Paradigma Orientado a Objetos

Módulo 05: Introducción al Testeo unitario automatizado

por Lucas Spigariol
Fernando Dodino
y colaboración de
Matías Freyre
Versión 2.0
Agosto 2017

Distribuido bajo licencia Creative Commons Share-a-like

Indice

- Introducción
- Casos de prueba
  - Ejemplo
  - Solución
- Pruebas manuales
  - Consola (REPL)
  - Programas
  - Wollok Game
  - Problemas
- Testeo unitario automatizado
  - Unitario
  - Automático
  - Independencia
  - Información sintética y precisa
- Definición de tests
  - Describe
  - Test
- Ejecución de los test
  - Éxito
  - Fallas
  - Errores
  - Resultados del test
- Resumen
- Apéndice: interfaz de assert

1. Introducción

Cuando hacemos las cosas, ¿de qué manera nos damos cuenta de si las estamos haciendo bien? En general, las probamos. Realizar pruebas o evaluaciones nos permite saber si lo que hicimos cumple con el objetivo. Lo mismo nos pasa cuando construimos software: es importante validar que funcione correctamente.

Si bien hay múltiples aspectos de un determinado software que se pueden analizar, como su eficiencia, su expresividad, su facilidad para realizar modificaciones, etc. y cada una de ellas tiene sentido desde diferentes puntos de vista, en esta oportunidad nos enfocamos en pruebas para determinar específicamente su correctitud, es decir, que el software esté haciendo realmente lo que debe hacer, y no otra cosa. En otras palabras, buscamos quedarnos tranquilos que los resultados que obtenemos al ejecutar un programa son los esperados.

Existen diversas formas de ejecutar código *Wollok* que nos permiten ver qué sucede y de esta manera asegurarnos que haga lo que debe hacer: está la consola interactiva (REPL), se cuenta con la interfaz gráfica que provee *Wollok Game* y se pueden definir programas. Todas ellas tienen en común que son relativamente sencillas y se utilizan de una manera manual, que puede ser útil en determinados contextos, pero que a medida que las soluciones se complejizan no resultan convenientes.

Frente a esta situación, de manera similar a otros lenguajes de programación de la actualidad, *Wollok* ofrece una herramienta para probar el código de una manera mucho más práctica, confiable y profesional, mediante lo que se denominan **tests unitarios**, que tienen la característica fundamental de ser **automatizados**.

2. Casos de prueba

Independientemente del método elegido, lo primero a considerar es qué queremos probar. Incluso antes de tener el código escrito, conociendo los detalles del problema a resolver ya podemos identificar un abanico de situaciones posibles, que vaya desde los casos típicos y frecuentes hasta las situaciones límite o de escasa probabilidad pero que de todas maneras también deben ser contemplados.

Lo importante es poder anticipar, para cada posible situación, cuáles son los resultados correctos que deberá brindar el código, para poder hacer efectivamente una validación. Y aquí, la palabra clave es "anticipar": un problema que suele suceder en la realidad, -cuando se trabaja con poco profesionalismo-, es probar el funcionamiento de una solución sólo viendo que el código ejecuta sin dar error, que arroja algún resultado e inocentemente confiar que es el correcto, sin tener contra qué contrastarlo.

De esta manera, cualquier estrategia de testeo se basa en construir casos de prueba donde se describen las diferentes situaciones de posible ocurrencia y se anticipa cuáles deben ser los resultados correctos a obtener en cada una de ellas. En particular, en el paradigma de la programación orientada a objetos… ¿Qué queremos probar de un objeto? Principalmente, su comportamiento: queremos enviarle mensajes al objeto y verificar que suceda lo esperado.

2.1. Ejemplo

Tomemos como ejemplo a *pepita*. Para determinar los casos de prueba veamos cómo está planteado el problema:

> *"Sabemos que pepita tiene inicialmente 100 unidades de energía. Cuando come alpiste, su energía se incrementa en 4 veces la cantidad de gramos de alimento ingerido. Al volar, su energía disminuye la cantidad de kilómetros que vuela más 10 unidades fijas. Queremos averiguar si pepita es fuerte, lo cual sucede cuando su energía supera las 50 unidades, y también nos interesa saber en cualquier momento cuál es su energía actual"*.

