Metaprogramación

Como siempre el TP debe tener una buena cobertura de tests, y estar ordenado, prolijo, sin código comentado, etc.
Y utilizando las buenas prácticas de programación OOP que vieron en materias anteriores.

A diferencia del TP 1 de mixins, esta vez vamos a hacer una seria de “mini ejercicios” más chiquitos, ya que resulta difícil aplicar todas las ideas de metaprogramación en un único dominio, sin caer en otros problemas arquitectónicos o que no son interesantes.

Parte 1: Transacciones Simples

Lo primero que vamos a querer es un mecanismo, inicialmente algo “manual” de transacciones sobre objetos. La idea de “transacción” es poder ejecutar cierto código en forma atómica (“se dice atómico”). De modo de que si algo falla durante la ejecución vuelva al “estado inicial”.

En objetos los estados serían las “variables de instancia” de nuestros objetos (para limitar un poco no consideramos variables de clase, ni tampoco efectos producidos por lecturas de archivos o cosas así).

Entonces “volver atrás” sería algo así como restaurar el estado de los objetos, sus variables de instancia, al último punto conocido.

Veamos un par de ejemplos.
Primero partimos de este objeto de dominio

```
class Persona
   attr_accessor :nombre, :edad
   def cumplirAnios
      self.edad = self.edad + 1     // noten que usamos el setter
   end
end
```

Ahora los casos

Transacción exitosa

Si va todo bien (no hay excepciones) entonces los objetos se ven afectados normalmente.
Ejemplo hacemos una función para hacer cumplir años al “tipo”, usando transacciones.

```
def queCumpla(p)
    Transactor.perform(p) { |p|
       p.cumplirAnios()
    }
end
```

La parte en azul es la interesante. Básicamente envolvemos nuestro código que queremos que sea transaccional en un bloque modelado como una transacción (“Transactor” sería una clase de uds).

Para limitar este primer punto, hace falta pasarle por parámetro cuales son los objetos “que nos interesa monitorear” para la transaccionalidad.
Y no los usamos directamente, sino que él nos los pasa por parámetro al bloque.
Suena medio inútil, pero eso permite al Transactor hacer magias. Podría pasarnos el mismo objeto modificado (?) u otro igualito, pero no… o lo que sea.

Luego un ejemplo de uso “exitosos” sería

```
p = Persona.new
p.edad = 22

queCumpla(p)
expect(p.edad).to eq 23
```

Como vemos la idea es que el tipo cumplió años bien.

Rollback automático ante un error

Ahora tenemos que soportar el caso en que, si algo falla (se produce una excepción) entonces, se deshagan los cambios que se hubieran hecho !

Ej

```
def queExploteAlCumplirAnios(p)
    Transactor.perform(p) { |p|
       p.cumplirAnios()
       raise ‘Kabooom!’   // forzamos a que explote luego de cumplir
!
    }
end
```

Luego, el test para esto sería

```
p = Persona.new
p.edad = 22

expect {
     queExploteAlCumplirAnios(p)
}.to raise_error(‘Kaboom’)        // la exception igual se propaga !

expect(p.edad).to eq 22      // pero el pobre tipo volvió a tener 22
```

Rollback manual (reificando la transacción)

Perfecto, ya tenemos un mecanismo automático. Ahora, poder modelar las transacciones como objetos y tener trazabilidad de las mismas nos permite empezar a jugar con ideas más locas.
Por ejemplo, dada una transacción que se ejecutó exitosamente, queremos poder rollbackearla manualmente, lo que sería como un “undo” / “deshacer”

Ej

```
    p = Persona.new
    p.edad = 22
    transaccion = Transactor.perform(p) { |p|
       p.cumplirAnios()
    }
    expect(p.edad).to eq 23     // anduvo !

    transaccion.undo()
    expect(p.edad).to eq 22     // la deshizo !
```

Incluso podríamos reaplicarla con **redo()**, luego del **undo()**

```
transaccion.redo()
expect(p.edad).to eq 23     // la re-hizo
```

Finalmente queremos poder pedirle la lista de cambios que produjo

```
cambios = transaccion.changes()
```

Donde es una lista de objetos representan cada cambio realizado durante la transacción.
Ej:

Objeto (object_id)	variable de instancia	valor anterior	valor nuevo
70121012831100	edad	22	23
...	..	...	...

