Técnicas Avanzadas de Programación – UTN – FRBA
2do cuatrimestre 2015
Ejercicio Integrador

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.
- Este ejercicio se desarrollará los días 12 y 19 de Septiembre en clase. El objetivo es poder desarrollar un ejercicio al nivel de dificultad de un trabajo práctico en clase aplicando los conceptos aprendidos durante el cuatrimestre sobre metaprogramación.

2. Introducción

En Ruby existen, por defecto, dos formas de definir y compartir comportamiento entre los distintos objetos de nuestro sistema: Clases y Mixins (modules). Pero debido a sus características dinámicas y a su metamodelo maleable, es posible alterar las formas de recibir comportamiento e incluso agregar nuevas.

El objetivo de este trabajo práctico es, entonces, implementar y analizar una nueva forma de definir y compartir comportamiento en Ruby, puntualmente a través de **prototipos**.

3. Comportamiento esperado

Los requerimientos de cómo debe comportarse el nuevo sistema serán proporcionados en forma de código. La intención es que lo que se implemente pueda funcionar, pero no se busca acotar las posibles implementaciones. Luego, cada grupo deberá analizar en detalle las características particulares de su implementación, mostrando cómo se comporta en casos de conflictos y en relación con los elementos del metamodelo existentes.

El comportamiento esperado es el requerimiento definido, y debe estar testeado con pruebas unitarias automáticas, pero **no debe probarse sólo eso**.

4. Partes

El trabajo práctico está dividido en partes. Las partes están pensadas para guiarlos en el desarrollo, teniendo en cuenta la planificación de la materia. También cada parte va elaborando sobre la anterior, por lo que es recomendable ir resolviendo los puntos en el orden propuesto. Si bien lo más importante es el resultado final al momento de la entrega, **es preciso ir elaborando el TP a medida que es posible, para que no se junte mucha carga cerca de la fecha de entrega**.

4.1. Prototipos programáticos

Para esta primera parte, se busca introducir la nueva forma de definir comportamiento, pero sin azúcar sintáctico. La intención es poder hacer algo como lo que sigue:

```
guerrero = PrototypedObject.new

guerrero.set_property(:energia, 100)
expect(guerrero.energia).to eq(100)

guerrero.set_property(:potencial_defensivo, 10)
guerrero.set_property(:potencial_ofensivo, 30)

guerrero.set_method(:atacar_a,
  proc {
  |otro_guerrero|
  if(otro_guerrero.potencial_defensivo < self.potencial_ofensivo)
    otro_guerrero.recibe_danio(self.potencial_ofensivo - otro_guerrero.potencial_defensivo)
  end
  });

guerrero.set_method(:recibe_danio, proc {...})

otro_guerrero = guerrero.clone #clone es un metodo que ya viene definido en Ruby

guerrero.atacar_a otro_guerrero

expect(otro_guerrero.energia).to eq(80)
```

Hasta aquí el único agregado es la posibilidad de definir métodos y propiedades en cualquier objeto prototipado, sin necesidad que ese comportamiento provenga de una clase en particular.

Lo interesante viene ahora:

```
espadachin = PrototypedObject.new

espadachin.set_prototype(guerrero)

espadachin.set_property(:habilidad, 0.5)
espadachin.set_property(:potencial_espada, 30)

espadachin.energia = 100

{...} #mas inicializaciones
#deberia llamar a super, pero eso lo resolvemos mas adelante
espadachin.set_method(:potencial_ofensivo, proc {
  @potencial_ofensivo + self.potencial_espada * self.habilidad
})

espadachin.atacar_a(otro_guerrero)
expect(otro_guerrero.energia).to eq(75)
```

Más interesante todavía es la relación que se crea entre el espadachin y su prototipo:

```
guerrero.set_method(:sanar, proc {
  self.energia = self.energia + 10
})

espadachin.sanar
expect(espadachin.energia).to eq(110)
```

Es decir, cualquier cambio en el prototipo impacta también en los objetos derivados de él. Cabe aclarar que el prototipo provee métodos y no estado.

Distinto sucede con la clonación: Los cambios no se impactan en el objeto que es clon:

```
expect {otro_guerrero.sanar}.to raise_error(NoMethodError)
```

Además, con la clonación se copia el estado del objeto, cosa que no debe suceder con los prototipos.

