Paradigma Orientado a Objetos

Módulo 11: Testeo unitario automatizado avanzado

por Lucas Spigariol
Fernando Dodino
y colaboración de
Matías Freyre
Versión 2.2
Septiembre 2018

Indice

1 Introducción

Hasta el momento hemos conocido el testeo unitario automatizado que tiene Wollok, que requiere definir **casos de prueba**, e implementarlos mediante **tests** que se agrupan en construcciones llamadas **describe**. La ejecución de los tests tiene tres resultados posibles: el test puede satisfacer las comprobaciones, no satisfacerlas o arrojar un error antes de poder determinar un resultado.

Continuaremos la explicación con conceptos que permitirán mejorar la calidad de nuestras pruebas automatizadas.

2 Código compartido entre tests

Cuando la complejidad de los casos de prueba crece, aparece la necesidad de compartir estado procurando mantener el carácter unitario de cada test. Si bien nunca hay dos tests idénticos, es frecuente que haya diferentes tests que guardan grandes coincidencias con otros, que necesiten realizar acciones en común, o que prueban diferentes aspectos sobre los mismos conjuntos de datos. Para lograr armonizar estas situaciones coherentemente con los criterios clásicos de la programación en cuanto a no repetir lógica, *Wollok* provee algunas herramientas adicionales para organizar los tests.

2.1 Ejemplo y solución

Para que se vea mejor la utilidad de las herramientas que vamos a presentar, reformulamos y ampliamos el ejemplo de pepita.

El planteo del problema ahora agrega:

> *"En vez de volar una cierta cantidad de kilómetros, vuela a diferentes ciudades y cada una de ellas implica un consumo de energía diferente. Pepita debe poder cumplir su deseo, que consiste en volar a la más reciente de sus ciudades favoritas. Una vez cumplido su deseo, esa ciudad deja de estar entre sus favoritas. También se quiere saber si una determinada ciudad es favorita para pepita"*

La solución ahora incluye la clase ciudad (muy trivial) y cambios en el objeto pepita:

```
class Ciudad {
	const consumo

	method consumo() {
		return consumo  // o consumo podría ser una propiedad...
	}
}

object pepita {
	var energia = 100
	const ciudadesFavoritas = [ ] // Nuevo

	method energia() { return energia }

	method esFuerte() { return energia > 50 }

	method volar(ciudad) { // Modificado para recibir una ciudad
		energia = energia - ( ciudad.consumo() + 10 )
	}

	method comer(gramos) {
		energia = energia + 4 * gramos
	}

	method cumplirDeseo() { // Nuevo
		const destino = ciudadesFavoritas.last()
		self.volar(destino)
		ciudadesFavoritas.remove(destino)
	}

	method agregarCiudad(ciudad) { // Nuevo
		ciudadesFavoritas.add(ciudad)
	}

	method esFavorita(ciudad) { // Nuevo
		return ciudadesFavoritas.contains(ciudad)
	}
}
```

Para probar la nueva versión, se mantienen algunos casos de prueba, se reformulan otros y se agregan nuevos:

Acciones	Verificación
ninguna	pepita tiene energía 100
ninguna	pepita es fuerte
que pepita vuele a San Martín (consume 5 unidades extra)	pepita queda con energía 85
que pepita coma 120 gramos	pepita queda con energía 580
que pepita coma 0 gramos	pepita queda con energía 100
que pepita vuele a Buenos Aires (consume 0 unidades extra)	pepita queda con energía 90
que pepita vuele a Campana (consume 60 unidades extra)	pepita no es fuerte

Planteamos un escenario con dos ciudades favoritas, Campana y San Martín. En algún momento pepita puede incorporar como nueva ciudad a Quilmes, que consume 1 unidad de energía.

Que pepita cumpla su deseo (volar a San Martín)	pepita queda con energía 85
Que pepita cumpla su deseo, luego conozca Quilmes y vuelva a cumplir su sueño dos veces más (volar a San Martín, luego a Quilmes por último a Campana)	pepita queda con energía 4

También puede ser conveniente verificar qué pasa con las ciudades favoritas de pepita.

