¿Para qué testeamos?

Probar una aplicación permite verificar que el sistema (o una parte de él) funciona de acuerdo a lo especificado. Los tests permiten bajar la incertidumbre y aumentar la confianza que tenemos sobre el software que desarrollamos.

Pruebas unitarias

Las pruebas unitarias las hace el desarrollador sobre un componente o unidad funcional, esto puede ser un objeto o un conjunto de ellos (ej: un alumno se inscribe a materias, la prueba unitaria sobre el alumno también puede incluir inscripciones a materias).

En este curso vamos a pedir que las pruebas que realicen sean:

- Automatizadas
- Repetibles
- Independientes entre sí (no debe haber una secuencia en la cual correr los tests)

Automatizar los tests requiere invertir un cierto tiempo, pero ese esfuerzo vale la pena dado que:

- Una vez programado, puedo testear n veces sin tener que invertir un tiempo adicional (como ocurre con la prueba manual). Con lo de n veces estamos hablando de que sean *repetibles*.
- Eso me permite agregar funcionalidad o mejorar el código y medir si esa modificación introdujo algún error en el sistema
- También permite entender qué output espero en base a un determinado input

Prueba automatizada y debugging

Muchas veces necesitamos encontrar errores o entender cómo funciona una aplicación a partir de técnicas de depuración, donde vemos el estado de un sistema en diferentes momentos. Esta es una herramienta muy útil, que la podemos aplicar a los tests inclusive, no obstante hay que considerar algunas limitantes del debugging:

- Mientras que cada test automatizado que se construye queda, el tiempo que utilizamos al debuggear no se aprovecha para ocasiones posteriores: siempre lleva el mismo tiempo.
- Debuggear es una actividad tediosa y aumenta en complejidad cuando la cantidad de variables aumenta.
- Es más fácil perder el foco después de 20 minutos de debuggear. Cuando codificamos tests unitarios automatizados, la herramienta guarda en una lista los tests correctos y los erróneos, permitiendo atacarlos de a uno por vez.

¿Qué necesito para desarrollar los tests de mi TP?

Un juego de datos o fixture

Es un conjunto de objetos que van a formar parte de los tests. **Por ejemplo:** un auto veloz y uno que anda despacio, un doctor que sólo atiende pacientes particulares y otro que atiende por obra social, un alumno que aprobó una materia, otro que no aprobó ninguna, etc.

Los tests propiamente dichos

Se plantean como condiciones que deben cumplirse.
¿Quiénes deben cumplir esas condiciones? Los objetos que formen parte del juego de datos.

Una cosa importante que vamos a asumir es que **no habrá efecto colateral** sobre el juego de datos. De esa manera nos garantizamos que ningún test se vea afectado por otro test que haya corrido previamente (así logramos que sean **independientes**). *Ejemplo:* si defino al alumno Pedrito que aprobó Paradigmas y Análisis I y en alguno de los tests lo hago aprobar Sistemas Operativos, al programar un nuevo test no debería asumir que Pedrito tiene 3 materias aprobadas en lugar de 2. En todos los tests en los que use ese alumno debo asumir que solamente aprobó Paradigmas y Análisis I.

¿Qué debemos testear?

Una regla heurística nos plantea que deberíamos probar todo "lo que se pueda romper". Un getter, un setter, un método que inicialice posiblemente no merezca nuestra atención. Pero por lo general los tests vienen dados por nosotros, de manera que por lo general tendrá el formato

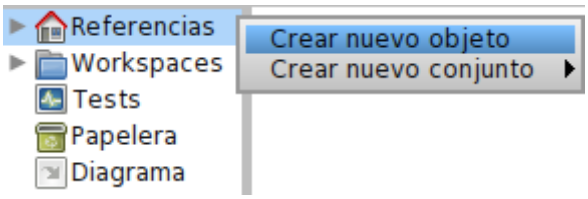
(opcional) Acciones que modifiquen el estado de un objeto. **Condición que el objeto *x* debe cumplir**	Valor esperado (un número, un booleano, etc.)