Empecemos por lo que queremos averiguar de pepita

- En la situación inicial de pepita, es decir sin que haya volado ni comido, si le preguntamos la energía, la respuesta debe ser 100.
- En las mismas condiciones, si le preguntamos si es fuerte, la respuesta debería ser afirmativa. (100 > 50)

Con esto, ya tenemos dos casos de prueba.

Sigamos con los otros requerimientos, que no son consultas que tienen una respuesta como resultado, sino que son indicaciones que deben provocar un cambio en el objeto.

- Al decirle a pepita que vuele 5 kms, asumiendo que estaba en su estado inicial, su energía debe quedar en 85. (100 - (5 + 10) = 85)
- Si pepita, en su estado inicial, recibe la orden de comer 120 gramos de alpiste, su energía pasa a ser 580. (100 + (120 * 4) = 580)

Y tenemos de esta manera dos nuevos casos de prueba.

Podríamos querer probar qué sucede con otros valores, por ejemplo, si pepita volase otra cantidad de kilómetros o comiese otra cantidad de alpiste. Sería no sólo tedioso (e interminable) plantear casos de prueba con todos los valores posibles sino fundamentalmente irrelevante, pero es recomendable prestar atención a ciertos valores

que representen situaciones particulares. Por ejemplo, sería interesante pedirle a pepita que vuele 0 km (que podría significar que arranca a volar y se arrepiente instantáneamente) o que coma 0 gramos (que, aunque parezca trivial, también es una orden válida) que son casos singulares que ayudan a garantizar que el objeto funcione correctamente en todos los casos.

Por lo tanto, obtenemos dos casos de prueba más:

- Al decirle a pepita que vuele 0 kms, asumiendo que estaba en su estado inicial, su energía debe pasar a ser 90. (100 - (0 + 10) = 90)
- Si pepita, en su estado inicial, recibe la orden de comer 0 gramos de alpiste, su energía sigue siendo la inicial de 100. (100 + (0 * 4) = 100)

Otra situación interesante a probar es que la pregunta por si pepita es fuerte tenga respuestas negativas cuando realmente corresponda. Para este caso, no basta con la situación inicial porque ya vimos que allí la respuesta debe ser afirmativa; lo que tenemos que hacer es preguntarle si es fuerte habiendo previamente volado una suficiente cantidad de kilómetros. Obtenemos así un nuevo caso de prueba.

- Partiendo de la situación inicial, si hacemos que pepita vuele 60 kilómetros de manera que su energía llegue a 30 (100 - (10 + 60)), ante la pregunta si es fuerte la respuesta debe ser negativa (30 < 50).

Por último, es también útil probar los casos límite. Ya vimos qué sucede con los niveles de energía menores y mayores al límite. Y si es exactamente el límite, ¿qué debería pasar? De acuerdo con la definición, es fuerte cuando SUPERA las 50 unidades. Entonces, nuestro último caso sería.

- Partiendo de la situación inicial, si hacemos que pepita vuele 40 kilómetros de manera que su energía llegue a 50 (100 - (10 + 40)), ante la pregunta si es fuerte la respuesta debe ser nuevamente negativa (porque no se cumple 50 > 50).

Esquemáticamente, los casos de prueba planteados son:

Acciones	Verificación
ninguna	pepita queda con energía 100
ninguna	pepita es fuerte
que pepita vuele 5 kilómetros	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 0 kilómetros	pepita queda con energía 90
que pepita vuele 60 kilómetros	pepita no es fuerte
que pepita vuele 40 kilómetros	pepita no es fuerte

