TADsPec

TADP - 2022 C2 - TP Metaprogramación

Descripción del dominio

El objetivo del trabajo práctico es aplicar los conceptos vistos en clase para construir un framework de testeo en Ruby que incorpore la funcionalidad básica para poder aplicarlo en un contexto laboral real y soporte extensiones de usabilidad basadas en convenciones e implementadas con metaprogramación.

Entrega Grupal

En esta entrega tenemos como objetivo desarrollar la lógica necesaria para implementar la funcionalidad que se describe a continuación. 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.
- Aprovechar las abstracciones provistas por el metamodelo de Ruby.
- Realizar un testeo integral de la aplicación cuidando también el diseño de los mismos.

1. Aserciones

Una parte fundamental de cualquier framework de testing es la posibilidad de describir que tiene que ocurrir para que el test falle o tenga éxito. Con este fin, vamos a definir un pequeño

DSL que soporte una sintaxis amigable para verificar las postcondiciones de nuestros tests. Para eso queremos que dentro de los tests (**Y SÓLO DENTRO DE LOS TESTS!**) todos los objetos entiendan un mensaje *debería* que permita describir una aserción. Este mensaje tiene que recibir un parámetro que represente la descripción de qué es lo que se espera que el objeto cumpla. Se espera que las aserciones puedan ser escritas con la siguiente sintaxis:

```
tadp.docentes[0].deberia ser nico


erwin.docente.deberia ser_igual true
erwin.edad.deberia ser_igual 18
erwin.edad.deberia ser menor_a 40
```

Nota: La sintaxis de Ruby es bastante permisiva a la hora de permitir o no omitir paréntesis. La manera de conseguir que nuestro framework soporte la sintaxis que pedimos al principio puede parecer confuso, pero tengan en cuenta que las siguientes dos líneas son exactamente iguales (si consiguen se soporte la primera, la segunda va a funcionar solita):

```
objeto.m1(self.m2(self.m3(parametro)))
objeto.m1 m2 m3 parametro #Ruby permite escribir lo de arriba, así
```

A continuación se describen las configuraciones mínimas que deben ser soportadas por el *debería*, pero es importante tener en cuenta que la implementación debe contemplar la posibilidad de agregar más en el futuro.

Ser o No Ser

Queremos poder crear aserciones que verifiquen que un objeto sea el esperado. Para eso esperamos disponer de una configuración *ser* para el *debería*.

```
objeto.deberia ser resultado_esperado
```

La configuración *ser* recibe por parámetro qué es lo que se espera que el objeto testeado sea.

```
7.deberia ser 7 # pasa
true.deberia ser false # falla, obvio
leandro.edad.deberia ser 25 #falla (lean tiene 22)
```

Además de comparar por igualdad, queremos que se puedan hacer verificaciones de rango:

```
leandro.edad.deberia ser mayor_a 20
# pasa si el objeto es mayor al parámetro

leandro.edad.deberia ser menor_a 25
# pasa si el objeto es menor al parámetro

leandro.edad.deberia ser uno_de_estos [7, 22, "hola"]
# pasa si el objeto está en la lista recibida por parámetro

leandro.edad.deberia ser uno_de_estos 7, 22, "hola"
# debe poder escribirse usando varargs en lugar de un array
```

Otra cosa que buscamos es disponer de un azúcar sintáctico para facilitar el chequeo de propiedades booleanas (que en Ruby se escriben, por convención, con un “?” al final del nombre). Para todos los mensajes que terminen con “?” debe existir una variante de la configuración *ser* que testee si el resultado de ese mensaje es true.

```
class Persona
  def viejo?
    @edad > 29
  end
end

nico.deberia ser_viejo    # pasa: Nico tiene edad 30.
nico.viejo?.deberia ser true # La linea de arriba es equivalente a esta

leandro.deberia ser_viejo # falla: Leandro tiene 22.
```

```
leandro.deberia ser_joven # explota: Leandro no entiende el mensaje joven?´’
```

Tener

Otra posible configuración para el *deberia* nos tiene que permitir verificar fácilmente los atributos de un objeto. Para eso queremos poder disponer de una serie de mensajes tener_<<nombre de atributo>> que controle el valor de dicho atributo para el objeto (ojo, no importa si el atributo tiene o no accesor, queremos chequear directo el estado interno).

```
objeto.deberia tener_<<nombre del atributo>> valor_esperado
```