Test unitarios en Object Browser

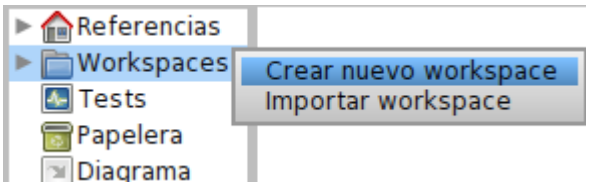
Juego de datos

El juego de datos o fixture se define de la siguiente manera:

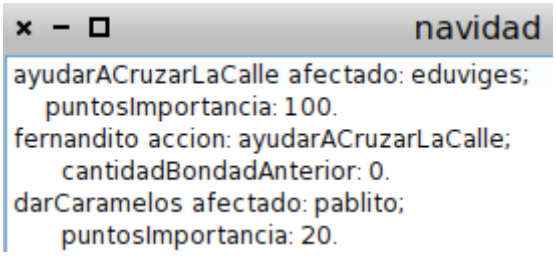
- Los objetos, se crean directamente desde el botón de Referencias > Crear nuevo objeto



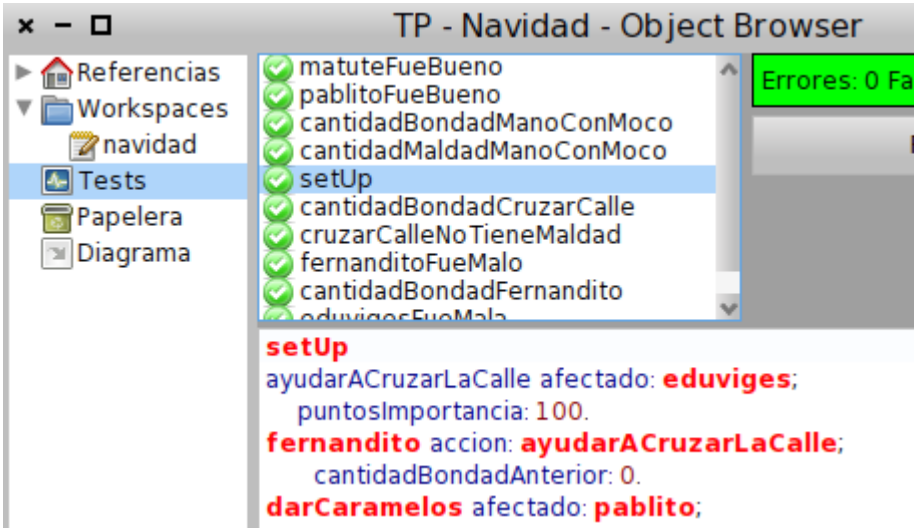
- Cada objeto tiene que inicializarse correctamente: esto es conocer a los objetos que tiene que conocer. Eso lo podemos escribir en un Workspace:



Y definiendo las relaciones entre objetos (enviando mensajes setters/initialize):



- Pero escribir en el workspace no nos sirve si queremos que cada test sea independiente, de manera que lo que tenemos que hacer es pasar ese workspace a un método setUp (las mayúsculas y minúsculas hay que respetarlas, así que es U mayúscula y el resto en minúscula). El método setUp debemos escribirlo dentro de la solapa Tests.



Tests

Parado en la solapa Tests, cada test se implementa como un método. En ese método definimos una aserción: algo que se debería cumplir. Pero en lugar de escribir:

```
testJuanEsBueno
      ^juan esBueno ifTrue: [ 'Todo bien' ]
                    ifFalse: [ 'Todo mal' ]
```

Lo que hacemos es trabajar un poco más declarativamente:

```
testJuanEsBueno
     ^self assert: juan esBueno
```

En general es una buena práctica tener varios tests que prueben cosas distintas. Eso nos permite mayor granularidad: si de 3 condiciones una no se cumple podemos saberlo en un golpe de vista. Si escribimos muchas condiciones en un test y la primera no se cumple, no sabemos qué pasa con las otras.

Mensajes que podemos enviar desde un test

- assert: *unBoolean* (espera que el boolean sea true)
- deny: *unBoolean* (espera que el boolean sea false)
- assert: *mensaje a un objeto* equals: *un valor* (en caso de no cumplirse se muestra un error estándar)
- assert: *unBoolean* description: *mensaje de error a mostrar en el test en caso de falla* (permite que el mensaje sea más descriptivo para el usuario)

