Programación Orientada a Aspectos

TADP - 2015 C2 - TP Metaprogramación

Descripción del dominio

La Programación Orientada a Aspectos (o AOP) es una extensión al paradigma Orientado a Objetos cuya intención es permitir una modularización más flexible y posibilitar una separación de responsabilidades transversales al dominio en módulos independientes que, con la ayuda de herramientas de preprocesado, se integran en caliente al código del sistema "alterando" el modelo original.

Para poder lograr esto es necesario trabajar en un entorno que soporte definir e identificar puntos de inserción en el código y *"tejerlos"* con la lógica deseada, para generar el nuevo modelo.

Este trabajo práctico no tiene como objetivo enseñar o aplicar AOP sino profundizar y ejercitar los temas vistos en clase, aplicándolos para construir un framework sencillo en Ruby que provea las herramientas mínimas necesarias para soportar dichas ideas.

Entrega Grupal

En esta entrega tenemos como objetivo extender Ruby, proveyendo las herramientas necesarias para definir comportamiento transversal a un dominio de negocio y aplicarlo a un programa estándar. Para esto vamos a definir un framework que permita filtrar los métodos de determinados **orígenes** que cumplan con ciertas ***condiciones*** definidas por el usuario y, posteriormente, permita realizar **transformaciones** sobre ellos.

Nota: 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.
- Realizar un testeo integral de la aplicación cuidando también el diseño de los mismos.

1. Orígenes

Llamaremos *"Origen"* al dominio de nuestros aspectos. Un *Origen* puede estar definido como uno o más objetos, módulos o clases (sin repetidos) sobre cuyos métodos nos interesa realizar transformaciones.

Se pide agregar la lógica necesaria para poder definir un origen con la siguiente sintaxis:

```
Aspects.on MiClase do
  # definición de aspectos para las instancias de MiClase
end

Aspects.on MiModulo do
  # definición de aspectos para las instancias de MiModulo
end

Aspects.on miObjeto do
  # definición de aspectos para miObjeto
end

Aspects.on UnaClase, UnModulo, OtroModulo do
  # definición para las instancias de UnaClase, UnModulo u OtroModulo
end
```

Además de referencias directas a clases o módulos, debe ser posible crear un Origen a partir de ***Expresiones Regulares***. Estas definiciones incluyen en el Origen todas las clases o módulos cuyo nombre matchee al menos una de las expresiones dadas.

```
Aspects.on MiClase, /^Foo.*/, /.*bar/ do
   # definición para las instancias de MiClase y cualquier clase o
módulo
  # cuyo nombre comience con "Foo" o termine con "bar"
end
```

El framework no debe permitir la definición de un Origen vacío, ya sea porque se use la expresión *on* sin parámetros o porque ninguno de ellos se resuelva a una fuente válida.

```
Aspects.on do
  # ...
end
# ArgumentError: wrong number of arguments (0 for +1)

Aspects.on /NombreDeClaseQueNoExiste/ do
  # ...
end
```

```
# ArgumentError: origen vacío

Aspects.on /NombreDeClaseQueNoExiste/, /NombreDeClaseQueSíExiste/ do
  # ...
end
# Exito!
```

2. Condiciones

Nuestro framework debe permitir filtrar los métodos de un Origen en donde realizar una transformación. Para eso deben implementarse una serie de ***Condiciones*** que permitan filtrar los métodos sobre los cuales actuar.

En este punto se pide implementar un mensaje ***where***, accesible desde dentro del contexto de los *Orígenes*, que reciba un conjunto de condiciones por parámetro.
Los métodos retornados no solo serán aquellos definidos en la clase o módulo inmediato, sino todos los de su jerarquía.

```
Aspects.on MiClase, miObjeto do
  where <<Condicion1>>, <<Condicion2>>, ..., <<CondicionN>>
   # Retorna los métodos que entienden miObjeto y las instancias de
MiClase
  # que cumplen con todas las condiciones pedidas
end
```

Las *Condiciones* que deben ser soportadas se listan a continuación (aunque se debe tener en cuenta que esta lista podría crecer en el futuro, queda en manos del grupo encontrar el mejor diseño para representar cada una de estas abstracción).

a. Selector

Esta condición se cumple cuando el selector del método respeta una cierta regex.

```
class MiClase
  def foo
  end

  def bar
  end
end

Aspects.on MiClase do
  where name(/fo{2}/)
  # array con el método foo (bar no matchea)

  where name(/fo{2}/), name(/foo/)
  # array con el método foo (foo matchea ambas regex)

  where name(/^fi+/)
  # array vacío (ni bar ni foo matchean)

  where name(/foo/), name(/bar/)
  # array vacío (ni foo ni bar matchean ambas regex)
end
```

b. Visibilidad

Se cumple si el método es privado o público.

```
class MiClase
  def foo
  end

  private

  def bar
  end
end

Aspects.on MiClase do
  where name(/bar/), is_private
  # array con el método bar

  where name(/bar/), is_public
  # array vacío
end
```

c. Cantidad de Parámetros

Debe poder establecerse una condición que se cumpla si el método tiene **exactamente** N parámetros **obligatorios**, **opcionales** o **ambos**.

```
class MiClase
 def foo(p1, p2, p3, p4='a', p5='b', p6='c')
 end
 def bar(p1, p2='a', p3='b', p4='c')
 end
end

Aspects.on MiClase do
  where has_parameters(3, mandatory)
  # array con el método foo

  where has_parameters(6)
  # array con el método foo

  where has_parameters(3, optional)
  # array con los métodos foo y bar
end
```