Ninguna acción previa	pepita tiene a San Martín como ciudad favorita
Que pepita cumpla su deseo (volar a San Martín)	pepita no tiene a San Martín como ciudad favorita.
Que pepita cumpla su deseo (volar a San Martín)	pepita tiene a Campana como ciudad favorita.
Que pepita cumpla su deseo, luego conozca Quilmes y vuelva a cumplir su	pepita no tiene a Campana como ciudad favorita.

sueño dos veces más (volar a San Martín, luego a Quilmes por último a Campana)	

Probablemente podrían hacerse otras pruebas diferentes o plantearlas de otra manera, pero para lo que queremos mostrar es suficiente.

2.2 Inicialización del describe

A continuación de la especificación del **describe** con su correspondiente nombre, y previo a la enumeración de los tests que abarca, se pueden declarar variables y constantes. Al igual que cuando se definen en objetos o clases, pueden inicializarse ya sea con valores puntuales, instanciando clases existentes o con el valor de retorno de cualquier mensaje. El alcance de estas variables y constantes son todos los tests del describe (y también fixture y métodos). Se permite definir constantes sin un valor inicial, siempre que sean inicializadas en el fixture.

Importante: Tener en cuenta que el estado de estas variables (y constantes) se reinicia con el valor especificado previo a ejecutar cada uno de los tests, al igual que todos los objetos existentes en el ambiente.

2.3 Fixtures

Cuando se requiere realizar previo a cada test otras acciones de configuración de la situación inicial más complejas, para las cuales no es suficiente la inicialización de variables, se puede definir un fixture. Se lo hace precisamente con la palabra reservada **fixture**, sin agregar identificador y luego entre { } se detalla la secuencia de mensajes que se necesite, de manera similar a los constructores de las clases. Se especifica luego de la declaración e inicialización de variables y previo a los tests.

2.4 Métodos auxiliares

Para ciertas acciones que se repiten entre algunos tests pero no son comunes a todos, se pueden definir métodos auxiliares, de igual manera que los métodos de cualquier objeto. Su nombre es cualquier identificador válido, puede tener parámetros, retornar valores, y ser invocado desde cualquier lugar del describe, -un test, otro método, la sección de inicialización-, utilizando self.

2.5 Tests completos

Considerando todos los elementos mencionados, el orden de ejecución de cada uno de los tests del describe será:

- Reseteo del ambiente
- Inicialización de las variables de la descripción
- Fixture
- Test propiamente dicho

En el ejemplo de pepita, los combinamos de la siguiente manera:

```
import pepita.*

describe "Tests de Pepita" {
	// son objetos que se usan en diferentes test
	const campana = new Ciudad(consumo = 60)
	const bsas = new Ciudad(consumo = 0)
	const sanMartin = new Ciudad(consumo = 5)
	const quilmes = new Ciudad(consumo = 1)

	// todos los tests parten con pepita teniendo como
	// favoritas a estas dos ciudades
	fixture {
		pepita.agregarCiudad(campana)
		pepita.agregarCiudad(sanMartin)
	}

	// hay tests que prueban cosas diferentes a partir de la misma
	//situación, por lo que conviene delegarla en un método como éste
	method viajeLoco() {
		pepita.cumplirDeseo()
		pepita.agregarCiudad(quilmes)
		pepita.cumplirDeseo()
		pepita.cumplirDeseo()
	}

	test "pepita comienza con 100 unidades de energía" {
		assert.equals(100, pepita.energia()) // igual al anterior
	}

	test "pepita comienza siendo fuerte" {
		assert.that(pepita.esFuerte()) // igual al anterior
```

```
	}

	test "pepita vuela a una ciudad con consumo mayor a cero y ese
consumo le quita energia" {
		pepita.volar(sanMartin) // Modificado, ahora con una ciudad
		assert.equals(85, pepita.energia())
	}

	test "cuando pepita come su energia baja" {
		pepita.comer(120) // igual al anterior
		assert.equals(580, pepita.energia())
	}

	test "pepita vuela a una ciudad que no tiene consumo, la energía
baja un valor constante" {
		pepita.volar(bsas) // Modificado, ahora con una ciudad
		assert.equals(90, pepita.energia())
	}

	test "pepita come 0 gramos y la energía se mantiene" {
		pepita.comer(0)
		assert.equals(100, pepita.energia())
	}

	test "pepita vuela a una ciudad que le consume mucho, ya no es
fuerte" {
		pepita.volar(campana) // Modificado, ahora con una ciudad
		assert.notThat(pepita.esFuerte())
	}

	test "pepita cumple su deseo, al volar queda con menos energia" {
		pepita.cumplirDeseo() // Nuevo
		assert.equals(85, pepita.energia())
	}

	test "pepita hace su viaje loco y queda con energia 4" {
		self.viajeLoco() // Nuevo. Usa método del describe
		assert.equals(4, pepita.energia())
	}

	test "al inicio pepita tiene a San Martin por ciudad favorita" {
		assert.that(pepita.esFavorita(sanMartin)) // Nuevo
	}

	test "pepita cumple su deseo y ya no tiene a San Martin como
ciudad favorita" {
```

```
            pepita.cumplirDeseo() // Nuevo
            assert.notThat(pepita.esFavorita(sanMartin))
        }

        test "pepita cumple su deseo y tiene a Campana como ciudad
favorita" {
            pepita.cumplirDeseo() // Nuevo
            assert.that(pepita.esFavorita(campana))
        }

        test "pepita hace su viaje loco y no tiene a campana como
favorita" {
            self.viajeLoco() // Nuevo. Usa método del describe
            assert.notThat(pepita.esFavorita(campana))
        }
}
```