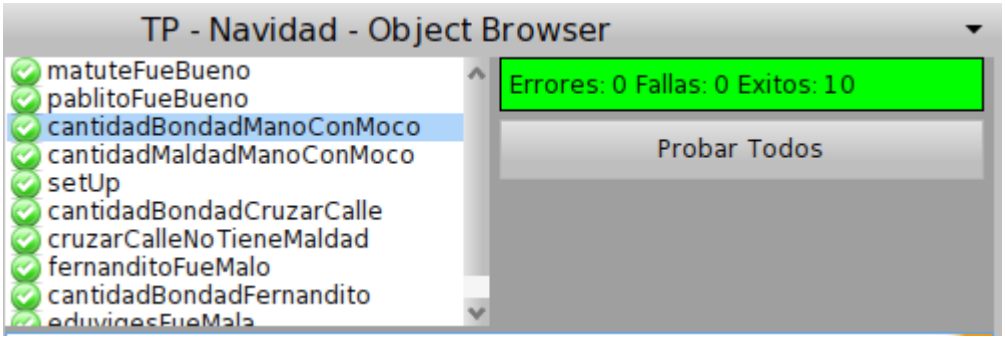
Ejecución de los tests

Fallas y errores

El resultado de un test puede ser

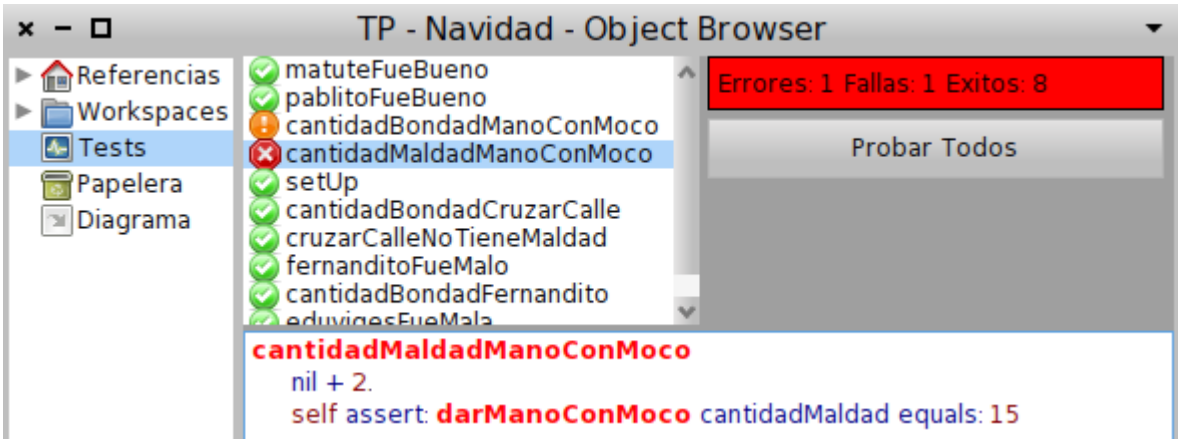
- **ok (verde)**: el resultado del test es el esperado
- **falla (amarillo/naranja)**: el resultado del test no es el esperado
- **error (rojo)**: al enviar mensajes a los objetos se produce un error que impide saber si el resultado es ok/falla (por ejemplo, por un doesNotUnderstand)

Al ejecutar los tests, si todos dieron ok veremos un semáforo verde arriba a la derecha, siempre parados en la solapa de tests:



Y a la izquierda de cada método, podemos ver el estado del test (verde, amarillo o rojo).

Si hay test con fallas o errores, los podremos ver al disparar los tests:



En este caso

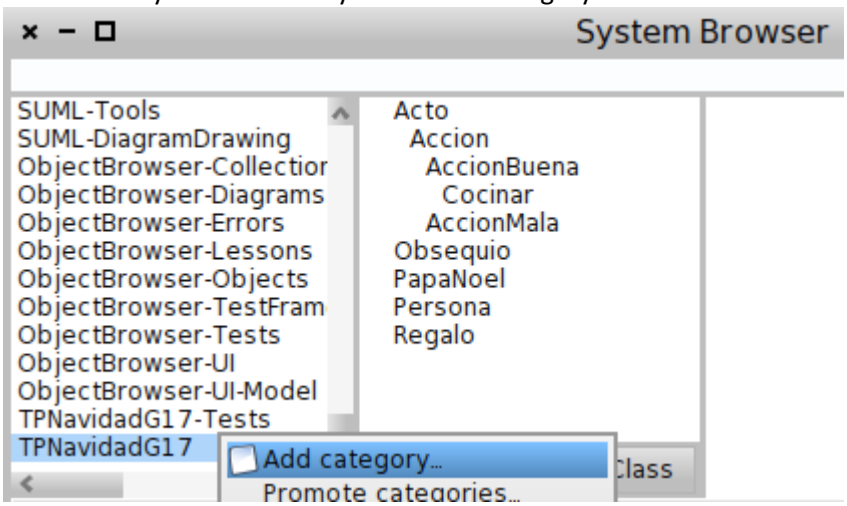
- el método cantidadBondadManoConMoco dio una falla (está en naranja)
- mientras que el método cantidadMaldadManoConMoco dio un error (se ve en el código que quisimos enviar el mensaje + 2 al objeto nil)

La solución es entrar en el test y verificar qué condición no se cumplió: esto puede resultar difícil de ver, en ese caso pueden pasar al workspace nuevamente y enviar el mensaje que forma parte del assert:/deny: para ver qué devuelve ese mensaje.