Nota: para identificar a un objeto en particular pueden usar el método “object_id” que retorna un número único por instancia.

Parte 2: Transacciones “de verdad”

Ahora que ya entramos en calor con metaprogramación en el punto 1, queremos repensar el ejercicio de modo de que

1. no necesite explicitar los objetos que vamos a modificar,
2. no nos reemplace los objetos originales, sino que podamos usar los objetos a través de las referencias originales que ya teníamos

Acá una propuesta/ejemplo de uso
(conviene que arranca una 2da solución o bien hagan un tag si van a trabajar modificando lo que hicieron en el punto 1. de modo de luego poder corregirlo. Suponemos que lo mejor es arrancar de cero)

```
class Persona
   transactional

   attr_accessor :nombre, :edad
   def cumplirAnios
      self.edad = self.edad + 1
   end
end
```

Esto debería ser suficiente para luego poder trabajar con una transacción sin tener que pasar los objetos “transaccionales” por parámetro.
Qué es “transaccional” ?
Primero que nada es una “sugerencia” o más bien un ejemplo de como se podría hacer, pero no es la única forma.
Ahora sí, tampoco les vamos a decir cómo se implementa eso. Simplemente recordarles con otras preguntas: qué otras “cosas” podemos decir ahí en el cuerpo de una clase ? qué dijimos que era “attr_accessor” ?? :)

Ejemplo actualizado

```
p = Persona.new
p.edad = 22
transaccion = Transactor.perform() {  # no pasamos parámetros
                            # el bloque no recibe p
   p.cumplirAnios()     # dentro usamos los obj q ya teníamos
}
expect(p.edad).to eq 23
```

Incluso ahora a un objeto Persona podemos mandarle algunos mensajes “extra” que tal vez sirva como “ayuda” para entender qué hace ese “transaccional” con la clase.

Get Changes

También le podemos preguntar los cambios que lleva “hasta ahora” en la transacción.

```
p = Persona.new
p.edad = 22
Transactor.perform() {
      p.cumplirAnios()
      cambios = p.changes    // 1 cambio “edad” 22 => 23

      p.cumplirAnios()
      cambios = p.changes    // 2 cambios “edad” 22 => 23, “edad” 23
=> 24
}
```

O sea que internamente cada objeto debe registrar los cambios que va sufriendo.

Un dato importante acá: en ruby no hay una forma de interceptar el acceso a variables de instancia (quizás usando algún fwk de AOP?) Con lo cual vamos a limitar nuestra solución a que la clases tengan que modificar las variables de instancia usando el "setter", que, como es un método es posible interceptarlo, cambiarlo, reemplazarlo, etc.

Get Current Transaction

Dada una instancia "transaccional" le podemos preguntar la transacción actual y a ese objeto también los cambios (que pueden ser a más de un objeto)

```
p1 = Persona.new
p1.edad = 22

p2 = Persona.new
p2.edad = 10

Transactor.perform() {
    transaction = p.currentTransaction   // aca existe
    p1.cumplirAnios()
    p2.cumplirAnios()

    transaction.changes  # 2 cambios [p1, edad, 22, 23], [p2, edad,
10, 11]
}

transaction = p.currentTransaction  // acá no existe !
```

Parte 3: Lockeos

Otra funcionalidad interesante es poder "lockear" objetos.
Lockear significa que a partir de ese momento, el objeto no será modificable. Es decir que cuando alguien intente enviar un mensaje que intente modificar el estado se va a lanzar un error

Tenemos 2 variantes de lockeo

- lockear **un objeto en particular**
- lockear **todos los objetos**.

