Contratos

TADP - 2C 2020 - Trabajo Práctico Grupal - Metaprogramación

Descripción del dominio

Conceptualmente, un contrato es un acuerdo entre dos partes, cuando una de ellas (la proveedora) realiza una tarea para la otra (la cliente). Cada una de las partes espera beneficios a partir del contrato, y a cambio acepta un conjunto de obligaciones. Usualmente, lo que una de las partes ve como obligaciones son beneficios para la otra.

Se puede mostrar un ejemplo a partir de una tabla como la siguiente, que ilustra el contrato entre una aerolínea y un cliente:

	Obligaciones	**Beneficios**
Cliente	*(debe satisfacer las precondiciones)* Estar en Aeroparque al menos 1 hora antes de la salida del vuelo. Tener sólo equipaje permitido. Haber pagado el pasaje.	*(puede beneficiarse por las postcondiciones)* Llegar a Tucumán en 2 horas.
Proveedor (Aerolínea)	*(debe satisfacer las postcondiciones)* Llevar al cliente a Tucumán en 2 horas.	*(puede asumir las precondiciones)* No necesita llevar clientes que lleguen tarde, que tengan equipaje no permitido o que no hayan pagado el pasaje.

Estas mismas ideas pueden aplicarse al diseño con objetos, si consideramos que cuando enviamos un mensaje a un objeto estamos solicitando que realice algo por nosotros. Es decir, se establece una relación en la cual nosotros jugamos el rol de cliente, y el objeto el de proveedor.

El objetivo del trabajo práctico es entonces aplicar los conceptos vistos en clase para construir un framework en Ruby que provea las herramientas necesarias para soportar la definición declarativa de restricciones similares a contratos.

Antes y Después

Para facilitar la implementación de los puntos que siguen, se propone implementar una interfaz que permita definir comportamiento para ejecutar antes y después de recibir cada mensaje:

```
class MiClase

  before_and_after_each_call(
```

```
        # Bloque Before. Se ejecuta antes de cada mensaje
        proc{ puts “Entré a un mensaje” },
        # Bloque After. Se ejecuta después de cada mensaje
        proc{ puts “Salí de un mensaje” }
      )

      def mensaje_1
        puts “mensaje_1”
        return 5
      end

      def mensaje_2
        puts “mensaje_2”
        return 3
      end

    end

    MiClase.new.mensaje_2
    # Retorna 3 e imprime:
    # Entré a un mensaje
    # mensaje_2
    # Salí de un mensaje
```

La operación a definir debe recibir dos procs, uno para ejecutar antes de cada mensaje que se reciba y otro para ejecutar después. Tienen la libertad de elegir (y cambiar a gusto) qué parámetros pasarle a los bloques y en qué contexto ejecutarlos, considerando que:

- Los bloques deben ejecutarse incluso en métodos definidos a posteriori, por ejemplo, abriendo la clase una vez cerrada.

- Debe ser posible definir más de un before_and_after_each_call para cada clase. En caso de haber muchos, se deben ejecutar en orden.

Pista: El mensaje *method_added* podría ser de utilidad para este punto (aunque no es la única forma!)

Invariantes

Una *invariante* es una aserción que describe una propiedad que se cumple para todas las instancias de una clase en todo momento. Visto desde un punto de vista conceptual, una invariante es una cláusula general que aplica a un conjunto de contratos en un dominio determinado.

Las invariantes de una clase deben poder definirse utilizando la sintaxis presente en el siguiente ejemplo:

```
class Guerrero

  attr_accessor :vida, : fuerza

  invariant { vida >= 0 }
  invariant { fuerza > 0 && fuerza < 100 }


  def atacar(otro)
    otro.vida -= fuerza
  end

end
```

En el caso ejemplificado, la vida de los guerreros debe mantenerse siempre igual o por encima de 0, y la fuerza entre 1 y 100, de lo contrario podemos asumir que nos encontramos ante un guerrero con estado inválido.

Como el estado de un objeto sólo debe ser modificado a partir del envío de mensajes, sólo es necesario chequear las invariantes luego de la creación del objeto y luego de la ejecución de cualquier método de instancia (**coff** **coff** punto anterior **coff** **coff**). Si en algunos de estos momentos la condición de la invariante no se cumple se debe lanzar una excepción que detenga el programa.

Precondiciones y postcondiciones

Llamamos *precondiciones* a las aserciones que esperamos que se cumplan al momento de recibir un mensaje, y *postcondiciones* a las aserciones que el emisor del mensaje espera que sean verdaderas luego de que el mensaje finaliza.
Las precondiciones y postcondiciones con respecto a un método pueden definirse de la siguiente manera:

```
class Operaciones
  #precondición de dividir
  pre { divisor != 0 }
  #postcondición de dividir
  post { |result| result * divisor == dividendo }
  def dividir(dividendo, divisor)
    dividendo / divisor
  end
```

```
    # este método no se ve afectado por ninguna pre/post condición
    def restar(minuendo, sustraendo)
      minuendo - sustraendo
    end

  end
```

Como se ve en el ejemplo, cada llamado a pre/post afecta solamente al método definido inmediatamente debajo. Cada método sólo debe contemplar una única precondición y/o postcondición. En las postcondiciones, `result` representa el resultado de la ejecución del método.

Si al momento de ejecutar el método no se cumplen las precondiciones, o no se cumplen las postcondiciones luego de ejecutarlo, se debe lanzar una excepción. Por ejemplo:

```
> Operaciones.new.dividir(4, 2)
=> 2

> Operaciones.new.dividir(4, 0)
RuntimeError: Failed to meet preconditions
```

Ejemplo

A continuación presentamos un ejemplo de una clase completa con contratos.

```
class Pila
  attr_accessor :current_node, :capacity

  invariant { capacity >= 0 }

  post { empty? }
  def initialize(capacity)
    @capacity = capacity
    @current_node = nil
  end

  pre { !full? }
  post { height > 0 }
  def push(element)
    @current_node = Node.new(element, current_node)
  end

  pre { !empty? }
```

```ruby
    def pop
      element = top
      @current_node = @current_node.next_node
      element
    end

    pre { !empty? }
    def top
      current_node.element
    end

    def height
      empty? ? 0 : current_node.size
    end

    def empty?
      current_node.nil?
    end

    def full?
      height == capacity
    end

    Node = Struct.new(:element, :next_node) do
      def size
        next_node.nil? ? 1 : 1 + next_node.size
      end
    end
  end
```