Test unitarios en System Browser (Class Browser)

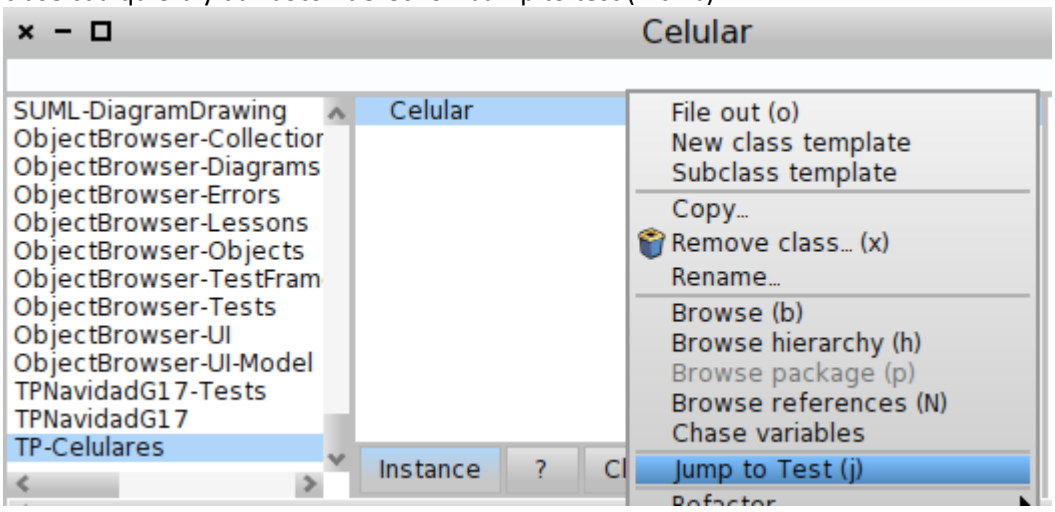
Definición de una categoría y una clase de test

Abrimos el System Browser y desde una Category definimos una nueva categoría de test:

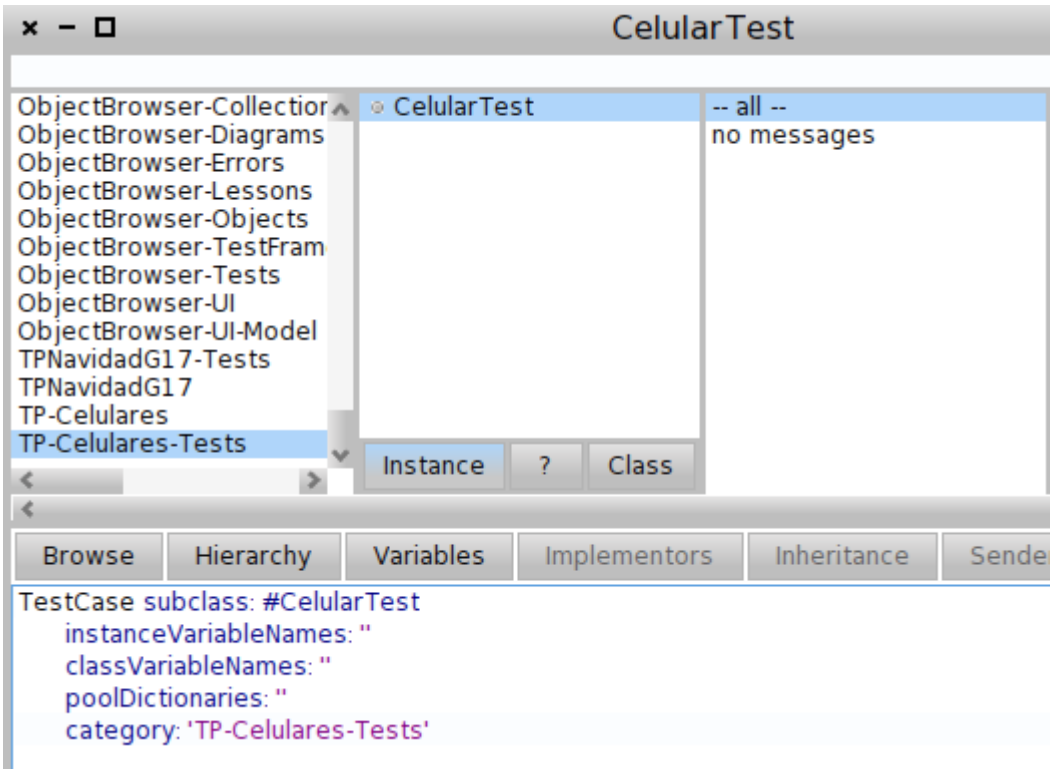


El nombre del package debería llevar el sufijo -Tests.
Después hay que crear una clase *XXXX*Test que herede de *TestCase*, donde *XXXX* es el nombre de la clase a testear.

