Técnicas de Diseño
75.10

Trabajo Práctico 2.2:

Framework para tests unitarios (3ª 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

Esta versión del framework permite identificar a los tests por un nombre y tagearlos, así como la posibilidad de seleccionar aquellos que se deseen ejecutar en base a los tags a los que pertenecen, a partir de aplicar expresiones regulares sobre sus nombres o sobre los nombres de las suites, o utilizando alguna combinación de esos criterios.

Para aprovechar está funcionalidad, el usuario debería crear una clase por cada test unitario que escriba, ya que el nombre del test, así como su lista de tags, son atributos de instancia de dicha clase.

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

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

Un test básico tendría la siguiente estructura:

```
import main.java.ClaseASerTesteada;
import main.java.TestMethod;

public class UnTestEnParticular extends TestMethod {

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

	@Override
	public void run() {
		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. `run()` 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. En el constructor el usuario también puede ya establecer una lista de tags para el test. Si bien también pueden agregarse los tags desde el main del proyecto de pruebas (se explica más adelante).

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 instanciar y 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 pueden utilizarse para agrupar tests bajo algún criterio, y para separar aquellos que deseen correrse en forma separada.

Asimismo, 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 desarrollado por el usuario del framework.

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);

}
```

- Criterios especiales de ejecución:

Al correr el ejemplo anterior, se ejecutarían todos los tests dentro de testSuiteAll. Es posible definir, para cada suite, criterios para que sólo se ejecuten los tests que cumplan ciertas condiciones. Las condiciones están relacionadas con los tags a los que está asociado cada test, y la aplicación de expresiones regulares sobre los nombres de test y los nombres de suites.
Estos son los métodos de TestSuite que permiten establecer dichos criterios:

```
public void setToRunByTestName(String testRegex)
public void setToRunByTag(String tag)
public void setToRunByTags(TagList tags)
public void setToRunByTagsAndTestName(TagList tags, String testRegex)
public void setToRunByTagsOrTestName(TagList tags, String testRegex)
public void setToRunByTagsAndSuiteName(TagList tags, String suiteRegex)
public void setToRunByTagsOrSuiteName(TagList tags, String suiteRegex)
public void setToRunByTagsAndTestNameAndSuiteName(TagList tags, String
testRegex, String suiteRegex)
public void setToRunByTagsAndTestNameOrSuiteName(TagList tags, String testRegex,
String suiteRegex)
public void setToRunByTagsOrTestNameAndSuiteName(TagList tags, String testRegex,
String suiteRegex)
public void setToRunByTagsOrTestNameOrSuiteName(TagList tags, String testRegex,
String suiteRegex)
```

Cada suite puede utilizar su propio criterio, pero en el caso de suites anidadas, siempre prevalece el criterio de la suite más interna.

En el siguiente fragmento de código de ejemplo se eligen ejecutar para una suite sólo los tests que tengan el tag “A” y cuyo nombre contenga la palabra “Objects”:

```
//tests instanciados
TestEqualsObjectTest testEqualsObjectTest = new TestEqualsObjectTest("Test two objects are equal using the Test class' methods");
TestEqualsIntTest testEqualsIntTest = new TestEqualsIntTest("Test two integers are equal using the Test class' methods");


//se taggean los tests
testEqualsObjectTest.addTag("A");
testEqualsIntTest.addTag("A");

//se establece el criterio para la suite
testSuiteTest.setToRunByTagsAndTestName(new TagList("A"), "^.*Object.*$");
```

Si luego se agregan los dos tests a la suite y se la ejecuta, solo se va a ejecutar el test `testEqualsObjectTest`.

- Visualización de los resultados:

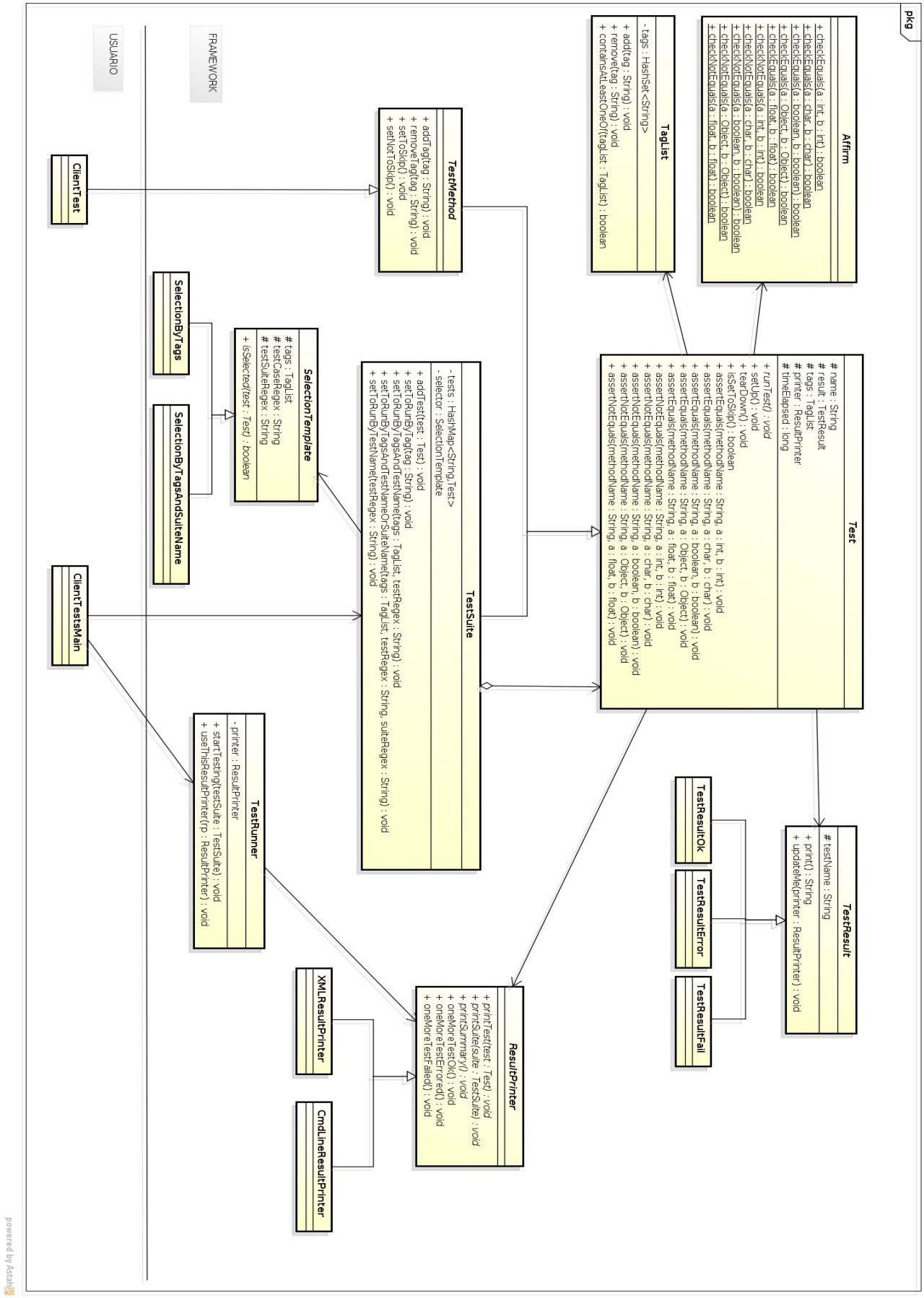
Es posible obtener el reporte con los resultados de la corrida de distintas maneras. Para que el reporte utilice un impresor de resultados distinto se puede invocar al método

```
public void useThisResultPrinter(ResultPrinter rp)
```

pasándole el ResultPrinter que se desee utilizar. Si no se invoca este método, el TestRunner utiliza por defecto un CmdLineResultPrinter. La única otra opción al momento de escribir este documento es el XMLResultPrinter, que genera un documento XML.

De esta manera permitimos la extensibilidad en caso de que se creen nuevos ResultPrinter, pero al mismo tiempo mateniendo el código de TestRunner cerrado ante cambios.

Estructura del framework



El principal punto de conexión entre el código cliente y el framework es la clase TestMethod, de la cual las clases con las pruebas del usuario deben heredar para poder usar los métodos de comparación 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 una abstracción para generar el reporte de la corrida. Se implementó utilizando el patrón “Strategy”. Las estrategias definidas consisten en mostrar el reporte por línea de comandos, o generando un archivo XML. De esta manera permitimos la extensibilidad en caso de que se creen nuevos ResultPrinter, y al mismo tiempo se mateniendo el código de TestRunner cerrado ante cambios.

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 la utilización del patrón “Composite”, por el cual TestSuite hereda de Test y redefine su método runTest, al igual que TestMethod, con lo cual se aprovecha el polimorfismo para diferenciar entre un test unitario y una suite al momento de la ejecución.

Para poder elegir diferentes criterios de ejecución de tests se utilizó el patrón “Template Method”: el usuario setea el criterio de selección de tests, que se usa dentro del método runTest para discernir entre tests que deben ser corridos y tests que no. TestSuite tiene un atributo “selector”, del tipo “SelectionTemplate”, clase abstracta de la cual heredan todos los criterios de selección.

Finalmente, la clase TestRunner elige qué tests ejecutar e invoca a su resultPrinter para mostrar el reporte de la corrida.