2.2. Solución

Para poder continuar con la explicación, veamos el código de una posible solución al problema planteado:

```
object pepita {
	var energia = 100

	method energia() { return energia }

	method esFuerte() { return energia > 50 }

	method volar(kms) {
		energia = energia - (kms + 10)
	}

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

Como se puede observar, tenemos definido al objeto pepita, con una variable que representa la energía y cuatro métodos, uno para cada requerimiento. Entre los métodos, dos de ellos tienen valores de retorno y se corresponden con los mensajes que representan preguntas, consultas o interrogantes a pepita, mientras que los otros dos tienen efecto, es decir provocan un cambio de estado en pepita, y se corresponden con los mensajes que son pedidos, órdenes o indicaciones para que pepita realice.

3. Pruebas manuales

Ahora que tenemos una solución, una alternativa es probarla con las herramientas que ya conocemos. Vamos haciendo cada una de las pruebas y mientras lo que observamos coincida con lo que esperamos, seguimos avanzando sin problemas, pero cuando obtenemos un resultado diferente al esperado, que es cuando la prueba que

hicimos tiene sentido, interrumpimos las pruebas para arreglar el problema. Luego volvemos a probar.

3.1. Consola (REPL)

Abrimos una consola y nos ponemos a interactuar con pepita enviándole mensajes, para ver allí mismo ver su resultado.

Para el primer caso de prueba, hacemos

```
>>> pepita.energia()
100
```

Si nos fijamos en los casos de prueba que teníamos planteados, podemos comprobar que efectivamente era el resultado esperado. Luego queremos saber si es fuerte, y lo hacemos de manera similar.

Cuando llega el momento de probar cómo vuela, hacemos

```
>>> pepita.volar(5)
>>>
```

Como se trata de un mensaje que es una orden, que no tiene retorno, no vemos nada. Entonces, inmediatamente usamos el otro mensaje, haciendo

```
>>> pepita.energia()
85
```

Tenemos que manualmente verificar que sea el valor que habíamos predicho.

Luego, queremos ver si funciona bien el comer, pero para ello sería un error enviar inmediatamente el mensaje. Como la consola conserva el estado del ambiente con todos los objetos, el efecto de los mensajes es acumulativo. Si queremos evaluar el caso de prueba planteado anteriormente, debemos volver a ejecutar la consola para que vuelva todo al estado inicial y recién allí enviar el mensaje comer, con el argumento indicado y ver qué sucede.

Así, podemos ir probando cada caso, teniendo la precaución de plantear cada vez la secuencia correcta y reiniciando la consola oportunamente.

3.2. Programas

Wollok permite crear programas (en archivos con extensión *wpgm*) en los que se especifica una secuencia de código que se ejecuta completa sin necesidad de interacción. En el programa se escribe la serie de mensajes que se envían a los objetos, y teniendo la precaución de mostrar por consola los resultados a medida que son enviados, podemos observar su funcionamiento.

Por ejemplo, podríamos escribir

```
import pepita.*

program pepitaFuerte {
	console.println("En la situación inicial")
	console.println("¿Pepita es fuerte?")
	console.println(pepita.esFuerte())
	pepita.volar(60)
	console.println("Luego de volar 60 kilómetros")
	console.println("¿Pepita es fuerte?")
	console.println(pepita.esFuerte())
}
```

Al ejecutarse, el resultado que vemos por consola sería