Similar al *ser*, queremos poder checkear por igualdad o por rango:

```
class Persona
  def viejo?
    @edad > 29
  end
end

leandro.deberia tener_edad 22 # pasa
leandro.deberia tener_nombre "leandro" # falla: no hay atributo nombre
leandro.deberia tener_nombre nil # pasa
leandro.deberia tener_edad mayor_a 20 # pasa
leandro.deberia tener_edad menor_a 25 # pasa
leandro.deberia tener_edad uno_de_estos [7, 22, "hola"] # pasa
```

Nota: Asegurense de no repetir lógica…

Entender

En los lenguajes dinámicos cómo ruby los programas pueden modificar las interfaces de los objetos en tiempo de ejecución; es por eso que a veces resulta necesario testear si un objeto entiende o no cierto mensaje.

```
class Persona
  def viejo?
    @edad > 29
  end
end

leandro.deberia entender :viejo? # pasa
leandro.deberia entender :class  # pasa: este mensaje se hereda de Object
leandro.deberia entender :nombre # falla: leandro no entiende el mensaje
```

Obviamente, testear si un objeto entiende un mensaje **NO DEBERÍA EJECUTARLO**, solamente verificar si el objeto responde a dicho mensaje.

Nota: No se olviden que el objeto podría tener sobreescrito el method_missing. Contemplen ese caso en su implementación.

Explotar

El camino feliz no lo es todo. A veces los métodos tienen que fallar y necesitamos una forma de testear que lo hacen de la forma correcta. Con ese fin, en los casos en que el objeto testeado sea un bloque, queremos poder configurar el *deberia* para testear que al evaluarlo se lanza una excepción en particular. Para eso agregamos una configuración *explotar_con*.

```
en { 7 / 0 }.deberia explotar_con ZeroDivisionError # pasa
en { leandro.nombre }.deberia explotar_con NoMethodError # pasa
en { leandro.nombre }.deberia explotar_con Error # pasa: NoMethodError <
Error
en { leandro.viejo?}.deberia explotar_con NoMethodError # falla: No tira
error
en { 7 / 0 }.deberia explotar_con NoMethodError # falla: Tira otro error
```

2. Tests y Suites

Las aserciones no sirven de nada si no podemos definirles un contexto y hacer que se ejecuten. Queremos modelar para nuestro framework la idea de "test" como una unidad de trabajo mínima, que permita inicializar las condiciones a testear, realizar la operación foco del testeo y validar sus post-condiciones.

Para simplificar las cosas, vamos a llamar "*test*" a cualquier método cuyo nombre empiece con las palabras "`testear_que_`" y no espere parámetros.

```
# Esto es un test
def testear_que_las_personas_de_mas_de_29_son_viejas
  persona = Persona.new(30)
  persona.deberia ser_viejo
end

# Esto no
def las_personas_de_mas_de_29_son_viejas
  persona = Persona.new(30)
  persona.deberia ser_viejo
end
```

Por otro lado, llamamos "Test Suite" a un conjunto de tests relacionados por algún criterio (normalmente, aquello que están testeando). En la práctica, queremos representar los Suites con Clases de Ruby. Sólo vamos a considerar Suites a aquellas clases que definan (o incluyan algún módulo que las haga definir) al menos un test.

```
# Esto es un suite de tests
class MiSuiteDeTests
  def testear_que_pasa_algo
```

```
  end

  def otro_metodo_que_no_es_un_test
  end
end

# Esto no
class UnaClaseComun
  def metodo_que_no_es_un_test
  end
end
```

¿Porqué es importante esta estructura? ¡Porque queremos poder correr los tests! Desde la consola tiene que ser posible:

- Correr una suite de tests en particular
- Correr un test específico de una suite
- Correr todas las suites que se hayan importado al contexto

```
# Corre todos los tests de todas las suites en contexto
> TADsPec.testear

# Corre todos los tests de la suite MiSuite
> TADsPec.testear MiSuite

# Corre los tests :testear_que_una_cosa, :testear_que_otra_cosa y cualquier
otro test de la suite MiSuite cuyo nombre se pase por parámetro
> TADsPec.testear MiSuite, :una_cosa, :otra_cosa, etc...
```

Al correr un test existen 3 posibles resultados:

- **El test pasó**: Ocurre si el método que define al test se ejecuta sin problemas en su totalidad y todas las aserciones pasan.
- **El test falló:** Ocurre si el método que define al test se ejecuta sin problemas en su totalidad pero al menos una de las aserciones falla.

- **El test explotó:** Ocurre si el método que define al test lanza una excepción durante su ejecución.

