

# TP Metaprogramación

## Traits empanadas 🥟



### 🎯 Objetivos y recomendaciones

El objetivo principal de este TP es el de introducirnos en el mundo de la metaprogramación aplicada a una idea particular. Gran parte del TP consiste en lograr construir abstracciones entendiendo que estamos creando una herramienta genérica, que podría utilizarse como parte de modelos de cualquier dominio.

El TP requiere la aplicación de mecanismos específicos del lenguaje asociados a la metaprogramación: introspección e intercesión, pero además requiere de conceptos de diseño orientado a objetos (en otras palabras, no tienen que olvidarse de los lineamientos que venimos usando hasta ahora).

Como siempre, la solución deberá estar acompañada de tests automatizados.

### 📖 Descripción general del Dominio

Se pide implementar un mecanismo de composición de objetos similar a “Traits” con su álgebra incluida, basándose en el paper [“Traits: Composable Units of Behaviour”](#), de Scharli, Ducasse, Nierstrasz y Black. Insistimos en que deben leer el paper para entender bien de qué se trata.



## Definición y aplicación de traits

El primer requerimiento es poder definir un trait y agregarlo a una clase. Nos gustaría poder definir un trait de la siguiente manera:

```
trait Atacante do
  def ataque
    10
  end
end
```

Y luego poder usarlo de esta forma:

```
class Fantasma
  uses Atacante

  def nombre(sufijo)
    "Casper" + sufijo
  end
end
```

Ahora si trato de usar un objeto de tipo `Fantasma`, debería ocurrir lo siguiente:

```
fantasma = Fantasma.new
fantasma.ataque      # => 10
fantasma.nombre("!") # => "Casper!"
```

De esto se desprende que al hacer `uses` de un trait deberían agregar a la clase los métodos definidos en el trait.

## Posibles conflictos

Supongamos que tenemos la siguiente clase, que usa el trait `Atacante` que definimos anteriormente:

```
class Fantasma
  def ataque
    20
  end

  uses Atacante
end
```

Luego, cuando le enviamos el mensaje `ataque` a una instancia de `Fantasma`, lo que debería ocurrir es lo siguiente:

```
fantasma = Fantasma.new
```

```
fantasma.ataque # => 20
```

Es decir, se deberían agregar a la clase los métodos definidos en el trait pero sin pisar los métodos que ya estén definidos en la clase. Esto se puede ver en el hecho de que, si bien el trait define `ataque`, la clase también, y “gana” la implementación de la clase.

## Métodos requeridos

Un trait puede especificar que requiere algunos métodos adicionales no definidos por él. Por ejemplo:

```
trait Humano do
  requires :nombre, :apellido

  def saludo
    "Hola, soy #{self.nombre} #{self.apellido}"
  end
end
```

Esto significa que esperamos que las clases que usen este trait tengan definidos los métodos `nombre` y `apellido` (ya sea porque los definió la propia clase, o porque los obtuvo a partir de otros traits).

Por ejemplo, la siguiente clase cumple con estos requisitos, ya que define ambos métodos:

```
class Ciudadano
  def nombre
    "Juan"
  end

  uses Humano


  def apellido
    "Pérez"
  end
end

Ciudadano.new.saludo # => "Hola, soy Juan Pérez"
```

En caso de que un método requerido no esté definido, se deberá lanzar una excepción al recibir el mensaje correspondiente:

```
class Ciudadano
  uses Humano
end

Ciudadano.new.saludo # ¡Excepción!
```

 Nota: si un método requerido está definido en la jerarquía de la clase que usa los traits, esto cuenta para satisfacer el requerimiento:

```
class ConNombre
  def nombre
    "Juan"
  end
  def apellido
    "Pérez"
  end
end

class Ciudadano < ConNombre
  uses Humano
end

Ciudadano.new.saludo # => "HoLa, soy Juan Pérez"
```

## Operaciones

### Composición de Traits

Se desea ahora agregar la operación de composición de traits. Esta operación debe permitir combinar dos traits y agregar a la clase los métodos de ambos traits:

```
trait Atacante do
  def ataque
    10
  end
end

trait Defensor do
  def defensa
    20
  end
end

class Guerrero
  uses Atacante + Defensor
end

guerrero = Guerrero.new
guerrero.ataque      # => 10
guerrero.defensa     # => 20
```

La operación de composición de traits debe ser conmutativa, asociativa e idempotente.