3 Testeo de errores

En el capítulo de manejo de errores, hemos presentado el ejercicio del monedero, donde podemos

- poner plata: un valor positivo
- sacar plata: un valor positivo sin que exceda lo que hay en el monedero

Dado un monedero que inicialmente tenga $ 500, ¿qué pruebas podemos hacer?

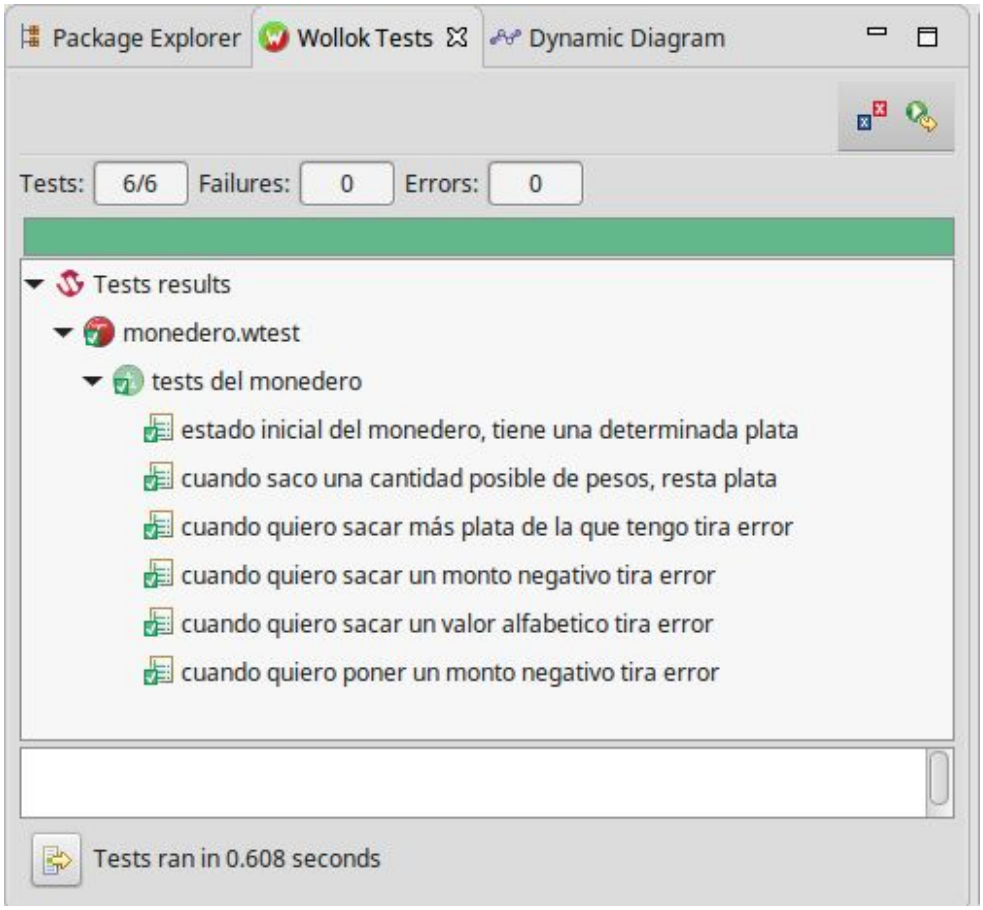
Precondiciones	Verificación
ninguna	El monedero tiene $ 500
sacamos $ 15 del monedero	Al monedero le quedan $ 485
quiero sacar $ 1000 del monedero (hay $ 500)	Tiene que dar error con mensaje “Debe retirar menos de 500”
sacar $ -20 del monedero	Tiene que dar error con mensaje “La cantidad debe ser positiva”
sacar “A” del monedero	Tiene que dar error
poner $ -20 en el monedero	Tiene que dar error con un mensaje “La cantidad a retirar debe ser positiva”