Otra opción (más piola) es: si ya estuvieron definiendo clases del TP, pueden pararse sobre una clase cualquiera y dar botón derecho > Jump to test (Alt + J):

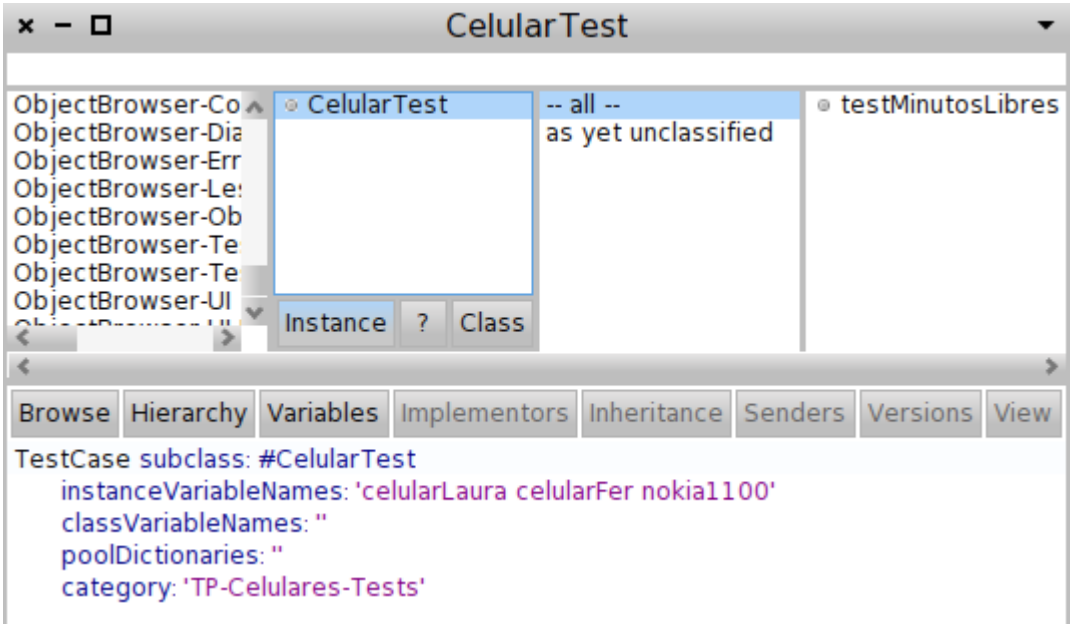


Esto les crea automáticamente una category con el sufijo Test, y una clase CelularTest que hereda de *TestCase*:



Juego de datos

- Se define cada objeto que participa del juego de datos como variable de instancia de la clase test (CelularTest en nuestro ejemplo, *XXXX*Test en el caso general).



- Se inicializan los objetos en un método setUp de la clase de test, al igual que en el caso del Object Browser.

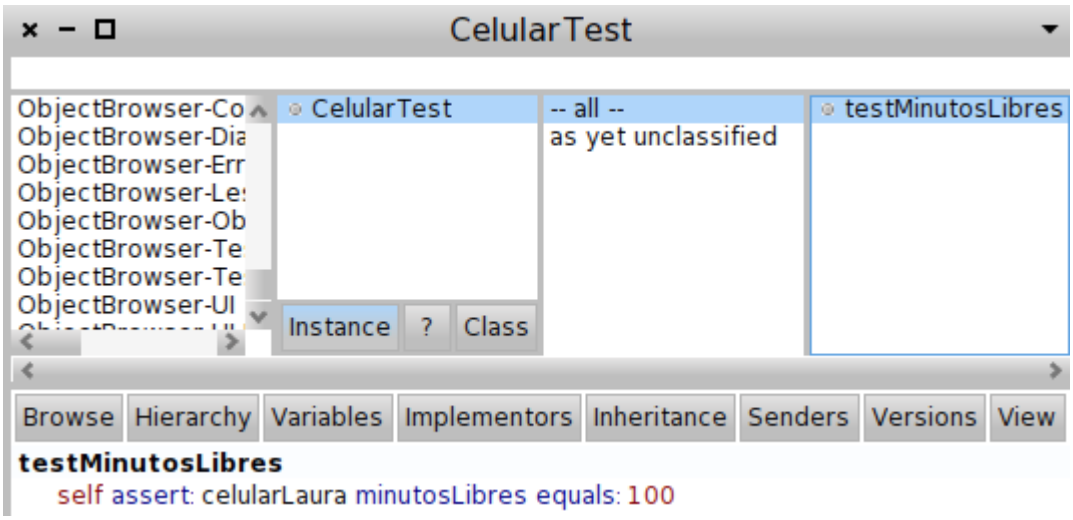
Tests

1. Como dijimos antes, las clases Tests deben heredar de TestCase (esto es muy importante para que entiendan los mensajes assert:, deny:, etc.)
2. Cada método test debe comenzar con el prefijo *test*. Ejemplo: si estamos probando que Pedrito aprobó dos materias, un nombre adecuado podría ser testPedritoAprobo2Materias. Lo importante es que comience con test para que la herramienta de testing (que se llama SUnit) lo considere.

