Técnicas Avanzadas de Programación – UTN – FRBA

1er cuatrimestre 2014

Trabajo Práctico Cuatrimestral

1 Objetivos y recomendaciones

- Desarrollar un framework en objetos de complejidad media.
- La aplicación se debe construir en Ruby.
- La solución debe presentar buenas prácticas de diseño, tales como evitar repetición de código, buena distribución de responsabilidades, manejo adecuado de excepciones, etc.
- El programa debe funcionar y se mostrará su funcionamiento a través de los test cases que sean necesarios.
- Junto con el programa construido se deberá entregar la documentación que refleje el diseño de la solución. Esta documentación deberá incluir los diagramas que muestren la estructura del sistema, justificación de las decisiones de diseño que tomaron y toda otra explicación que consideren necesaria.
- El 3 de Mayo se deberá presentar una entrega parcial para validar los avances con el tutor asignado. Es imperativo que para esa entrega ya haya código escrito. Caso contrario no se considerará que hubo avance.
- El 17 de Mayo se corregirá el TP en clase, no se aceptarán entregas fuera de término.
- Todas las semanas el grupo debe reunirse con el tutor para mostrar avances e intercambiar ideas.
- Es obligatorio que todos los integrantes participen del diseño y de la codificación, el tutor tendrá la potestad de realizar una evaluación personal en caso de ser necesario.
- Aprovechen al tutor para hacer el TP, háganle preguntas, no lo vean solo como "el que corrije", debe servirles como guía para llegar a la mejor solución posible.

2 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.

3 Operaciones

3.1 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.

3.2 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, 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
```

3.3 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 "mundo"
o.metodo1 # Devuelve "hola"
```

3.4 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
```

4 Resolución de conflictos

Además del álgebra existente, se desea tener distintas estrategias ya programadas para resolver conflictos.

Estás 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 comparé si un número es positivo. Entonces si tenemos un conflicto con 3 mensajes t1_m, t2_m, t_3m, 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.

5 Referencias del lenguaje

5.1 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
```

5.2 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
```