Tampoco son afectados los métodos que fueron redefinidos por el objeto derivado:

```
guerrero.set_method(:potencial_ofensivo, proc {
  1000
})

expect(espadachin.potencial_ofensivo).to eq(45)
```

Finalmente, también nos interesa hacer que el comportamiento "prototipable" sea algo que yo pueda usar en cualquier lado. Por ejemplo:

```
class Object
  include Prototyped
end
```

4.2. Constructores

Las clases proveen, además de comportamiento, un mecanismo para construir los objetos de manera tal que cada uno pueda tener inicializado su estado para funcionar según corresponda.

En principio, queremos un objeto o función que nos permita construir un objeto en base a un prototipo, inicializando su estado con parámetros:

```
Guerrero = PrototypedConstructor.new(guerrero, proc {
 |guerrero_nuevo, una_energia, un_potencial_ofensivo, un_potencial_defensivo|
  guerrero_nuevo.energia = una_energia
  guerrero_nuevo.potencial_ofensivo = un_potencial_ofensivo
  guerrero_nuevo.potencial_defensivo = un_potencial_defensivo
})

un_guerrero = Guerrero.new(100, 30, 10)
expect(un_guerrero.energia).to eq(100)
```

El resultado del new sería un objeto con "guerrero" como prototipo y el estado inicializado como indica el proc que se pasa como parámetro.

Pero esa forma de definir los constructores requiere mucho trabajo. Luego veremos cómo hacerla más linda y usable, pero por lo pronto lo que podemos hacer es definir un constructor sencillo por defecto. Tenemos la información de las propiedades del prototipo para trabajar. Por ejemplo:

```
Guerrero = PrototypedConstructor.new(guerrero)

un_guerrero = Guerrero.new(
  {energia: 100, potencial_ofensivo: 30, potencial_defensivo: 10}
)
expect(un_guerrero.potencial_ofensivo).to eq(30)
```

Por el hecho de que no está claramente definido el orden de los parámetros, usamos un mapa.

Además, queremos definir un constructor que cree un prototipo y copie el estado actual del prototipo. Para ello, usamos lo siguiente:

```
Guerrero = PrototypedConstructor.copy(guerrero)

un_guerrero = Guerrero.new
expect(un_guerrero.potencial_defensivo).to eq(10)
```

Finalmente, con lo que hicimos también podemos producir un constructor que altere los métodos que entiende el objeto, produciendo un nuevo tipo de objeto. Nos interesa que un constructor pueda extender de otro:

```
#Guerrero es la primer variante de constructores, que recibe 3 parametros
Espadachin = Guerrero.extended {
 |espadachin, habilidad, potencial_espada|
  espadachin.set_property(:habilidad, habilidad)
  espadachin.set_property(:potencial_espada, potencial_espada)

  espadachin.set_method(:potencial_ofensivo, proc {
    @potencial_ofensivo + self.potencial_espada * self.habilidad
  })

})

espadachin = Espadachin.new(100, 30, 10, 0.5, 30)
expect(espadachin.potencial_ofensivo).to eq(45)
```

La convención que se define (de momento) es que los parámetros del constructor se pasan y se consumen en orden. Los primeros 3 parámetros del constructor generado van para Guerrero, porque Guerrero recibe 3 parámetros en su constructor. Si fuera la segunda variante del constructor de guerrero, el constructor de Espadachín debería llamarse así:

```
espadachin = Espadachin.new({energia: 100, potencial_ofensivo: 30, potencial_defensivo: 10},
    0.5, 30)
```

4.3. Azúcar sintáctico

La implementación de prototipos que hicimos funciona, pero tiene problemas de expresividad, y realmente no es muy usable, muy cómoda para el programador. La intención ahora es entonces agregar una interfaz a nuestros objetos prototipados para que sean más fácilmente utilizables, aprovechando las características de Ruby.