```
En la situación inicial
¿Pepita es fuerte?
true
Luego de volar 60 kilómetros
¿Pepita es fuerte?
false
```

Contrastando manualmente con los valores que anticipamos en los casos de prueba correspondiente, podemos verificar que esta parte de nuestra solución funciona correctamente.

Podríamos agregar más líneas de código a este mismo programa para contemplar los casos de prueba restantes o hacer nuevos programas, y teniendo la precaución de mostrar en cada caso los resultados de una manera entendible por la consola e interpretarlos adecuadamente, logramos el objetivo de analizar todos los casos de prueba.

3.3. Wollok Game

Una variante es utilizar *Wollok Game*, que nos permite que los objetos sean representados gráficamente en una interfaz de usuario. Ya sea que configuremos eventos en dicha interfaz para que se envíen mensajes a los objetos ante su ocurrencia, que los escribamos uno por uno en la consola o que los hayamos predefinido en una secuencia mediante un programa, a medida que el objeto va ejecutando los métodos correspondientes, se modifica su representación gráfica y vemos su efecto en la pantalla. Por ejemplo -modificando adecuadamente la solución planteada- podemos tener dos imágenes diferentes, una para representar a pepita fuerte y otra para pepita no fuerte, mostrar en un recuadro visible en todo momento la energía de pepita, hacer que el volar implique también un desplazamiento en la pantalla, o lo que se nos ocurra para poder comprobar visualmente lo esperado en cada uno de los casos de prueba[1].

3.4. Problemas

Con cualquiera de estas estrategias las pruebas son validadas **manualmente.** Si bien pueden resultar prácticas en ciertas ocasiones o problemas sencillos, traen aparejados una serie de problemas que aumentan exponencialmente a medida que la cantidad de código aumenta:

- Es tedioso tener que recordar cada vez que se quiere probar algo cuál es el conjunto de casos de prueba a considerar. Resulta reiterativo escribir nuevamente la misma serie de consultas.
- La verificación manual de cada uno de los valores esperados frente a cada uno de los mensajes, no solo resulta poco práctico sino que es propenso a equivocaciones y se corre el riesgo de aceptar como válido cualquier valor que no sea exactamente el esperado.
- Puede demandar un esfuerzo considerable el envío de mensajes para llevar a un objeto al estado deseado para una determinada prueba.
- En la consola, si luego de enviar numerosos mensajes se detecta un error y se realizan cambios en el código para corregirlo, suele ser dificultoso reproducir la misma secuencia para verificar si el cambio fue adecuado.
- En un programa, cuando se envía una sucesión de mensajes y el resultado final no es el esperado, es dificultoso detectar en cuál de todos los mensajes se produjo el problema.

[1] Para más detalles pueden descargar el ejemplo https://github.com/wollok/pepitaGame

- Al realizar cambios en el código y por lo tanto modificar los casos de prueba, el tiempo que hay que invertir cada vez en probar es similar al que se dedicó originalmente.
- Frente a un cambio o corrección en el código, si solo probamos lo que se modificó, no siempre se puede estar seguro que no se haya afectado a otras partes del sistema, por lo que debería volver a hacerse las pruebas anteriores, con todo lo que ello implica.

4. Testeo unitario automatizado

Teniendo un panorama más completo de las limitaciones, riesgos y dificultades que tienen estas formas manuales de probar la correctitud de una solución, cobra sentido la implementación de otra forma de realizar pruebas que no solo simplifique y sistematice la tarea, sino que garantice su confiabilidad.

Para ello, *Wollok* cuenta con una herramienta para definir **test unitarios automatizados**, que se inscribe en la tendencia actual de los lenguajes de programación de uso profesional de utilizar el concepto **Unit Test** como parte del proceso de desarrollo de software.

Un test, esencialmente, es una porción de código *Wollok* en la que se describe una determinada situación, mediante el envío de mensajes a los objetos correspondientes, y se especifica cuál es el resultado esperado.

4.1. Unitario

El tipo de testeo que abordamos se califica como "unitario", porque la estrategia, en vez de hacer una gran y extensa prueba de un programa completo, se basa en identificar unidades significativas del código y probar casos puntuales donde éstas intervengan. Vamos a definir tests para los casos de prueba, explicitando mediante código la vinculación entre las situaciones planteadas y los resultados esperados.