El lector podrá notar que en los últimos 4 casos lo que se espera es lo mismo: un error. Vemos cómo se implementa[1]:

```
import monedero.*

test "estado inicial del monedero tiene 500 $" {
	assert.equals(500, monedero.plata())
}

test "cuando saco una cantidad posible de pesos, resta plata" {
	monedero.sacar(15)
	assert.equals(485, monedero.plata())
}

test "cuando quiero sacar más plata de la que tengo tira error" {
	assert.throwsExceptionWithMessage("Debe retirar menos de 500",
	{ monedero.sacar(1000)})
}

test "cuando quiero sacar un monto negativo tira error" {
	assert.throwsExceptionLike(
		new UserException(message = "La cantidad a retirar debe ser
positiva"),
		{ monedero.sacar(-20) }
	)
}

test "cuando quiero sacar un valor alfabetico tira error" {
	assert.throwsException({ monedero.sacar("A") })
}

test "cuando quiero poner un monto negativo tira error" {
	assert.throwsExceptionWithMessage("La cantidad a retirar debe ser
positiva", { monedero.poner(-20) })
}
```

Ejecutamos los tests y vemos que todo está ok:

[1] Esta variante permite mostrar que el uso de la construcción *describe* es opcional. Si no se necesita compartir estado entre los tests, se pueden definir tests aislados en un archivo *.wtest*



3.1 Esperando un error

Repasemos el test que saca un monto negativo. Allí el test está escrito dentro de un bloque de código que le pasamos al objeto assert enviando el mensaje `throwsExceptionLike`. Nos podemos imaginar lo que hace: esperamos que dentro de ese código se produzca una excepción.

Entonces, si el bloque de código tira una excepción, eso significa que está funcionando de acuerdo a lo "esperado".

Si no recibimos una excepción, esto es una mala señal. Por ese motivo si el bloque de código no falla, el test está mal.
Así es, el test falla.

El objeto assert nos permite hacer diferentes chequeos por error:

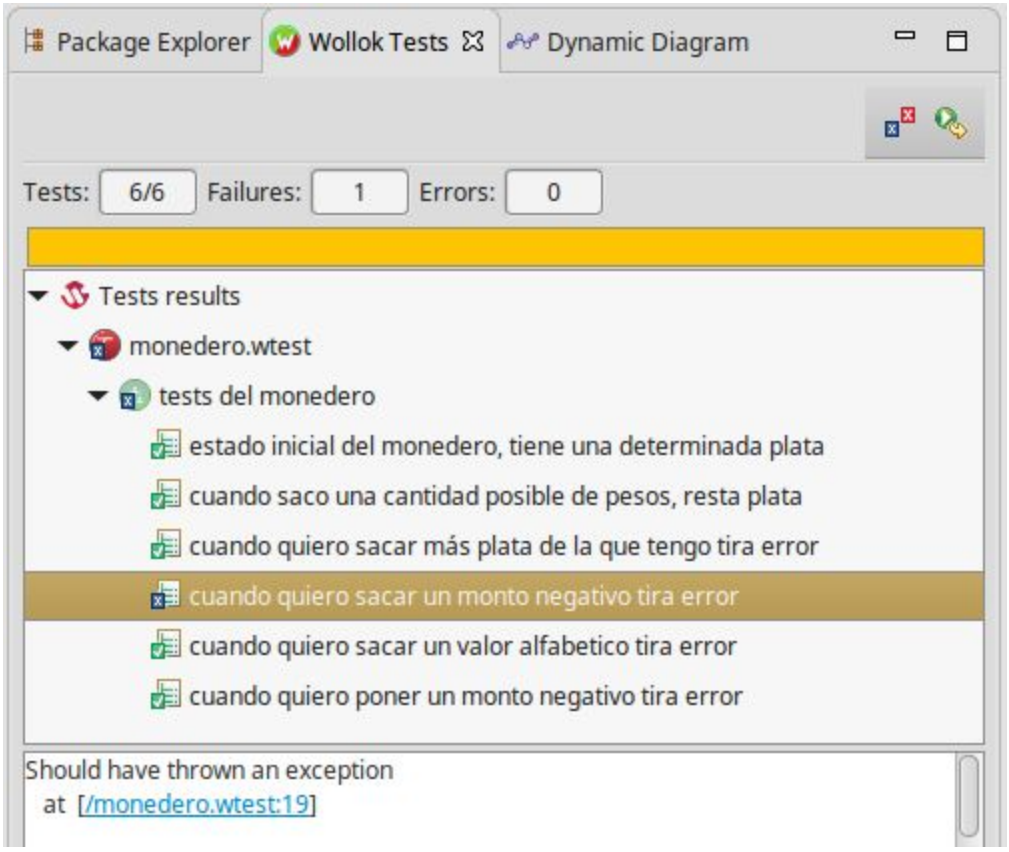
- **`throwsException(block)`:** es el más básico, en el test esperamos un error (no importa cuál).
- **`throwsExceptionWithMessage(errorMessage, block)`:** esperamos un error con un mensaje específico (si no coincide el mensaje el test falla)
- **`throwsExceptionLike(exceptionExpected, block)`:** es la opción más restrictiva, ya que tienen que coincidir
    - el tipo de excepción (UserException en este caso)
    - y el mensaje de error

Para comprobar que el test está funcionando correctamente, vamos a modificar el monedero, comentando la validación de los importes negativos:

```
object monedero {
	...

	method sacar(cantidad) {
//		self.validarMonto(cantidad)
		if (cantidad > plata) {
			throw new UserException(message = "Debe retirar menos
de " + plata)
		}
		plata = plata - cantidad
	}
```

Al ejecutar el test, se produce un error, porque monedero.sacar(-20) no cortó la secuencia, y la siguiente línea dispara un error en el test:

```
test "cuando quiero sacar un monto negativo tira error" {

	assert.throwsExceptionLike(
		new UserException("La cantidad a retirar debe ser
	positiva"),
		{ monedero.sacar(-20) }) // no tira error!
}
```



En cambio, cuando monedero.sacar(-20) dispare la excepción, el assert.throwsExceptionLike internamente atrapará el error y no hará nada, porque es lo que nosotros queríamos que pasara:

```
test "cuando quiero sacar un monto negativo tira error" {
	assert.throwsExceptionLike(
		new UserException(message = "La cantidad a retirar debe ser positiva"),
		{ monedero.sacar(-20) }) // tira error y se atrapa en el assert
}
```

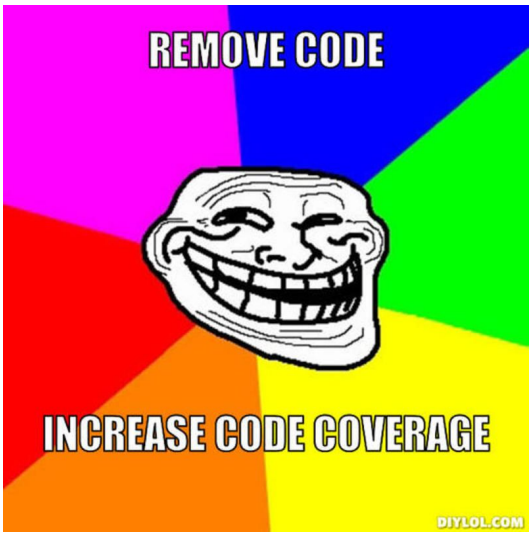
4 ¿Qué probar? ¿Cuánto probar?

