anotaciones descritas arriba pueden combinarse y un mismo campo o clase admite muchas anotaciones. En caso de que quede alguna duda de qué debería pasar en cierta combinación de circunstancias, preguntenle a su docente amigo.

# Anexo

Les damos la siguiente implementación para facilitarles la generación de Strings de XML. Pueden copiarla y pegarla en su proyecto:

```
class Tag
  attr_reader :label, :attributes, :children

  def self.with_label(label)
    new(label)
  end

  def initialize(label)
    @label = label
    @attributes = {}
    @children = []
  end

  def with_label(label)
    @label = label
    self
  end

  def with_attribute(label, value)
    @attributes[label] = value
    self
  end

  def with_child(child)
    @children << child
    self
  end

  def xml(level=0)
    if children.empty?
      "#{"\t" * level}<#{label}#{xml_attributes}/>"
    else
      "#{"\t" * level}<#{label}#{xml_attributes}>\n#{xml_children(level
+ 1)}\n#{"\t" * level}</#{label}>"
    end
  end

  private

  def xml_children(level)
    self.children.map do |child|
```

```
      if child.is_a? Tag
        child.xml(level)
      else
        xml_value(child, level)
      end
    end.join("\n")
  end

  def xml_attributes
    self.attributes.map do |name, value|
      "#{name}=#{xml_value(value, 0)}"
    end.xml_join(' ')
  end

  def xml_value(value, level)
    "\t" * level + if value.is_a? String
      "\"#{value}\""
    else
      value.to_s
    end
  end
end

class Array
  def xml_join(separator)
    self.join(separator).instance_eval do
      if !empty?
        "#{separator}#{self}"
      else
        self
      end
    end
  end
end
```

## Ejemplo de uso

```
Tag
      .with_label('alumno')
      .with_attribute('nombre', 'Mati')
      .with_attribute('legajo', '123456-7')
      .with_attribute('edad', 27)
      .with_child(
        Tag
```

```
            .with_label('telefono')
            .with_child('12345678')
      )
      .with_child(
        Tag
            .with_label('estado')
            .with_child(
              Tag
                  .with_label('value')
                  .with_child('regular')
              )
      )
      .with_child(Tag.with_label('no_children'))
      .xml
```

**XML generado:**

```
<alumno nombre="Mati" legajo="123456-7" edad=27>
      <telefono>
            "12345678"
      </telefono>
      <estado>
            <value>
                  "regular"
            </value>
      </estado>
      <no_children/>
</alumno>
```