Al ejecutar los tests por consola, todos los tests solicitados deben ejecutarse, independientemente de si alguno de ellos falla o explota y, al finalizar, se debe informar:

- La cantidad total de test corridos, pasados, fallados y explotados.
- El nombre de cada test que pasó con éxito.
- El nombre de cada test que falló, junto con una explicación de porqué (cada aserción fallida debe mostrarse de la forma más clara posible, explicando que se esperaba y que se obtuvo en cambio).
- El nombre de cada test que explotó, junto con la excepción lanzada y el stack que la produjo.

Nota: Si bien en la consola se espera ver un string con el resultado les recomendamos mucho que modelen la ejecución con objetos de alto nivel y no concatenando strings…

3. Mocking y Spying

Al testear es muy común querer analizar una porción de mi programa de forma aislada del resto, para simplificar la complejidad del contexto, acotar el problema y minimizar el set-up. Con este fin, muchos frameworks de testeo (especialmente aquellos implementados en lenguajes dinámicos) proveen herramientas que permiten mockear y/o espiar el código.

Entender en profundidad estos conceptos escapa a los objetivos de la materia. Alcanza con entender que ambas ideas giran en torno a reemplazar temporalmente la implementación de una parte del modelo de negocio con una más conveniente para testear durante la ejecución de los mismos.

Es super importante que estos cambios no sobrevivan la ejecución de los tests; es decir, todas las operaciones que describimos en esta sección no deben producir un efecto sobre el modelo de negocio que persista una vez finalizada la ejecución.

Mocks

Queremos poder reemplazar (de forma temporal) la implementación de un método de cualquier clase por un cacho de código que resulte más conveniente en el momento. Para eso, se pide implementar la lógica necesaria para que, dentro de los tests, las clases y módulos entiendan

un mensaje ”mockear” que reciba el nombre de un método y un bloque con el cuerpo que esperamos que dicho método tenga durante la ejecución del test.

```
class PersonaHome
  def todas_las_personas
    # Este método consume un servicio web que consulta una base de datos
  end

  def personas_viejas
    self.todas_las_personas.select{|p| p.viejo?}
  end
end

class PersonaHomeTests
  def testear_que_personas_viejas_trae_solo_a_los_viejos
    nico = Persona.new(30)
    axel = Persona.new(30)
    lean = Persona.new(22)

    # Mockeo el mensaje para no consumir el servicio y simplificar el test
    PersonaHome.mockear(:todas_las_personas) do
      [nico, axel, lean]
    end

    viejos = PersonaHome.personas_viejas

    viejos.deberia ser [nico, axel]
  end
end
```

El mockeo de un método debe estar scopeado al contexto en el que se lo definió. Esto quiere decir que, una vez finalizado el test, el método mockeado debe volver a su implementación original.

Spies

Similar al mockeo, espiar un objeto permite modificar su comportamiento durante los tests pero, en lugar de reemplazar la lógica que el método ejecuta, se mantiene la lógica original agregando además la capacidad de analizar a posteriori las interacciones con el objeto durante la ejecución del test.

Para obtener objetos espiables se debe desarrollar el mensaje *espiable*.

```
objeto_espiado = espiar(objeto_comun)
```

A partir de ahora nuestro objeto espiado debe poder responder a una interfaz extendida de *deberia* que acepte una configuración “haber_recibido” que nos permitan inspeccionar los mensajes que recibió durante el test.

En caso de que se intente usar la configuración “haber_recibido” con un objeto que no haya sido espiado previamente, la aserción debe fallar, explicando la situación.

```
class Persona
  attr_accessor :edad
  def viejo?
    self.edad > 29
  end
end

class PersonaTest

  def testear_que_se_use_la_edad
    lean = Persona.new(22)
    pato = Persona.new(23)
    pato = espiar(pato)

    pato.viejo?
```

```
pato.deberia haber_recibido(:edad)
# pasa: edad se llama durante la ejecución de viejo?

pato.deberia haber_recibido(:edad).veces(1)
# pasa: edad se recibió exactamente 1 vez.
pato.deberia haber_recibido(:edad).veces(5)
# falla: edad sólo se recibió una vez.

pato.deberia haber_recibido(:viejo?).con_argumentos(19, "hola")
# falla, recibió el mensaje, pero sin esos argumentos.

pato.deberia haber_recibido(:viejo?).con_argumentos()
# pasa, recibió el mensaje sin argumentos.

lean.viejo?
lean.deberia haber_recibido(:edad)
# falla: lean no fue espiado!
```

Al igual que pasa con los mocks, cualquier cambio a la lógica del objeto requerido para espiarlo debe desaparecer al finalizar el test.