## Posibles conflictos

Un conflicto surge cuando se componen traits que tienen algún método con el mismo nombre. Por ejemplo:

```
trait Atacante do
  def recuperarse
    "recuperarse como Atacante"
  end
end

trait Defensor do
  def recuperarse
    "recuperarse como Defensor"
  end
end

class Guerrero
  uses Atacante + Defensor # ¡Excepción!
end
```

La excepción a esto es cuando, si bien hay métodos con el mismo nombre, son *el mismo* método (es decir, vienen del mismo trait). Por ejemplo:

```
class Guerrero
  uses Atacante + Atacante # No debería lanzar una excepción, es
  equivalente a usar solo Atacante
end
```

Otro ejemplo:

```
class Guerrero
  uses (Atacante + ConNombre) + (ConNombre + Atacante) # No debería
  lanzar una excepción, es equivalente a usar Atacante + ConNombre
end
```

## Interacción con métodos requeridos

Si hay métodos requeridos, éstos pueden satisfacerse al componer el trait que los requiere con otros que definan esos métodos. Por ejemplo:

```
trait Humano do
  requires :nombre, :apellido

  def saludo
    "Hola, soy #{self.nombre} #{self.apellido}"
  end
end
```

```

trait ConNombre do
  def nombre
    "Juan"
  end
end

trait ConApellido do
  def apellido
    "Pérez"
  end
end

class Ciudadano
  uses ConNombre + Humano + ConApellido
end

Ciudadano.new.saludo # => "HoLa, soy Juan Pérez"

```

## Excluir método de un trait

La siguiente operación nos permite obtener un nuevo trait que sea igual a otro pero excluyendo cierto método.

Entonces, se podría resolver un conflicto de la siguiente manera:

```

class Guerrero
  uses Atacante + (Defensor - :recuperarse)
end

guerrero = Guerrero.new
guerrero.recuperarse # => "recuperarse como Atacante"

```

O, alternativamente, si no quisiéramos ninguna implementación para recuperarse

```

class Guerrero
  uses (Atacante + Defensor) - :recuperarse
end

guerrero = Guerrero.new
guerrero.respond_to?(:recuperarse) # => false

```

## Soporte para excluir múltiples métodos

Queremos también poder excluir más de un método a la vez. Para especificar esto en forma más concisa, tenemos que poder hacer lo siguiente:

```

UnTrait - [:metodo1, :metodo2, :metodo3]

```

Por supuesto, también podría hacerse utilizando múltiples veces la operación de exclusión:

```
UnTrait - :metodo1 - :metodo2 - :metodo3
```

## Definición de alias de métodos

Por último, se pide tener la operación de “alias” para poder ponerle un nombre adicional a un método. Es decir, que se incluya otro mensaje para el mismo método, con el nombre que nosotros especificamos. Esto nos permite resolver los conflictos sin perder acceso a los métodos conflictivos.

Veamos cómo resolver el caso anterior, en el cual queremos implementar `recuperarse` invocando las dos implementaciones de los traits `Atacante` y `Defensor`

```
class Guerrero
  uses
    ((Atacante << {recuperarse: :recuperarse_como_atacante}) - :recuperarse)
    +
    ((Defensor << {recuperarse: :recuperarse_como_defensor}) - :recuperarse)

  def recuperararse
    recuperararse_como_atacante + " y " + recuperararse_como_defensor
  end
end

guerrero = Guerrero.new

guerrero.recuperarse
# => "recuperarse como Atacante y recuperararse como Defensor"
```



## Estrategias de resolución de conflictos

Además de las operaciones existentes, se desea tener distintas estrategias ya programadas para resolver conflictos. Estas estrategias deben definirse **por cada método conflictivo**.