Si mediante un mismo test probamos muchas cosas a la vez, cuando el test falla es más difícil saber cuál de los múltiples motivos falló. En cambio, si en un test probamos una sola cosa, al encontrarnos con que falla, tenemos una mayor certeza sobre cuál es el problema a corregir. Por este motivo, es fundamental el carácter unitario del test.

Los mismos principios de modularización y delegación de responsabilidades que caracteriza al paradigma de objetos, ayudan a que sea relativamente sencillo identificar las unidades sobre las que se van a definir los test.

Por otra parte, la forma en que anteriormente planteamos los casos de prueba ya tuvo en cuenta implícitamente los criterios de unicidad, por lo que -entendidos de esta manera- a cada caso de prueba le va a corresponder un test específico.

4.2. Automático

La característica fundamental de los tests es que se puede automatizar su ejecución y consecuentemente su validación.

Como en el código de cada test, junto a la descripción de la situación a probar se especifica el resultado esperado, la validación es realizada por la misma máquina, sin mediar la interpretación de la persona, logrando mayor velocidad y confiabilidad.

A su vez, tratándose de código escrito, los tests quedan guardados al igual que el código de la solución propiamente dicha y cuando se desea, se pueden ejecutar todos de una vez, obteniendo un informe del resultado de cada uno de ellos. De esta manera, cuando en el proceso de desarrollo, al correr los tests se detecta algún problema y se corrige la solución, es sencillo volver a ejecutar todas las pruebas anteriores. También, si una solución que ya se probó que funciona correctamente se quiere refactorizar, basta con correr nuevamente todos los tests para garantizar que sigue funcionando adecuadamente.

Ante el aumento en la cantidad y variedad de tests, éstos se pueden agrupar y organizar de diferentes maneras, para permitir variantes en su ejecución.

4.3. Independencia

Una característica que se desprende de su carácter unitario y que posibilita la automatización de los tests sin generar conflictos entre ellos, es que cada uno se concibe en forma independiente de cualquier otro test.

La lógica de ejecución de tests parte del supuesto que cada uno se corre a partir de la situación inicial del sistema, es decir que el ambiente se reinicia entre test y test, garantizando su total independencia.

Además, ante la detección de un problema por parte de algún test, los demás pueden seguir ejecutándose sin inconvenientes. El informe final detalla cuáles tests ejecutaron sin inconvenientes y cuáles detectaron algún problema.

4.4. Información sintética y precisa

Teniendo en cuenta que la importancia de una prueba está dada por su capacidad para detectar problemas, la información más valiosa que aporta no es cuando funciona,

sino cuando nos permite darnos cuenta que el resultado obtenido no es el esperado. En este caso, es importante que nos oriente con precisión acerca de dónde se produjo el problema y nos pueda brindar la mayor información posible.

Ya sea que se decida ejecutar uno, muchos o todos los tests, cuando el test corre bien, la información que arroja es bien sintética, simplemente dice que funcionó: no hay nada más que agregar que nos distraiga. En cambio, cuando un test falla, nos advierte con mayor contundencia y detalla cuál era el resultado esperado y el que realmente encontró.

5. Definición de tests

Los tests se definen en el mismo proyecto en que está la solución propiamente dicha, pero en archivos aparte. Pueden crear un archivo con extensión ***.wtest***, con un nombre representativo diferente al de los otros archivos .wlk donde se encuentran las definiciones. Si usaron el comando ***wollok init*** para inicializar el proyecto el mismo ya les crea un archivo de tests llamado ***testExample***.

El elemento clave es **assert,** un objeto autodefinido de la biblioteca *lib*, que entiende los mensajes que se utilizan en el siguiente ejemplo (y otros más que oportunamente presentaremos) y que permite una forma de plantear las expectativas declarativamente, delegando internamente al framework de testing que provee Wollok toda la lógica de la ejecución y validación automatizada.

Retomando el ejemplo de pepita, los tests quedarían de esta manera:

```
import pepita.*

describe "Tests de Pepita" {

	test "energía inicial de pepita" {
		assert.equals(100, pepita.energia())
	}

	test "pepita comienza siendo fuerte" {
		assert.that(pepita.esFuerte())
	}

	test "pepita vuela y baja su energía" {
		pepita.volar(5)
		assert.equals(85, pepita.energia())
	}

	test "pepita come y sube su energía" {
```

```
        pepita.comer(120)
        assert.equals(580, pepita.energia())
    }

    test "pepita empieza a volar, no vuela y la energía baja" {
        pepita.volar(0)
        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 muchos kilómetros, y ya no es fuerte" {
        pepita.volar(60)
        assert.notThat(pepita.esFuerte())
    }

    test "pepita vuela pocos kilómetros, y ya no es fuerte" {
        pepita.volar(40)
        assert.notThat(pepita.esFuerte())
    }
}
```

Al inicio, necesitamos hacer el **import** para incluir el código a testear, indicando el nombre de archivo donde se encuentran definidos los objetos. Se recomienda poner *nombreDeArchivo*.* para contemplar todas las entidades definidas en dicho archivo.

5.1. Describe

Identificamos a un conjunto de tests con la palabra reservada **describe** y un nombre expresivo con el cual lo identificamos. Al igual que los objetos y las clases, se utilizan { } para delimitar el inicio y fin de de la presente entidad, agrupando a todos los tests que forman parte de ella.

5.2. Test

Dentro del describe, cada test se define de la siguiente manera:

- Cada uno comienza con la palabra reservada **test** seguido de una cadena de caracteres que explique lo que se está probando. Es importante ponerle una

buena descripción porque esa misma leyenda es la que va a aparecer en el informe de errores, y cuando el test falle nos va a permitir detectar más fácilmente cuál fue el problema: es bueno identificar la clase de equivalencia, es decir, el escenario de prueba y no entrar en detalles redundantes (decir que la energía queda en 85 implica que si cambian las reglas de cálculo debemos cambiar en dos lugares).

- Se usan { } para marcar el inicio y fin del test, de igual manera que para los métodos.
- En las primeras líneas, salvo tests muy sencillos, generalmente es necesario enviar los mensajes que van configurando la situación a probar. Es lo que en los casos de pruebas denominamos "acciones". Algunas bibliografías separan la instanciación de objetos de prueba y las acciones propiamente dichas, en lo que se denomina el patrón Arrange-Act-Assert.
- Por último se le envía un mensaje al objeto **assert** para realizar concretamente la prueba, contrastando lo que se que se quiere probar con lo que se espera. Es lo que en los casos de prueba titulamos "validación". Hay básicamente tres alternativas:
    - El mensaje típico es **equals**, con dos parámetros: en primer lugar va el valor que se espera que retorne el mensaje que va en segundo lugar. Se lo puede ver en la mayor parte de los tests.
    - En los casos donde se espera que la respuesta sea **true**, se le envía el mensaje **that**, con un único parámetro que es el mensaje booleano que se espera que se valide afirmativamente. Se lo puede ver en el segundo test.
    - En forma análoga, existe un mensaje **notThat** que testea que no se verifique el mensaje booleano. Se lo puede ver en el último de los tests.

En los primeros dos tests, como se trata de probar mensajes que tienen valor de retorno y parten de la situación inicial, no hizo falta enviar mensajes previo al assert. Del tercero al último se busca probar que funcionen adecuadamente métodos con efecto como volar o comer, se envían primero dichos mensajes y luego el mensaje que se coloca en el assert es uno muy sencillo que se limita a exhibir el efecto causado. Los últimos 2 tests son similares en su estructura, aunque lo que se desee probar sea el mensaje que va en el assert.

Dentro de un test escribimos un assert por caso de prueba (esto puede implicar uno o más asserts en el test, lo importante es que todos tengan que ver con el mismo escenario que estamos probando).

También es válido tener tests "sueltos", sin declarar el **describe**, pero lo que se estila y recomienda es agruparlos por su coincidencia en cuanto a los objetos que

utilizan. Más adelante explicaremos otras características que refuerzan la idea de agrupar los test.

Es posible tener más de un describe por archivo, aunque si pertenecen a unidades diferentes (ej: aves, entrenadores) es conveniente definirlo en archivos diferentes.

6. Ejecución de los test

Teniendo definidos los tests, es momento de probar nuestra solución.

Podemos correr los test directamente desde el *menú contextual* que ofrece el IDE, ya sea un solo test, un solo describe o todo el archivo.

