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 que requiere diseño de OOP tradicional (es decir no olvidaremos los lineamientos hasta ahora utilizados) pero que además requiere de mecanismos específicos del lenguaje asociados a la metaprogramación: *reflection* y *self-modification*. Cabe aclarar que es también y quizás aún más importante tener en claro un diseño que involucre una separación entre los ejemplos o dominio "base" y la **solución genérica** que deberán implementar (el metamodelo).

En conclusión: gran parte del TP consiste en lograr la abstracción necesaria para entender que estamos creando una herramienta genérica, que podría utilizarse con cualquier dominio, y saber expresar eso en la solución.

Como siempre la solución deberá estar acompañada de test cases.

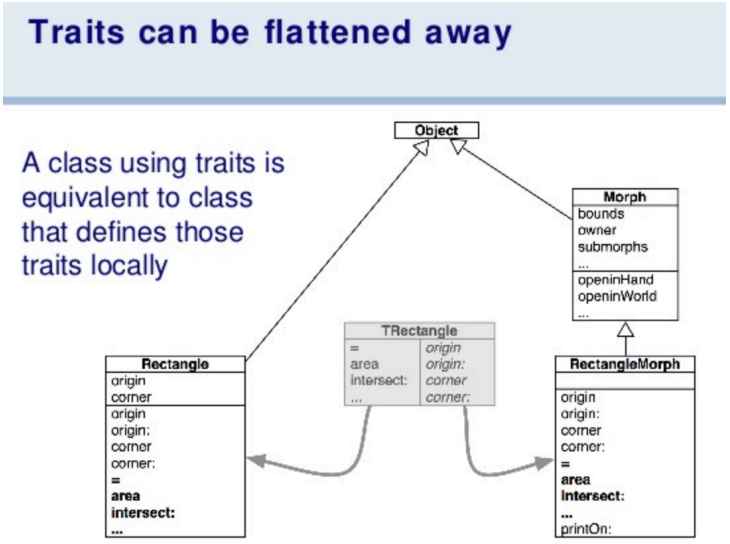
Descripción general del Dominio

Se pide implementar un mecanismo de composición de objetos similar a "Traits" con su álgebra incluída, basándose en el paper *"Traits: Composable Units of Behaviour"*, de Scharli, Ducasse, Nierstrasz y Black. A continuación un pequeño resumen nuestro del mismo:

- Los "Traits" aquí referenciados no son los traits de Scala. Es un poco nefasto pero lo que en Scala llaman "traits" son en realidad, para la teoría de computación "mixins".
- Traits es entonces un mecanismo alternativo para componer clases/comportamiento, propuesto en el paper antes citado e implementado en Pharo Smalltalk.
- Tienen bastantes puntos en común con mixins
    - Representan una forma de definir comportamiento "parcial" en pequeñas unidades reutilizables.
    - Podemos aplicarlos a cualquier clase y así evitar las limitaciones de la herencia simple.
    - Una clase puede combinar *múltiples* traits
    - Un trait puede requerir ciertos mensajes, es decir se envía mensajes a "self" que luego alguien implementa: la clase base, o alguna de sus superclases, o bien otro trait.
- Diferencias con mixins
    - Los traits no permiten definir estado (variables de instancia), es decir sólo definen comportamiento (métodos).
    - La forma en la que se aplican a la clase es completamente distinta: no involucra linearización, ni un mecanismo de dispatching especial. Simplemente los métodos se inyectan en la clase, "aplanándolos". Es decir que luego en tiempo de ejecución sólo

existe la clase que ahora contiene un montón de métodos más que salieron de los traits. Los traits simplemente desaparecen o no tienen importancia al ejecutar el programa. Fueron simplemente herramientas para “armar clases”.

- Como los traits se “aplanan” necesitamos un mecanismo para evitar conflictos o definir cómo combinarlos en forma correcta. A esto se lo llama “álgebra de traits” y vamos a ir viéndolo en cada punto del TP. Aunque nuevamente insistimos en que deben leer el paper para entender bien de qué se trata.



