Traits

TAdeP - 1C 2020 - Trabajo Práctico Grupal 1: Metaprogramación

Descripción general del Dominio

Se pide implementar una composición de objetos similar a Traits con su álgebra incluída, basándose en el paper ***Traits: Composable Units of Behaviour***, de Schärli, Ducasse, Nierstrasz y Black.

En la sección operaciones se muestra 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. Por ejemplo, los traits podrían definirse directamente como modules de Ruby, y luego en el momento de la composición hacer los ajustes necesarios para que se aplique el álgebra.

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.

Requerimientos

Operaciones

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 deberá hacer:

```
class MiClase uses MiTrait
  def metodo1
    "mundo"
  end
end
```

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.

2. Suma 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 y la clase que usa esos traits no tiene una definición propia del mismo, lo esperable al enviar ese mensaje es que tire una excepción:

```
Trait.define do
  name :MiOtroTrait
  method :metodo1 do
    "kawuabonga"
  end
```

```
  method :metodo3 do
    "zaraza"
  end
end

class Conflicto
  uses MiTrait + MiOtroTrait
end

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

3. Resta selectores

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

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

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

4. Renombrar selectores

Por último se pide tener la operación de alias para poder usar los métodos conflictivos:

```
class ConAlias
  uses MiTrait << (:metodo1 >> :saludo)
end
```

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

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 traits.

Las estrategias de resolución mínimas a implementar son:

- Que ejecute todos los mensajes conflictivos en orden de aparición.
- 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).
- Que vaya llamando 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.

BONUS: Que el framework permita definir estrategias al usuario del framework además de las ya definidas en el mismo.

Referencias del lenguaje

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)
ObjetoMagico + 2 # esto da 42
```

Sobrecarga de operadores

Puede definirse para los operadores (+ − << >) un método en el objeto que se necesite cuyo nombre sea el operador, permitiendo usarlos con los objetos que tengan ese método:

```
class Persona
  def >(otra_persona)
    false
  end
end
Persona.new > Persona.new # Esto da falso
```