```
Run all tests
import pepita.*

Run describe
describe "Tests de Pepita" {

    Run test
    test "energía inicial de pepita" {
        assert.equals(100, pepita.energia())
    }
```

En caso que se tengan definidos varios archivos de tests, además de ejecutarlos uno por uno de la manera explicada, se los puede correr todos juntos usando el comando Ctrl + R (en Windows) parados dentro de alguno de los archivos de test.

6.1. Éxito

Asumiendo que contamos con el código de la solución y los tests tal como se plantearon, al ejecutar “Run all tests”, se corren todos los tests definidos en el archivo y se muestra lo siguiente por consola:

```
Running 8 tests...
 testsPepita
   "Tests de Pepita"
     ✓ "energía inicial de pepita"
     ✓ "pepita comienza siendo fuerte"
     ✓ "pepita vuela y baja su energía"
     ✓ "pepita come y sube su energía"
     ✓ "pepita empieza a volar, no vuela y la energía baja"
     ✓ "pepita come 0 gramos y la energía se mantiene"
     ✓ "pepita vuela muchos kilómetros, y ya no es fuerte"
     ✓ "pepita vuela pocos kilómetros, y ya no es fuerte"


 ✓ 8 passing
```

Nombre del archivo
Nombre del describe
Nombre del test

¿Cuál es nuestra salida? En este caso, todos los tests pasaron las verificaciones exitosamente: pasaron 8 tests de los 8 que se ejecutaron, obteniendo para cada test un tilde verde y la descripción
Nos quedamos tranquilos que nuestra solución funciona.

6.2. Fallas

En nuestro primer ejemplo pasaron correctamente todos los tests, ciertamente es lo deseable, pero es algo que no siempre nos va a pasar. Modifiquemos el código para que el programa no funcione adecuadamente. ¿Qué pasa si hacemos que pepita al volar le sume 5 como valor fijo en lugar de 10?

```
method volar(kms) { energia = energia - (kms + 5) }
```

Ejecutamos nuevamente los tests:

```
Running 8 tests...
 testsPepita
   "Tests de Pepita"
     ✓ "energía inicial de pepita"
     ✓ "pepita comienza siendo fuerte"
     ✗ "pepita vuela y baja su energía"
     ✓ "pepita come y sube su energía"
     ✗ "pepita empieza a volar, no vuela y la energía baja"
     ✓ "pepita come 0 gramos y la energía se mantiene"
     ✓ "pepita vuela muchos kilómetros, y ya no es fuerte"
     ✗ "pepita vuela pocos kilómetros, y ya no es fuerte"
```

¿Cuál es nuestra salida ahora? Vemos que algunos test corrieron correctamente y otros fallaron, detectando problemas en el código.

- Todos los tests con su descripción, los validados con tilde verde y los que fallaron con un tilde rojo
- Debajo podemos ver la información detallada sobre los test que fallaron, por ejemplo:

```
✗ testsPepita."Tests de Pepita"."pepita vuela y baja su energía"
  wollok.lib.AssertionException: Expected <85> but found <90>
    at testsPepita."Tests de Pepita"."pepita vuela y baja su energía" [testsPepita.wtest:13]
```

Que aparezca como ***AssertionException*** nos dice que es una ***falla***, lo que implica que pudo terminar de ejecutar el test pero se obtuvo un valor distinto al esperado.

Detectó precisamente la diferencia entre lo que esperaba el test y el resultado que arroja nuestro código: al volar 5 kilómetros, hizo (100 - (5 + 5)) = 90 que difiere de los 85 esperados de acuerdo a la definición original del problema.

:

6.3. Errores

Ahora veamos qué pasa si modificamos el código de la solución, de una manera más contundente que eventualmente produzca un error de ejecución. Si gramos es 0, como ocurre en uno de los tests, la división da un error:

```
method comer(gramos) {
      energia = energia + 4 / gramos
}
```

Corremos nuevamente los tests:

```
Running 8 tests...
 testsPepita
   "Tests de Pepita"
     ✓ "energía inicial de pepita"
     ✓ "pepita comienza siendo fuerte"
     ✗ "pepita vuela y baja su energía"
     ✗ "pepita come y sube su energía"
     ✗ "pepita empieza a volar, no vuela y la energía baja"
     ✗ "pepita come 0 gramos y la energía se mantiene"
     ✓ "pepita vuela muchos kilómetros, y ya no es fuerte"
     ✗ "pepita vuela pocos kilómetros, y ya no es fuerte"
```