Vemos como una clase generada con un Trait debería ser equivalente a haber definido los métodos en ella misma, ya que el mecanismo simplemente “copia” los métodos y los incrusta en la clase.

Operaciones

Se mostrará aquí a modo de ejemplo la forma de definir un nuevo trait. Es solamente una forma sugerida y puede reemplazarse o adaptarse a otra más conveniente al diseño de cada trabajo práctico.
Lo importante de estos puntos es la definición de la funcionalidad que deberán implementar. No es necesario seguir la misma forma de definirlos.

1) Definición de trait y aplicación

El primer requerimiento es poder definir un trait y agregarlo a una clase. La definición de un Trait podría ser como la que sigue:

```
Trait.define do
   name :MiTrait

   method :metodo1 do
```

```
    "hola"
  end

  method :metodo2 do |un_numero|
    un_numero * 0 + 42
  end
end
```

Luego para poder usarlo se debería hacer:

```
class MiClase
  uses MiTrait

  def metodo1
    "mundo"
  end
end
```

(nótese que el "**uses**" es un mensaje nuevo que deberían implementar uds)

Ahora si trato de usar un objeto de tipo MiClase debería ocurrir:

```
o = MiClase.new
o.metodo1          # Devuelve "mundo"
o.metodo2(33)      # Devuelve 42
```

De esto se desprende que al hacer uses de un trait, 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 en que el trait define **metodo1**, pero la clase también, y "gana" la implementación de la clase (lo mismo sucede en mixins).

2) Sumar Traits

Se desea ahora agregar la operación de suma (composición) de traits. Esta operación debe permitir combinar dos traits y agregar a la clase los métodos de ambos traits. Si hay algún método de los traits que esté repetido entre sí, debe crear un método que tire una excepción:

```
Trait.define do
  name :MiOtroTrait

  method :metodo1 do
    "kawuabonga"
  end

  method :metodo3 do
    "mundo"
```

```
  end
end

class Conflicto
  uses MiTrait + MiOtroTrait
end

o = Conflicto.new
o.metodo2(84)       # Devuelve 42
o.metodo3           # Devuelve "mundo"
o.metodo1           # Tira una excepcion
```

Nótese que estamos usando **MiTrait**, es decir el primer trait que ya vimos, que definía **metodo1** y **metodo2.** Este nuevo trait ConConflicto aporta **metodo1** (mismo nombre -> conflicto) y **metodo3**.

3) Resta de selectores

(nota: en smalltalk se denomina “selector” al nombre del mensaje)

La siguiente operación nos permite eliminar métodos en la aplicación de un trait (sólo para la aplicación del mismo). Entonces el conflicto del ejemplo anterior se podría resolver como sigue:

```
class TodoBienTodoLegal
  uses MiTrait + (MiOtroTrait - :metodo1)
end

o = TodoBienTodoLegal.new
o.metodo2(84)       # Devuelve 42
o.metodo3           # Devuelve "mundo"
o.metodo1           # Devuelve "hola"
```

Es decir que la resta es un mensaje del siguiente tipo `Trait - Symbol` donde Symbol es el nombre del método que NO queremos heredar el trait.

3.1) Incluso deberíamos poder soportar más de un symbol, para especificar en forma más concisa múltiples métodos a omitir

`MiOtroTrait - [:metodo1, :metodo2, :metodo3]`

3.2) Y obviamente debería permitir hacer esto mismo utilizándo múltiples veces la operación de resta

`MiOtroTrait - :metodo1 - :metodo2) - :metodo3`

3) Renombrar selectores (aliases)

Por último se pide tener la operación de “alias” para poder usar los métodos conflictivos. Es decir que no los omitimos en la composición, sólo que le indicamos al framework que incluya otro método con el nombre que nosotros especificamos, que sea un alias al original. Así esto nos permite resolver los conflictos como nosotros querramos, teniendo acceso a las implementaciones de los traits.