Lockear 1 objeto

Ejemplo

```
p = Persona.new
p.edad = 10

Transactor.perform() {
    p.lock()
    p.cumplirAnios()   // explota con ObjectLockedError (suya)
}
p1.cumplirAnios()  // no explota
```

El lockeo se mantiene durante la ejecución de la transacción, luego de la misma (ejecución del bloque) el objeto se deslockea automáticamente

Lockear todos los objetos

Una variante del caso anterior es poder lockear todos los objetos

```
p = Persona.new
p.edad = 10

Transactor.perform() {
    p.transaction.lock()
    p.cumplirAnios()   // explota con ObjectLockedError (suya)
}
p1.cumplirAnios()  // no explota
```

Deslockear (manualmente)

Así como podemos lockear, podemos deslockear tanto un objeto como la transacción.
Esto sólo se puede hacer durante la transacción (O sea.. una vez que terminó la transacción no se puede lockear ni deslockear nada, porque ya la transacción no tiene control sobre los objetos).

Ejemplo:

```
Transactor.perform() {
    p.lock()
    // blah blha.. operaciones que no debería cambiar a p
    p.unlock()
    p.cumplirAnios()   // no explota
}
```

Parte 4: Transacciones “de verdad 2”: multithreading/aislamiento

“Transacciones” se refiere a poder mutar estado en forma completa o no mutarlos directamente (atomicidad). Y de ahí se desprende también la idea de asilamiento/“isolation” si recuerdan el viejo principio ACID . Esto quiere decir que mientras 2 o más usuarios estan usando el sistema, sus respectivas mutaciones/transacciones no deben interferir entre sí.

Entonces tenemos que soportar eso.
En ruby (y en otros lenguajes), múltiples usuarios significa “multithreading”. Si cada thread tiene que trabajar en forma aislada, es decir tener su “scope de estado”, independientemente de los otros thread, entonces suena natural pensar que entonces ese estado tenga que estar en el thread.

Entonces para eso les tiramos un par de puntas, porque es un tema un poco “tecnológico”..

```
# retorna el thread actual. El mismo código ejecutándose en
distintos threads va a retornar distintos objetos threads
Thread.current
=> #<Thread:0x007fb6440bc410 run>
```

Ahí vemos que es un objeto de tipo Thread. Así que uno podría inspeccionar qué mensajes entiende.

```
Thread.current.class.instance_methods(false)
 => [:pending_interrupt?, :raise, :join, :value, :kill, :terminate,
:exit, :run, :wakeup, :[], :[]=, :key?, :keys, :priority,
:priority=, :status, :thread_variable_get, :thread_variable_set,
:thread_variables, :thread_variable?, :alive?, :stop?,
:abort_on_exception, :abort_on_exception=, :safe_level, :group,
:backtrace, :backtrace_locations, :inspect, :set_trace_func,
:add_trace_func]
```

Ahí vemos que Thread tiene métodos similares a un Hash, para indexarlo.
Bueno básicamente esa es la forma de guarda estado en un thread.
Se puede pensar como una variable “global” un poco mágica ya que la VM se encarga de que este objeto Thread sea el apropiado para cada ejecución.

Ejemplo para guardar un dato en este “thread local” (más info acá)

```
Thread.current[:pregunta] = “Donde está Santiago Maldonado?”
```

Luego lo accedemos con

```
Thread.current[:pregunta]
 => "Donde está Santiago Maldonado?"
```

Entonces de esta forma deben modificar su implementación de transacciones de modo de que el estado (los cambios) que se van realizando dentro/durante una transacción no se realicen directamente sobre las variables de instancia del objeto (porque entonces serían visibles para todos los demás), sino que se “aislen” en el thread actual.
Y sólo apliquen al objeto cuando la transacción se commitea.

Ejemplo como ejecutar un bloque de código en un nuevo thread

```
Thread.new {
   puts ‘thread ejecutando’
   sleep 6     # duermte 6 segundos
   puts ‘thread despertó y ya termina’
}
```

Y un test concreto del aislamiento de transacciones

```
p = Persona.new
p.edad = 22
Thread.new {
     Transactor.perform() {
      p.edad = 44
      sleep 3
  }
}

Thread.new {
     Transactor.perform() {
     p.edad = 55
      sleep 4

      p.cumplirAnios()
  }
}

sleep 2
expect(p.edad).to eq (22)    # no se ven las modificaciones hechas
en Thread1 ni en Thread 3

sleep 5      # los 2 threads deberían haber terminado
expect(p.edad).to eq (56)    # thread 2 ganó y pisó los cambios de
thread 1
```