Las estrategias de resolución a implementar son:

1. Que, en caso de haber conflicto entre dos métodos de traits al combinarse, se tome automáticamente el método definido por uno de los traits.
2. Que ejecute todos los métodos conflictivos.  
Por ejemplo, en el caso de combinar `Atacante` con `Defensor`, querríamos que al enviar el mensaje `recuperarse` a una instancia de la clase que use el trait resultante, se ejecute el método `recuperarse` definido en `Atacante` y el método `recuperarse` definido en `Defensor`.
3. Hacer un *inject/reduce*, para combinar el resultado de haber evaluado los métodos conflictivos. Para esto, tenemos que recibir como argumentos un valor inicial y una función que cumpla el rol de combinar un acumulador con cada uno de los resultados de los métodos.

Por ejemplo, supongamos que tenemos un TraitA que define `m1` y retorna un número, y otro TraitB que también define `m1` y retorna otro número, queremos que al combinarlos “se sumen” ambos valores. Entonces, para combinarlos, pasamos al cero (0) como valor inicial, y la función de suma (`{|acc, n| acc + n }`).

4. Que se permita la definición de estrategias adicionales al usuario, además de las ya definidas.

## Bonus

### Descripción de traits

Los traits deberán entender el mensaje `description`, devolviendo una descripción del mismo. En el caso de los traits simples, la descripción es su nombre. En el caso de los traits compuestos, la descripción representa una manera de recrear el trait.

Ejemplos:

```
Atacante.description # => "Atacante"
(Atacante + Defensor).description # => "Atacante + Defensor"
(Atacante - :recuperarse).description # => "Atacante - :recuperarse"
```

### Uso de traits por otros traits

Los traits pueden usar otros traits. Por ejemplo:

```
trait ConNombre do
  def nombre
    "Atila"
  end
end

trait Atacante do
  uses ConNombre
  def ataque
    10
  end
end

class Guerrero
  uses Atacante
end

atila = Guerrero.new
atila.ataque # => 10
atila.nombre # => "Atila"
```

Esto es semánticamente equivalente a:



```
trait ConNombre do
  def nombre
    "Atila"
  end
end

trait Atacante_Parcial do
  def ataque
    10
  end
end

Atacante = ConNombre + Atacante_Parcial

class Guerrero
  uses Atacante
end

atila = Guerrero.new
atila.ataque # => 10
atila.nombre # => "Atila"
```

## Anexo: cómo hacer algunas cosas con Ruby



### Definir una constante dinámicamente

Pueden definirse constantes dinámicamente usando el mensaje `Object.const_set(constante, objeto)`.

Ejemplo:

```
n = 40
nombre = :ObjetoMagico
Object.const_set(nombre, n)
# o Object.const_set(:ObjetoMagico, 40)

ObjetoMagico + 2 # => 42
```

### Hacer algo si no se encuentra una constante

Ruby envía el mensaje `const_missing` cada vez que se intenta acceder a una constante que no está definida, pasándole el nombre de la constante como argumento, y el resultado se devolverá como el valor de la constante que no se había encontrado. Si redefinimos el método `const_missing`, podemos implementar un comportamiento personalizado que se ejecutará cuando esto suceda. Pueden leer más en [la documentación de Ruby](#).

Por ejemplo, para devolver 2 como valor de todas las constantes que sean referenciadas sin haberse definido anteriormente, podemos escribir:

```
def Object.const_missing(name)
  2
end
```

## Comportamiento *custom* para operadores

En Ruby, cuando hacemos `2 + 3`, estamos enviando un mensaje como cualquier otro (en este caso el mensaje `+`, enviado al objeto `2` y el objeto `3` como argumento). Por lo tanto, si queremos que nuestros objetos se comporten de una manera particular al recibir estos mensajes sólo debemos definir los métodos correspondientes.

Ejemplo:

```
class Point
  attr_reader :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end

  def +(aPoint)
    self.class.new(@x + aPoint.x, @y + aPoint.y)
  end
end

p1 = Point.new(1, 2)
p2 = Point.new(3, 4)
p3 = p1 + p2
p3.x # => 4
p3.y # => 6
```