```
✗ testsPepita."Tests de Pepita"."pepita come 0 gramos y la energía se mantiene"
  wollok.lang.EvaluationError wrapping TypeScript RangeError: other
    at pepita.pepita.comer(gramos) [pepita.wlk:12]
```

Podemos ver que ahora nos figura como ***EvaluationError***, lo que nos dice que tuvimos un ***error*** que no permitió que termine de ejecutar el test. Ahora también nos dice en qué línea de código se generó el error, en este ejemplo fue en la línea 12 de pepita.wlk, que justamente es donde se encuentra el método *comer* que habíamos modificado.

6.4. Resultados del test

Cada uno de los test tiene tres salidas posibles

- ¡Todo bien! Ok. El código hace lo que esperamos.
- ¡Falló! El código no hace lo que esperábamos. ¡Menos mal que hicimos un test para detectarlo!
- ¡Error! No se pudo hacer la validación. Hay errores que impidieron completar su ejecución.

Es importante notar que cuando hubo fallas o errores, no tuvimos que tocar el código en los tests. Cambiamos el código de pepita, y pudimos verificar su comportamiento en todos los casos ejecutando el mismo conjunto de tests. Es lo que se denomina un **test de regresión**[2].

7. Resumen

El testeo unitario es una herramienta fundamental para poder garantizar que cada objeto está cumpliendo con la responsabilidad que le fue encomendada. Para lograr este objetivo, además de las tradicionales pruebas manuales, podemos definir tests unitarios que son automatizados, independientes y aislados, que especifican una serie de condiciones a verificar. Tener una gran batería de los tests no es sinónimo de un correcto diseño, pero constituye un buen punto de partida. Además fomenta el abandono de malas prácticas como “código que anda no se toca”, por una mucho más sana en la cual el miedo a modificar el software se contrasta con la posibilidad de correr pruebas de regresión todas las veces que sea necesario.

[2] Pruebas que se hacen sobre una pieza de software existente que sufrió modificaciones. Para más información ver Regression Testing.

Apéndice: interfaz de assert

`assert.that(condition)`
Verificar que la condición se cumple `assert.that(cliente.esMoroso())`
`assert.notThat(condition)`
Verificar que la condición sea falsa (false = ok) `assert.notThat(cliente.esMoroso())`
`assert.equals(actual, expected)`
Verificar que dos valores sean iguales `assert.equals(90, pepita.energia())`
`assert.notEquals(actual, expected)`
Verificar que dos valores sean distintos `assert.notEquals(90, pepita.energia())`
`assert.throwsException(block)`
Verificar que al ejecutar block (un bloque de código) se produce un error cualquiera `assert.throwsException({monedero.sacar("A")})`
`assert.throwsExceptionLike(exceptionExpected, block)`
Verificar que al ejecutar block (un bloque de código) se produce la misma excepción que la exceptionExpected, con el mismo mensaje assert.throwsExceptionLike(new BusinessException("hola"), { throw new BusinessException("hola") } ✅ assert.throwsExceptionLike(new BusinessException("chau"), { throw new BusinessException("hola") } ⚠️(no coincide el mensaje) assert.throwsExceptionLike(new OtherException("hola"), { throw new BusinessException("hola") } ⚠️(no coincide la excepción)
`assert.throwsExceptionWithMessage(errorMessage, block)`
Verifica que al ejecutar block (un bloque de código) se produce el mismo mensaje de error que el esperado (errorMessage), sin importar el tipo de excepción que se trate assert.throwsExceptionWithMessage("hola",{ throw new UserException("hola") } ✅ assert.throwsExceptionWithMessage("hola",{ throw new OtherException("hola") } ✅ assert.throwsExceptionWithMessage("chau",{ throw new UserException("hola") }⚠️