Así como vamos tomando conciencia de la importancia de tener tests para probar nuestro código, también vamos notando que lleva tiempo y esfuerzo. Por lo tanto empieza a sobrevolar la pregunta ¿vale la pena probar absolutamente todo?

4.1 Code coverage

¿Qué porcentaje de nuestro código estamos testeando? En estos casos, el 100%, porque se trata de ejemplos breves. Es una condición que rara vez encontraremos en las aplicaciones productivas.

Acabamos de presentar el *code coverage* que es una métrica utilizada para medir qué nivel de cobertura de código tienen nuestros tests[2]. Es un número que va de 0 a 100, donde 1 es muy poco y 100 es *demasiado*. ¿Por qué demasiado? Porque a medida que nos acercamos al 100% nuestros tests pierden valor (por ejemplo porque nos obsesionamos con probar getters, setters, métodos con lógica trivial, etc.) Además, conforme aparecen nuevos métodos de negocio no siempre podemos contemplar y generar los casos de prueba para todos ellos.



No hay un número mágico, es una cuestión de equilibrio y razonabilidad en función de las características del desarrollo y del equipo que lo lleva adelante. Si bien se busca que sea un porcentaje elevado, llega un punto donde dedicar demasiado esfuerzo a incrementar levemente el porcentaje de cobertura se vuelve contraproducente.

4.2 Clases de equivalencia en los tests

Retomando lo que decíamos con el ejemplo de pepita, teniendo un test que le pide a pepita que vuele 5 kilómetros. ¿Tiene sentido hacer otro test que le pida volar 10 kilómetros? ¿y luego otro que le pida a pepita que vuele 6 kilómetros?

No en tanto sabemos que pepita queda con una energía positiva. En cambio, al pedirle a pepita que vuele 91 kilómetros, su energía quedaría negativa: 100 - (91 + 10) = -1. Así como entonces nos pareció que volar 0 km era un caso singular que ameritaba probarse, con más razón éste que deja a pepita en una situación extraña.

Lo que sucede entonces es que los casos de prueba forman **clases de equivalencia** [3], lo que ayuda a definir pruebas representativas que sean de nuestro interés.

[2] Para más información recomendamos leer http://c2.com/cgi/wiki?CodeCoverage
[3] para más información véase https://es.wikipedia.org/wiki/Relaci%C3%B3n_de_equivalencia

El **análisis de los valores límite** de un método nos permite diferenciar dos clases de equivalencia diferentes. En el ejemplo de pepita, pedimos que

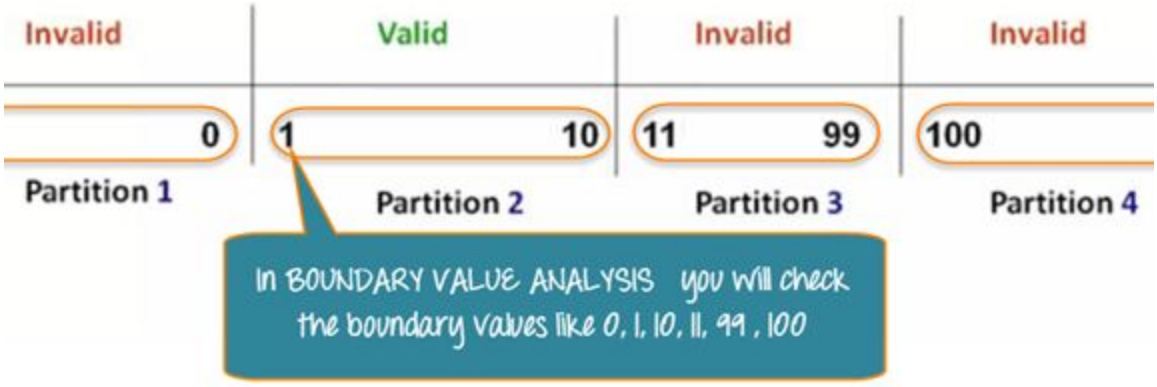
- coma una distancia valor mayor a cero
- y que coma una cantidad nula

Para el método esFuerte(),

```
method esFuerte() { return energia > 50 }
```

una buena prueba consiste en hacer la prueba considerando 50 como valor límite para la energía. Tendríamos entonces

- una clase de equivalencia a “las pepitas que tienen 50 unidades de energía”
- otra clase de equivalencia a “las pepitas que tienen 51 unidades de energía”
- podríamos considerar como clases de equivalencia adicionales las pepitas que tienen menos de 50 de energía o más de 51.



