TP 2 - Aspectos

Introducción

El objetivo de este TP es crear un reducido framework de ***aspectos*** (leer esta entrada para una introducción mas completa: http://paco.uqbar-project.org/conceptos/aop).
Un framework de aspectos permite componer las definiciones de comportamiento de forma diferente a los ya conocidos de clases o mixins. La idea es que al existir ciertos comportamientos que son transversales al dominio, es necesaria una forma alternativa para poder describirlos sin tener que duplicar código.
Por ejemplo, si se quisiera saber cuanto tiempo tarda en ejecutar cada método de cada clase de mi dominio, habrá que tocar todas las clases para que guarden el instante de tiempo en cuanto empezaron a ejecutar, el instante de tiempo cuando terminaron y luego hacer la cuenta. Claramente programarlo de esta forma será complicado, propenso a errores y altamente repetitivo.
Utilizando un framework de aspectos, se puede definir por un lado el código correspondiente a la funcionalidad a agregar (p.ej. registrar el timestamp de cuando se entró y de cuando se salió), y por otro cuál es el alcance que se le quiere dar a esta funcionalidad (p.ej. a todos los métodos de una lista de clases). Llamamos ***aspecto*** a la unión de una definición de acción y otra de alcance.

Parte 1 - join-points y point-cuts

Modelar la definición de join-points y point-cuts.
Un ***join-point*** es una condición sobre eventos a nivel de metamodelo, que define el alcance de un aspecto. Los eventos más interesantes son envío de un mensaje y asignación de una variable. En este TP vamos a cubrir solamente los envíos de mensaje. Por lo tanto, un join-point será un predicado (o sea, una condición) que aplica a un envío de mensaje.
Para este TP, todas las condiciones estarán referidas al método que se ejecuta al enviar un mensaje, o sea, van a ser condiciones sobre los métodos.
Un ejemplo de join-point es “todos los métodos definidos en la clase Punto”.

La definición de join-point tiene que soportar estas variantes

A. Restringir a métodos definidos en una clase, o un conjunto de clases.
B. Restringir a métodos definidos en una clase o cualquiera de sus descendientes.
C. Restringir a métodos de un nombre indicado, o cuyo nombre esté en un conjunto.
D. Restringir a métodos cuyo nombre empiece de cierta forma, p.ej. todos los métodos que empiecen con “set”.

E. Restringir a métodos con una determinada aridad, p.ej. “todos los métodos que tengan un parámetro”.

Un ***point-cut*** es el resultado de combinar join-points, usando los operadores “Y”, “O” y “NO”. P.ej. un point-cut posible es “todos los métodos definidos en Punto o en Rectangulo, cuyo selector empiece con ‘set’, y que lleven un parámetro”.

Estos objetos tienen que entender los siguientes mensajes

- *affected_classes*, devuelve el conjunto de clases para las cuales el alcance del join-point o point-cut incluye al menos uno de los métodos que define la clase.
- *affected_methods*, devuelve un conjunto de pares <clase,selector> para los métodos en el alcance del join-point o point-cut.
- *affects_method?(clase, selector)*, que indica si un método está incluido o no en el alcance.del join-point o point-cut. P.ej. si defino *pointCutPunto* como “todos los métodos de Punto” y *pointCutEs* como “todos los métodos cuyo selector empieza con ‘es’ “, entonces
    - pointCutPunto.affects_method?(Punto, :es_el_origen?) tiene que devolver true.
    - pointCutEs.affects_method?(Punto, :es_el_origen?) tiene que devolver true
    - pointCutPunto.affects_method?(Punto, :coordenada_mas_grande) tiene que devolver true
    - pointCutEs.affects_method?(Punto, :coordenada_mas_grande) tiene que devolver false
    - pointCutPunto.affects_method?(Mesa, :es_cuadrada?) tiene que devolver false
    - pointCutEs.affects_method?(Mesa, :es_cuadrada?) tiene que devolver true

Parte 2 - advices

Un advice es la especificación de código que se desea que se ejecute cuando se envía un mensaje que un aspecto decide interceptar. El código se le puede pasar al advice como una instancia de Proc, que encapsula un bloque.
Este bloque debe esperar los siguientes parámetros

- el primero es un array horrible, en el que vienen, en orden
    - el receptor del mensaje atrapado por el advice.
    - la clase del método atrapado.
    - el selector del método atrapado
- del segundo parámetro en adelante, van los argumentos recibidos por el método interceptado.