Valen los mismos mensajes que en el object browser, que son entre otros:

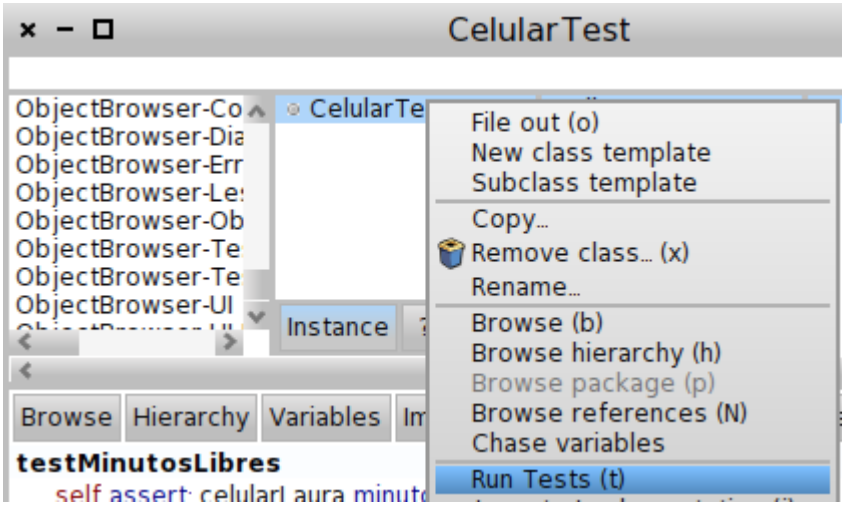
- assert: *unBoolean*
- deny: *unBoolean*
- assert: *mensaje a un objeto* equals: *un valor*
- assert: *unBoolean* description: *mensaje de error a mostrar en el test en caso de falla*

El código que va en el test es similar al que escribimos en el object browser, solo que ahora el objeto es una variable de instancia de la clase Test en lugar de ser una "referencia" del Object Browser:

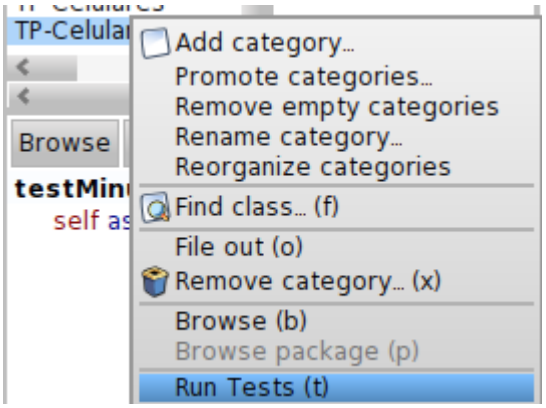


Ejecución de los tests

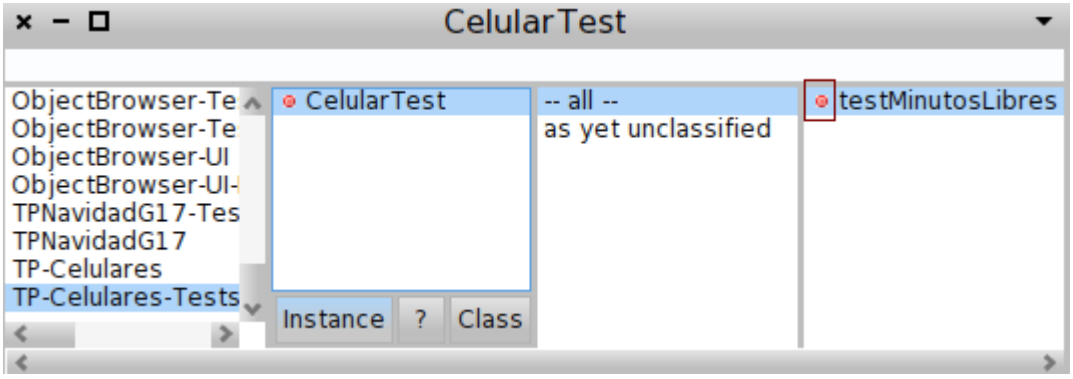
Prevalece el mismo concepto de ok, falla y error. Para ejecutar los tests nos podemos parar sobre una clase, botón derecho y Run Tests (Alt + T):



