Técnicas de Diseño
75.10

Trabajo Práctico 2.1:

Framework para tests unitarios (2ª parte)

Grupo 15:
Emiliano Suárez
Federico Rodriguez
Leandro Miguenz

Introducción

El framework FWK provee un conjunto de funciones que permiten al usuario escribir pruebas unitarias y obtener sus resultados de forma automática. Está construido sin utilizar reflexión, RTTI ni anotaciones, por lo que, a diferencia de otros frameworks de pruebas, no se requiere seguir convenciones de nombres en los tests ni marcarlos de alguna forma, sino que el usuario debe explicitar qué métodos en particular desea correr como pruebas, invocándolos dentro de un método en particular e incluyéndolos en un test suite. A continuación se detalla el modo de uso del framework.

Modo de uso

Escritura del test unitario

En esta versión del framework se agregó la posibilidad de identificar a los tests por un nombre, así como seleccionarlos a partir de aplicar expresiones regulares sobre esos nombres. Para aprovechar está funcionalidad, el usuario debería crear una clase por cada test unitario que escriba, ya que el nombre del test es un atributo de instancia de dicha clase.

En primer lugar, cada una de esas clases debe extender la clase Test del framework. Esto le brinda acceso a los métodos de comparación de valores (Assert...) allí definidos, que le brindan al framework la información sobre los resultados de las pruebas.

Concretamente, la clase de prueba del usuario debe:
- importar la clase cuyo comportamiento se quiere probar
- importar y extender la clase Test
- tener un constructor que invoque al de la clase madre, un método de prueba, y redefinir el método runTest para que invoque a dicho método de prueba.

Un test básico tendría la siguiente estructura:

```
import main.java.ClaseASerTesteada;
import main.java.Test;

public class UnTestEnParticular extends Test {

	public UnTestEnParticular(String newName) {
		super(newName);
	}

	@Override
	public void runTest() {
		unTestEnParticular();
	}

	private void unTestEnParticular() {
		assertEquals("unTestEnParticular", valor1, valor2);
	}
}
```

`unTestEnParticular()` es el método definido por el usuario dentro del cual se escribe la prueba propiamente dicha. `runTest()` debe ser redefinido para contener la llamada a dicho método. Finalmente, el constructor se define llamando al constructor de la clase madre, de manera tal que se le pueda asignar un nombre al test.

Nótese que, si bien en el ejemplo se sigue una convención en pos de la claridad, los nombres de la clase, del método que contiene la prueba, y el nombre que se asignará en el constructor, son totalmente independientes entre sí.

Los métodos assertEquals/assertNotEquals comparan los valores recibidos en el segundo y tercer parámetro, y en base al resultado el framework determina si el test pasó con éxito o no. Para poder identificar el test que pasó o falló, el usuario debería pasar un mensaje adecuado como primer parámetro del assert (el nombre del test por ejemplo).

- Set Up y Tear Down:

En caso de necesitarlo, el usuario puede redefinir los métodos `setUp()` y `tearDown()`. Estos serán ejecutados respectivamente antes y después de correr el test, de manera automática.

El SetUp es de la forma

```
@Override
public void setUp() {
      //creación de variables objetos
      //utilizados por el test
}
```

El TearDown es análogo.

Organización y ejecución de los tests

Además de una clase por test, el usuario debe crear una clase que contenga el método main necesario para correr el proyecto, y en la cual se podrán agrupar los tests de manera apropiada.

- Test suite:

Para poder ser ejecutados, los tests unitarios deben instanciarse e incluirse dentro de una test suite (ya sea que esta contenga un sólo test unitario, o varios) previamente creada.

Ejemplo:

```
TestSuite suiteA = new TestSuite("Suite A");

Test1 test1 = new Test1("test nro. 1");
Test2 test2 = new Test2("test nro. 2");

suiteA.addTest(test1);
suiteA.addTest(test2);
```

Las suites puedem utilizarse para agrupar tests bajo algún criterio, y para separar aquellos que deseen correrse en forma separada.

Asímismo, es posible agrupar varias suites dentro de otra, sin restricción en los niveles de anidamiento:

```
suiteTodo.addTest(suiteA);
suiteTodo.addTest(suiteB);
```

- Ejecución:

Para ejecutar los tests, es necesario instanciar al test runner e invocar su método startTesting, pasándo como parámetro la suite que se desea correr. Todo esto debe realizarse desde el método main.

Ejemplo:

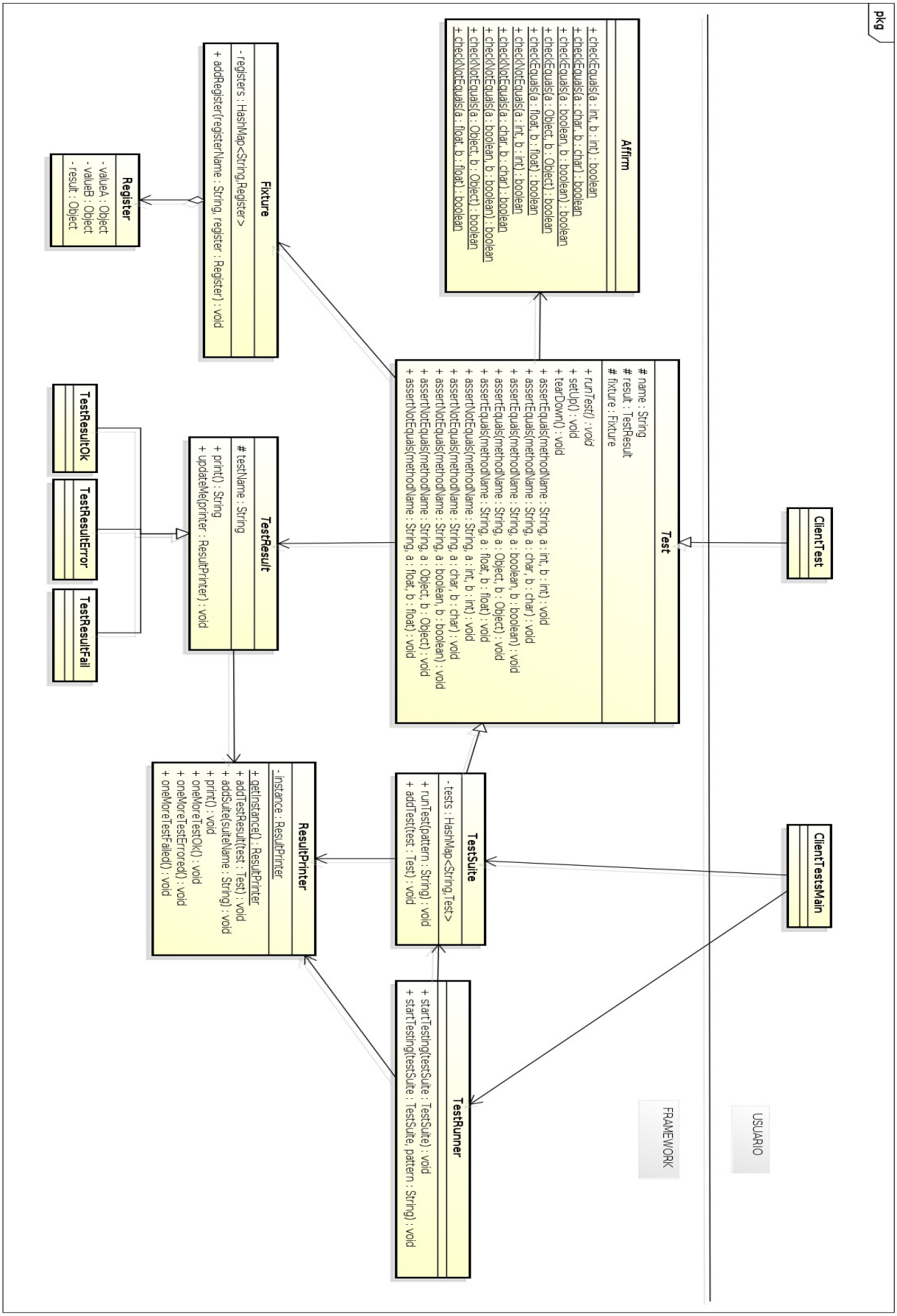
```
public static void main(String[] args) {

	//se instancian los tests
	Test1 test1 = new Test1("test nro. 1");
	Test2 test2 = new Test2("test nro. 2");
	Test3 test3 = new Test3("test nro. 3");
	Test4 test4 = new Test4("test nro. 4");

	//se crean las suites
	TestSuite suiteA = new TestSuite("Suite A");
	TestSuite suiteB = new TestSuite("Suite B");
	TestSuite suiteTodo = new TestSuite("Suite Todo");

	//test1 y 2 agredados a suiteA
	suiteA.addTest(test1);
	suiteA.addTest(test2);

	//test3 y 4 agregados a suiteB
	suiteB.addTest(test3);
	suiteB.addTest(test4);

	//suites A y B agregadas a suiteTodo
	suiteTodo.addTest(suiteA);
	suiteTodo.addTest(suiteB);

	//instanciación del testRunner y ejecución de los tests
	TestRunner testRunner = new TestRunner();
	testRunner.startTesting(testSuiteAll);

}
```

Al llamar al test runner también podría utilizarse el método

```
public void startTesting(TestSuite testSuite, String pattern)
```

que ejecutará sólo aquellos tests cuyo nombre coincida con la expresión regular que se haya pasado por parámetro.

Estructura del framework



El principal punto de conexión entre el código cliente y el framework es la clase Test, de la cual las clases con las pruebas del usuario deben heredar para poder usar los métodos de comparación definidos en dicha clase y redefinir setUp() y tearDown().

Además, la clase con el main desde donde se ejecutan todos los tests hace uso de TestSuite y TestRunner.

Los métodos básicos de comparación están definidos en la clase Affirm. Lo que hace la clase Test es tomar esos métodos y envolverlos en otros, que además de comparar valores generan un objeto del tipo TestResult. Se distingue entre tres resultados posibles (una clase por cada uno): Ok, Fail y Error (tests que fallaron no por un resultado incorrecto en el assert, sino porque se lanzó una excepción dentro del mismo).

La clase ResultPrinter es la encargada de generar el reporte de la corrida. Se implementó utilizando el patrón singleton para mejorar la solidez de los reportes presentados, al asegurar que toda la información de las corridas esté concentrada bajo el dominio de un mismo objeto.

ResultPrinter utiliza double dispatch para llevar la cuenta de los TestResult de cada tipo que son ejecutados: para cada testResult llama al método updateMe, que a su vez llama al método de ResultPrinter que se encarga de incrementar el contador correcto.

El anidamiento de test suites y su ejecución es posible debido a que TestSuite hereda de Test y redefine su método runTest, con lo cual se aprovecha el polimorfismo para diferenciar entre un test unitario y una suite.

Finalmente, la clase TestRunner elige qué tests ejecutar (en caso de que se pida ejecutar por una expresión regular) e invoca a ResultPrinter para mostrar el reporte de la corrida.