Respecto de la aridad, vale interceptar solamente métodos sin parámetros o con un parámetro fijo.
Además del código, hay que indicar cuándo debe ejecutarse. Incluir estas dos opciones:

A. antes de la ejecución del método original.

B. después de la ejecución del método original.

Modelar advices, los objetos que se definan tienen que entender este mensaje:

- apply_to(clase, selector): tiene que aplicar el advice al método indicado. O sea, si después del apply_to se envía un mensaje que bindea con este método, entonces debe evaluarse el código del advice, antes o después del método original según lo que indique el advice. Validar que la clase tenga un método definido para el selector que se indica, y que sea de la misma aridad del advice (que es la aridad del bloque que se ejecuta - 1).

OJO que el mismo advice puede ser aplicado a muchos métodos, eso va a pasar en el punto 3.

Trato de contarlo con un ejemplo. Ponele que tenemos este método

```
class Punto
  def initialize(x,y)
    @x = x
    @y = y
  end

  def shift_left(how_long)
    @x = @x + how_long
  end
end
```

Para definir un advice que se pueda aplicar a este método, el bloque del advice debe recibir dos parámetros. Esta sería una posible definición de un advice.

```
interceptor = Advice.new.
interceptor.code(Proc.new { | data_intercepcion param_original | … hacer algo … })
interceptor.apply_before   # esto dice que se aplique antes del método interceptado
```

Después de hacer

```
    interceptor.apply_to(Punto, :shift_left)
```

todas las llamadas al método shift_left de Punto, p.ej.\

```
   puntito = Punto.new(5,3)
   puntito.shift_left(4)
```

deben ser interceptadas por el advice, que se tiene que ejecutar antes del código de shift_left. Debe ejecutarse el bloque asociado al advice. En este ejemplo, en el primer parámetro del bloque (el que llamamos data_intercepcion) va a llegar el Array [puntito, Punto, :shift_left], o sea, [receptor del mensaje interceptado, clase del método, selector]. En el segundo parámetro del bloque va a llegar un 4, que es el parámetro del mensaje original.

Parte 3 - modelo de un aspecto

A partir de lo construido en las dos partes anteriores, modelar un aspecto, que incluye un point-cut, y uno o dos advices. Si un aspecto tiene dos advices, tienen que ser uno para ejecutar antes del método interceptado, y otro para ejecutar después.
Los objetos que implementan aspectos tienen que entender el mensaje apply, que aplica el o los advices a todos los métodos en el alcance del point-cut.

El test más fácil es un aspecto que muestre por pantalla “entrando a método tal de clase tal” y “saliendo de método tal de clase tal”.
Como otro test fácil, pueden hacer un aspecto que acumule (ponele, en una variable de clase de una clase que creen ustedes) cuántas veces se invocó cada método interceptado.

Bonus - aspectos reaplicables y quitables

Lograr que

- si le digo apply a un aspecto que ya se había aplicado, que lo aplique a los métodos donde no se había aplicado antes. Esto permite que si se agregan nuevas clases o métodos que estén en el alcance de un aspecto, dándole apply de nuevo, cubra también lo que se haya agregado después de haberle dado apply antes.
- si le digo unapply a un aspecto se “desaplique”, o sea, que deje de interceptar los métodos que había interceptado.

OJO esto **es difícil**. En particular, al aplicar un aspecto, se tiene que acordar de dónde se aplicó, guardando la información necesaria para desaplicarse.