Veamos como resolveríamos el caso anterior, en el cual queremos implementar metodo1 invocando las dos implementaciones de los traits MiTrait y MiOtroTrait

```
class ConAlias
   uses (MiTrait << (:metodo1 > :m1MiTrait)) + (MiOtroTrait << (:metodo1 > :m1MiOtroTrait))

  def metodo1
    m1MiTrait()
    m1MiOtroTrait()
  end

end

o = ConAlias.new
o.saludo            # Devuelve "hola"
o.metodo1           # Devuelve "hola"
o.metodo2(84)    # Devuelve 42
```

La sintaxis en este ejemplo es

`Trait << ( Symbol > Symbol)`

Es decir

`Trait << ( nombreMensajeOriginal > nuevoNombre)`

4) Bonus: Estrategias de resolución de conflictos

Además del álgebra existente, se desea tener distintas estrategias ya programadas para resolver conflictos.
Estas estrategias deben definirse **por cada método conflictivo antes de aplicarse los traits**.

Por ejemplo, una estrategia sería que en caso de haber conflicto, se genere un método que llame a cada mensaje conflictivo de cada trait. Entonces si se suman el trait **T1** y el trait **T2** y ambos tienen un mensaje “m” entonces en la clase se deberá generar un mensaje “m” que llame a **t1_m** y luego a **t2_m**, siendo estos alias a los respectivos métodos de los traits.

Las estrategias de resolución mínimas a implementar son:

- a) Que ejecute todos los mensajes conflictivos en orden de aparición.
- b) Que aplique una función (que viene por parámetro) al resultado de todos ellos y devuelva ese valor. Esto sería análogo a hacer un **fold** (tomar el resultado del primer método como valor inicial). La función cumple el rol de definir como "mergeamos" los resultados de los métodos de los traits. Supongamos que tenemos un Trait Berseker (deja vu?) que define el calcular daño de cierta forma, y retorna un número, y otro trait Demente que genera otro valor, quizás queramos al combinarlos que "se sumen" ambos valores. Entonces especificamos la función de suma. O bien queremos quedarnos con el mayor valor, entonces usaríamos un bloque |a, b| [a,b].max
- c) Que vaya llamando a los métodos conflictivos pero aplicando una condición con el último valor de retorno para saber si devolver ese valor o si probar con el siguiente método. Por ejemplo: Se puede pasar una función que compare si un número es positivo. Entonces si tenemos un conflicto con 3 mensajes t1_m, t2_m, t3_m, se llamará primero a t1_m y se aplica la función. Si t1_m devuelve 5, se devuelve 5. Sino se llamará a t2_m , y así sucesivamente.
- d) Que el framework permita definir estrategias al usuario del framework además de las ya definidas en el mismo.

Anexo: Tips de Ruby

Acá un par de tips de cosas específicas (oscurantismo) de ruby

Definir una constante dinámicamente

Pueden definirse constantes dinámicamente usando el mensaje **Object.const_set(constante, objeto)**
Ejemplo:

```
o = 40
nombre = :ObjetoMagico

Object.const_set(nombre, o)        # o lo mismo Object.const_set(:ObjetoMagico, 40)

ObjetoMagico + 2                   # esto da 42
```

Sobrecarga de operadores

En ruby la "sobrecarga de operadores" no es más que la definición de un método normal, solo que el nombre del mismo es el símbolo (Por ejemplo + − << >). Y el método se definirá en la clase del primer objeto usado en la operación (parte izquierda). Lo cual es el receptor del mensaje.
Ejemplo

```ruby
class Persona
   attr_accessor :edad
   def initialize(_edad)
      edad = _edad
   end

   def >(otra_persona)
      edad > otra_persona.edad
   end

end

p1 = Persona.new(28)
p2 = Persona.new(18)

p1 > p2     # Esto da true
```