Por último, los **casos de error** también constituyen una importante clase de equivalencia en los tests, como hemos visto en el caso del monedero.

4.3 ¿Pruebas unitarias?

La prueba se considera unitaria en tanto y en cuanto estén probando una unidad funcional. Podríamos asociar la prueba unitaria a la prueba de una única clase/objeto, pero los ejemplos expuestos hasta el momento no son generalizables: ni pepita, ni un empleado ni el monedero tenían dependencias con otros objetos de nuestro dominio. ¿Qué pasaría si estuviéramos probando un cliente que tiene un conjunto de facturas, conformadas por una serie de ítems, cada una con una cantidad y un producto?

Cuando tenemos dependencias entre objetos (no entre tests, que siguen siendo independientes), tenemos que elegir una estrategia de separación de unidades, ya no

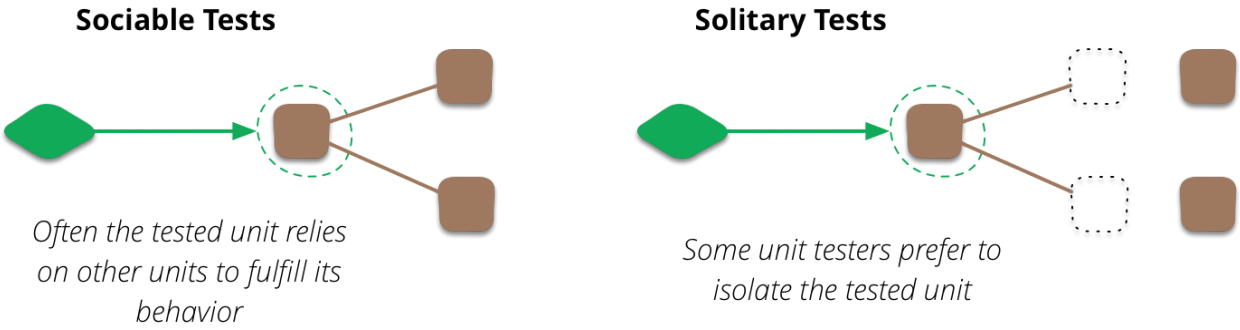
como una clase. Por ejemplo podríamos separar al cliente, a la factura y al producto como tres unidades que cumplen funciones en forma independiente. Más allá de esta elección, no resulta sencillo armar un juego de datos para probar un cliente: ¿puedo encarar los tests de cliente si todavía no construí la factura? ¿puedo hacer pruebas sobre una factura si no terminé de definir el comportamiento del producto?

De la misma manera,

- cuando tenemos efecto colateral (como en el caso de las bases de datos, o al enviar un mail)
- cuando hay un proceso no-determinístico (como un error de red, un momento del día, o una temperatura dada)
- cuando la construcción del juego de datos es compleja
- cuando hay objetos que son poco performantes

nos conviene utilizar objetos que sean polimórficos con los que usaremos en la aplicación "real", pero que tienen un comportamiento simple, rápido, determinístico y fácilmente configurable.

Martin Fowler diferencia los tests en "sociales" y "solitarios"[4]. Mientras que en los primeros se prueba toda la unidad y sus dependencias (en el caso del cliente se estaría utilizando la factura y los productos, y deberíamos esperar a que estén codificadas dichas clases), en los segundos se utilizan implementaciones "de resguardo", lo que facilita mantener la unitariedad del test del objeto cliente.