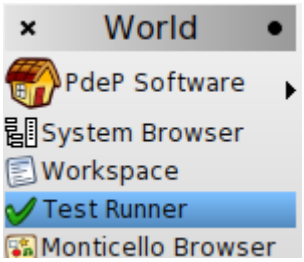
O bien podemos ejecutar todos los test de una category, con un botón derecho sobre la category > Run Tests (Alt + T):



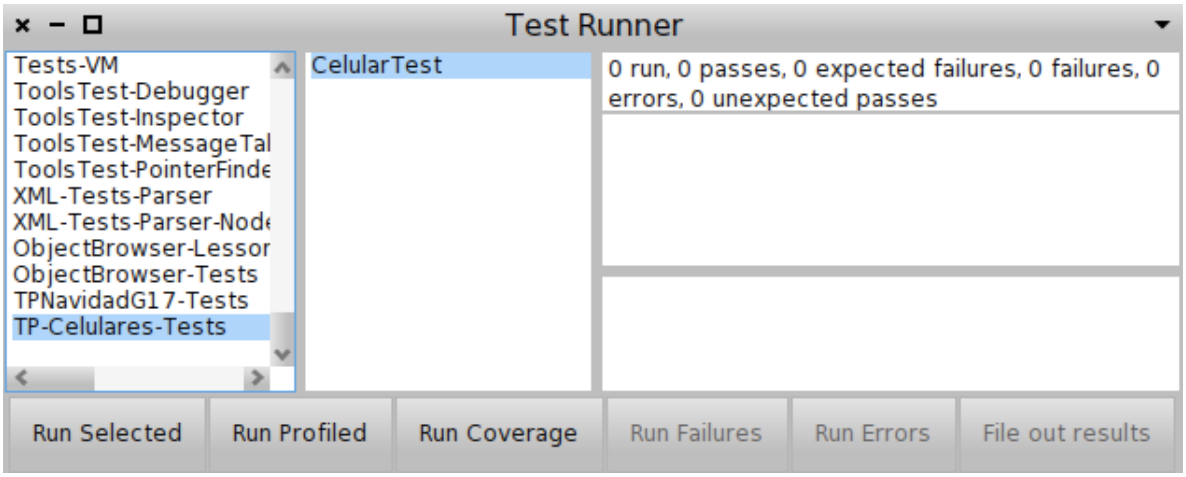
A diferencia del Object Browser, aquí saltan los errores en el Stack Trace y podemos debuggear el estado del test. Al finalizar vemos en rojo y amarillo/naranja los métodos con errores y fallas:



También podemos ir a la herramienta Test Runner, haciendo click izquierdo sobre el fondo del Pharo/Croquet:

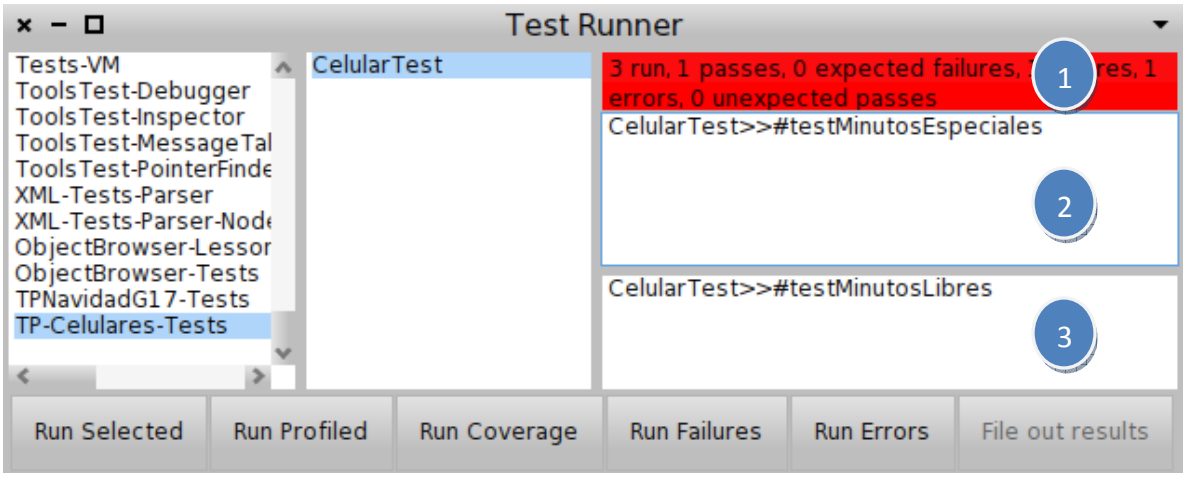


Desde allí, nos aparecen las categories que tengan asociadas clases tests. Seleccionamos con un click nuestra Category y podemos disparar la ejecución de los tests mediante un “Run Selected”:

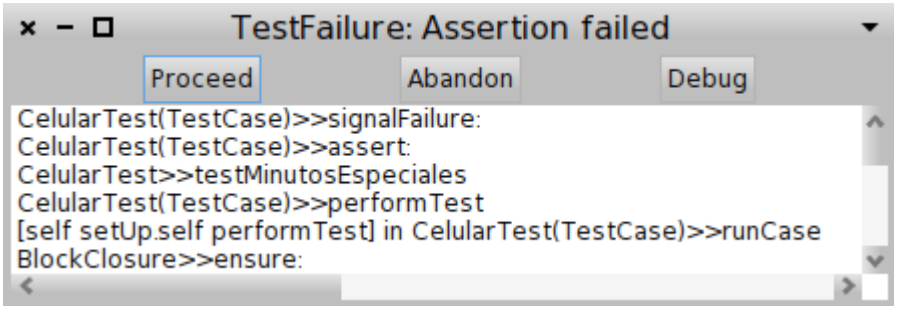


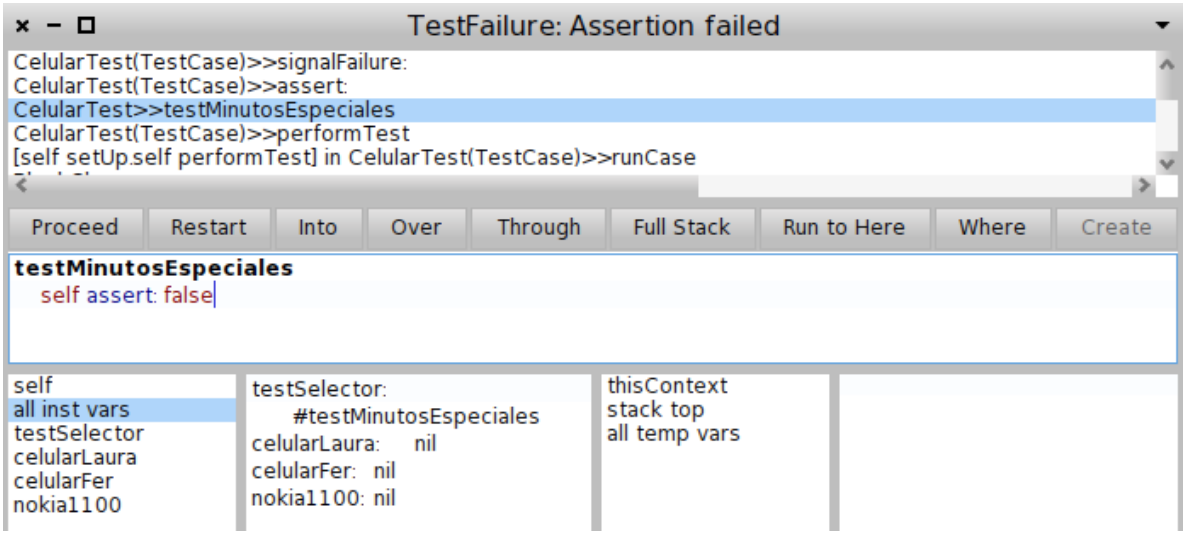
Al presionar “Run Selected” vemos el extremo derecho de la pantalla dividido en tres paneles:

- en el primer panel (1) vemos el semáforo, en rojo nos indica que hubo algún test con error, en amarillo algún test dio falla (ninguno dio error) y en verde todos los tests funcionaron ok (siempre se muestra el peor resultado)
- en el segundo panel (2): los tests con fallas
- en el tercer panel (3): los tests con errores



Tanto en el segundo panel como en el tercero, al dar click sobre cada test que falló o dio error veremos cuál fue el problema asociado (y podremos debuggear su estado):


TestFailure: Assertion failed
Proceed
Abandon
Debug
CelularTest(TestCase)>>signalFailure:
CelularTest(TestCase)>>assert:
CelularTest>>testMinutosEspeciales
CelularTest(TestCase)>>performTest
[self setUp.self performTest] in CelularTest(TestCase)>>runCase
BlockClosure>>ensure:


TestFailure: Assertion failed
CelularTest(TestCase)>>signalFailure:
CelularTest(TestCase)>>assert:
CelularTest>>testMinutosEspeciales
CelularTest(TestCase)>>performTest
[self setUp.self performTest] in CelularTest(TestCase)>>runCase
Proceed
Restart
Into
Over
Through
Full Stack
Run to Here
Where
Create
testMinutosEspeciales
self assert: false
self
all inst vars
testSelector
celularLaura
celularFer
nokia1100
testSelector:
#testMinutosEspeciales
celularLaura: nil
celularFer: nil
nokia1100: nil
thisContext
stack top
all temp vars