Comenzamos con los elementos de la primera parte:

```
guerrero_proto = PrototypedObject.new
guerrero_proto.energia = 100
expect(guerrero_proto.energia).to eq(100)

guerrero_proto.potencial_defensivo = 10
guerrero_proto.potencial_ofensivo = 30

guerrero_proto.atacar_a = proc { |otro_guerrero| ... });
guerrero_proto.recibe_danio = proc {...}

Guerrero = PrototypedConstructor.copy(guerrero_proto)

un_guerrero = Guerrero.new
Guerrero.new.atacar_a(un_guerrero)

expect(un_guerrero.energia).to eq(80)
```

La idea es que, dinámicamente, uno puede "asignar" propiedades y métodos a los objetos prototipados.

También es incómodo repetir el receptor del mensaje en todo momento. Podemos hacer algo mejor:

```
guerrero_proto = PrototypedObject.new {
  self.energia = 100
  self.potencial_ofensivo = 30
  self.potencial_defensivo = 10

  self.atacar_a = proc { |otro_guerrero| ... });
  self.recibe_danio = proc {...}
}
```

Podemos aplicar ideas similares para los constructores:

```
Guerrero = PrototypedConstructor.new(guerrero) do |una_energia, un_potencial_ofensivo,
    un_potencial_defensivo|
  self.energia = una_energia
  self.potencial_ofensivo = un_potencial_ofensivo
  self.potencial_defensivo = un_potencial_defensivo
end
```

También, puede ser práctico definir y utilizar el prototipo en el mismo momento:

```
Guerrero = PrototypedConstructor.create {
  self.atacar_a = proc { |otro_guerrero| ... });
  self.recibe_danio = proc {...}
  }.with {
  |una_energia, un_potencial_ofensivo, un_potencial_defensivo|
    self.energia = una_energia
    self.potencial_ofensivo = un_potencial_ofensivo
    self.potencial_defensivo = un_potencial_defensivo
}
```

Es necesario pasarle el procedimiento del constructor para inicializar las propiedades porque no tenemos información de las propiedades existentes. También se podría definir un método para crear un constructor por defecto más sencillo:

```
Guerrero = PrototypedConstructor.create {
  self.atacar_a = proc { |otro_guerrero| ... });
  self.recibe_danio = proc {...}
 }.with_properties([:energia, :potencial_ofensivo, :potencial_defensivo])
```

Si yo quisiera obtener el prototipo de un constructor de estos, debería poder pedírselo, y valen las mismas condiciones que para los prototipos con nombre:

```
atila = Guerrero.new(100, 50, 30)

expect(atila.potencial_ofensivo).to eq(50)

proto_guerrero = Guerrero.prototype
proto_guerrero.potencial_ofensivo = proc {
   1000
}

expect(atila.potencial_ofensivo).to eq(1000)
```

Finalmente, debería poder aprovechar esta nueva sintaxis también para la extensión de constructores:

```
Espadachin = Guerrero.extended {
  |una_habilidad, un_potencial_espada|
  self.habilidad = una_habilidad
  self.potencial_espada = un_potencial_espada

  self.potencial_ofensivo = proc {
    @potencial_ofensivo + self.potencial_espada * self.habilidad
  }
}
```

4.4. Llamar al prototipo anterior y Múltiples prototipos

Anteriormente, cuando definimos Espadachin nos encontramos con el inconveniente de que queríamos utilizar la implementación previa de potencial_ofensivo (que venía de guerrero), pero no teníamos forma de acceder a ella, porque estábamos pisando ese método.

Nos interesa entonces definir un mecanismo para poder llamar a la implementación siguiente de la cadena de prototipos:

```
Espadachin = Guerrero.extended {
  |una_habilidad, un_potencial_espada|
  self.habilidad = una_habilidad
  self.potencial_espada = un_potencial_espada

  self.potencial_ofensivo = proc {
    call_next + self.potencial_espada * self.habilidad
  }
}
```

Finalmente, nos interesa incorporar una idea similar a lo que hacemos con Mixins o con herencia múltiple y permitir múltiples prototipos en un mismo objeto:

```
Guerrero.prototype.set_prototypes([proto_atacante, proto_defensor])
```

Cabe aclarar que proto_atacante y proto_defensor no deben verse afectados en su cadena de prototipado: No es válido que proto_defensor pase a ser prototipo de proto_atacante ni vice versa.

Queda a criterio del grupo cómo se efectúa la resolución de conflictos en caso que existieran y cómo funciona el call_next al tener múltiples prototipos.