Ej: Tenemos una clase cliente, y una implementación de prueba de la clase Factura

```
class Cliente {
      const facturas = []

      method agregarFactura(factura) {
```

[4] http://martinfowler.com/bliki/UnitTest.html

```
            facturas.add(factura)
      }

      method saldo() {
            return facturas.sum { factura => factura.saldo() }
      }
}

class FacturaPrueba {
      method saldo() = 100
}
```

¿Para qué sirve generar una factura de prueba que devuelva 100 cuando le pregunto el saldo? Para poder hacer el test del cliente. Si el cliente tiene una factura de prueba, debería asumir que su saldo es 100:

```
import cliente.*

describe "Test de clientes" {
      const cliente = new Cliente()
      const facturaPrueba = new FacturaPrueba()

      fixture {
            cliente.agregarFactura(facturaPrueba)
      }

      test "el saldo del cliente de una sola factura de prueba es el
saldo de esa factura" {
            assert.equals(100, cliente.saldo())
      }

      ...
}
```

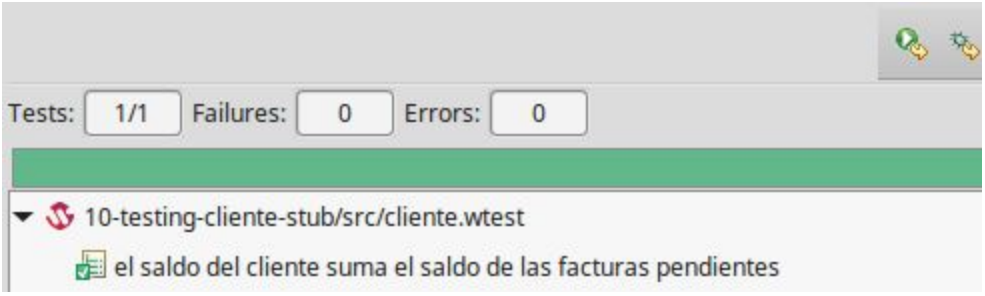
De la misma manera podríamos haber definido un FacturaPrueba cuyo saldo sea configurable en la instanciación:

```
class FacturaPrueba {
      const property saldo
      method estaPendiente() = true // o self.saldo() > 0
}
```

Esto nos permitiría tener varias facturas con montos diferentes, y cambiar el describe:

```
import cliente.*

describe "Test de clientes" {
   const cliente = new Cliente()
   const facturaPrueba1 = new FacturaPrueba(saldo = 150)
   const facturaPrueba2 = new FacturaPrueba(saldo = 250)

   fixture {
      cliente.agregarFactura(facturaPrueba1)
      cliente.agregarFactura(facturaPrueba2)
   }

   test "el saldo del cliente suma el saldo de las facturas
pendientes" {
      assert.equals(400, cliente.saldo())
   }
}
```

Corremos el test, y efectivamente funciona, aun cuando no tenemos la implementación real de la factura[5]:



Otra forma de definir objetos de resguardo es utilizando objetos anónimos, como hemos contado previamente:

```
describe "Test de clientes" {
      const cliente = new Cliente()
      const facturaPrueba1 = self.facturaPrueba(saldo = 150)
      const facturaPrueba2 = self.facturaPrueba(saldo = 250)

      method facturaPrueba(saldo) = object {
            method estaPendiente() = true
            method saldo() = saldo
      }

      fixture ... (idem)
```

[5] Para una visión más profunda de estos conceptos recomendamos el siguiente apunte de Testing

```
}
```

En este caso nuestras facturas de prueba cumplen el rol de objetos impostores o *stubs*. Su sentido es proveer una implementación de resguardo para poder testear un cliente.

5 Resumen

En este capítulo profundizamos el mecanismo de testeo unitario automatizado que provee Wollok. En particular, el describe no solo agrupa los casos de prueba, sino que permite definir variables que pueden ser utilizados por todos los tests. Esas variables referencian a objetos que pueden inicializarse en la misma definición, en un **fixture** o utilizando un **método** dentro del mismo describe. También vimos cómo implementar pruebas cuando el comportamiento esperado es un error.

El diseño de los casos de prueba es una tarea que está lejos de ser repetitiva y mecánica: no hay una metodología que nos asegure que nuestros tests son buenos, aunque algunas métricas como el **code coverage** pueden ayudar (siempre que tengamos en cuenta que es una medida y no un fin en sí mismo).

Por último, para mantener la unitariedad de los tests podemos usar objetos de resguardo (a veces llamados **stubs** o **mocks**) y facilitar así el testeo de una unidad funcional.

El apunte concluye aquí, no así el aprendizaje de la actividad de testing que es mucho más abarcativa, y que dejamos a cargo del lector.