d. Nombre de Parámetros

Esta condición se cumple si el método tiene **exactamente** N parámetros cuyo nombre cumple cierta regex.

```
class MiClase
  def foo(param1, param2)
  end

  def bar(param1)
  end
end

Aspects.on MiClase do
  where has_parameters(1, /param.*/)
  # array con los el método bar

  where has_parameters(2, /param.*/)
  # array con el método foo

  where has_parameters(3, /param.*/)
  # array vacío
end
```

e. Negación

Esta condición recibe otras condiciones por parámetro y se cumple cuando ninguna de ellas se cumple.

```
class MiClase
  def foo1(p1)
  end
  def foo2(p1, p2)
  end
  def foo3(p1, p2, p3)
  end
end

Aspects.on MiClase do
  where name(/foo\d/), neg(has_parameters(1))
  # array con los métodos foo2 y foo3
end
```

Nota: Usamos “neg” como nombre de la condición en vez de “not” porque el “not” de Ruby toma precedencia.

3. Transformaciones

Se pide aplicar transformaciones sobre todos los métodos que matchean todas las condiciones.

```
Aspects.on MiClase, miObjeto do
  transform(where <<Condicion1>>, <<Condicion2>>, ..,<<CondicionN>>) do
    <<Transformación1>>
    <<Transformación2>>
    ...
  end
end
```

Cuando los orígenes son módulos o clases, todas sus instancias se verán afectadas. Cuando el origen sea una instancia, sólo ésa misma será afectada por las transformaciones.

Las *Transformaciones* que deben ser soportadas se listan a continuación (aunque se debe tener en cuenta que esta lista podría crecer en el futuro, queda en manos del grupo encontrar el mejor diseño para representar cada una de estas abstracciones).
Debe ser posible aplicar múltiples transformaciones (sucesivas o no) para las mismas Condiciones u Origen.

a. Inyección de parámetro

Esta *Transformación* recibe un hash que representa nuevos valores para los parámetros del método. Al momento de ser invocado, los parámetros con nombres definidos en el hash deben ser sustituidos por los valores presentes en el mismo.

```
class MiClase
  def hace_algo(p1, p2)
    p1 + '-' + p2
  end
  def hace_otra_cosa(p2, ppp)
    p2 + ':' + ppp
  end
end

Aspects.on MiClase do
  transform(where has_parameters(1, /p2/)) do
    inject(p2: 'bar')
  end
end

instancia = MiClase.new
```

```
instancia.hace_algo("foo")
# "foo-bar"

instancia.hace_algo("foo", "foo")
# "foo-bar"

instancia.hace_otra_cosa("foo", "foo")
# "bar:foo"
```

Si el valor a inyectar es un Proc, no se debe inyectar el Proc, sino el resultado de ejecutarlo, pasando como parámetros al objeto receptor del mensaje, el selector del método y el parámetro original.

```
class MiClase
  def hace_algo(p1, p2)
    p1 + "-" + p2
  end
end

Aspects.on MiClase do
  transform(where has_parameters(1, /p2/)) do
    inject(p2: proc{ |receptor, mensaje, arg_anterior|
      "bar(#{mensaje}->#{arg_anterior})"
    })
  end
end

MiClase.new.hace_algo('foo', 'foo')
# 'foo-bar(hace_algo->foo)'
```

b. Redirección

Esta transformación recibe un objeto sustituto por parámetro. Al momento de ser invocado el método, en lugar de ser ejecutado sobre el receptor original, debe ejecutarse sobre el sustituto.

```
class A
  def saludar(x)
    "Hola, " + x
  end
end

class B
```

```
  def saludar(x)
    "Adiosín, " + x
  end
end

Aspects.on A do
  transform(where name(/saludar/)) do
    redirect_to(B.new)
  end
end

A.new.saludar("Mundo")
# "Adiosín, Mundo"
```

c. Inyección de lógica

Esta transformación recibe un bloque con una extensión al código del método original. Cuando el método en cuestión sea invocado, el bloque recibido debe ejecutarse **Antes**, **Después** o **En Lugar De** el código original del método.

```
class MiClase
  attr_accessor :x
  def m1(x, y)
   x + y
  end
  def m2(x)
   @x = x
  end
  def m3(x)
   @x = x
  end
end

Aspects.on MiClase do
transform(where name(/m1/)) do
 before do |instance, cont, *args|
   @x = 10
   new_args = args.map{ |arg| arg * 10 }
   cont.call(self, nil, *new_args)
 end
end
transform(where name(/m2/)) do
 after do |instance, *args|
   if @x > 100
     2 * @x
```

```
    else
      @x
    end
  end
end
transform(where name(/m3/)) do
  instead_of do |instance, *args|
    @x = 123
  end
end
end

instancia = MiClase.new
instancia.m1(1, 2)
# 30
instancia.x
# 10

instancia = MiClase.new
instancia.m2(10)
# 10
instancia.m2(200)
# 400

instancia = MiClase.new
instancia.m3(10)
instancia.x
# 123
```

Nota: Debe ser posible aplicar múltiples transformaciones (sucesivas o no) para las mismas Condiciones u Origen.

```
class A
  def saludar(x)
    "Hola, " + x
  end
end

class B
  def saludar(x)
    "Adiosín, " + x
  end
end
```

```ruby
Aspects.on B do
  transform(where name(/saludar/)) do
    inject(x: "Tarola")
    redirect_to(A.new)
  end
end

B.new.saludar("Mundo")
# "Hola, Mundo"
```