Planificación y Pruebas de Sistemas Software

Curso 2015-16

Sesión 3: Implementación: Drivers

Pruebas unitarias

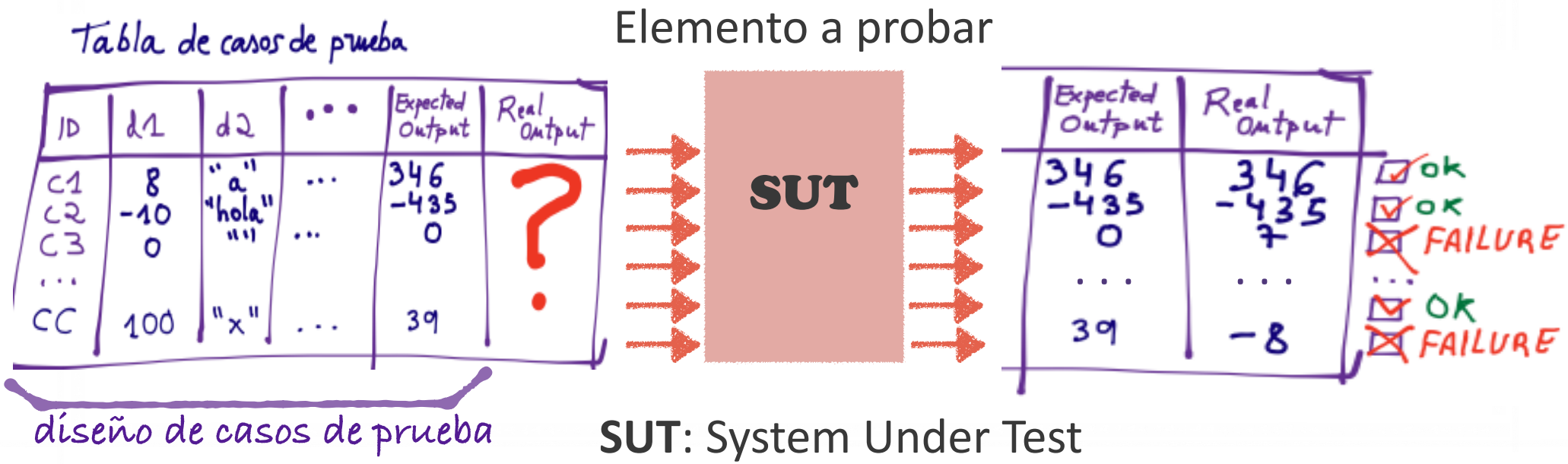
Automatización de las pruebas: Drivers

Implementación de drivers: JUnit

María Isabel Alfonso Galipienso
Universidad de Alicante eli@ua.es

Automatización de los casos de prueba

- La ejecución de un caso de prueba requiere: *implementar un DRIVER!!!*
  - Establecer las precondiciones (asunciones sobre lo que es cierto antes de ejecutar el elemento)
  - Proporcionar los datos de entrada + resultado esperado
  - Observar la salida (resultado real)
  - Comparar el resultado esperado con el resultado real
  - Emitir un informe (para poner de manifiesto si hemos detectado un fallo o no)



SUT: System Under Test

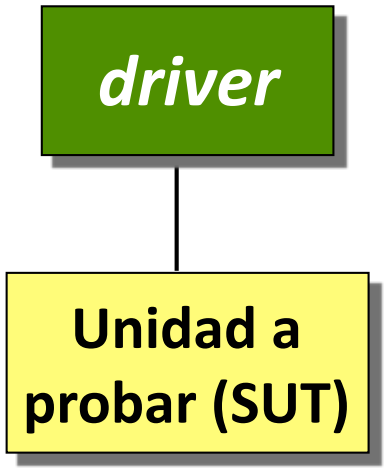
Pruebas unitarias

- Constituyen el primer nivel de pruebas. El objetivo principal es **aislar** la ejecución de una unidad de programa
  - Sintácticamente, una unidad de programa es una "pieza" de código, como por ejemplo una función o método de una clase, que es invocada desde fuera de la unidad y puede invocar a otras unidades de programa
  - Una unidad de programa implementa una función bien definida, y proporciona un nivel de abstracción para la implementación de funcionalidades de mayor nivel
- Las pruebas unitarias son realizadas por los propios programadores. Éstos necesitan VERIFICAR si el código funciona correctamente (tal y como se esperaba)
- Hasta que el programador no implemente la unidad y esté completamente probada, el código fuente de una unidad no se pone a disposición del resto de miembros del grupo (normalmente a través de un sistema de control de versiones)
- Pueden realizarse pruebas unitarias de forma estática y dinámica

Pruebas de unidad dinámicas: *drivers*

- Requieren ejecutar la unidad (SUT: System Under Test) de forma AISLADA para poder detectar defectos en dicha unidad
  - La ejecución de la unidad tiene lugar "fuera" del entorno de producción. Por lo tanto tendremos que añadir código adicional para emular dicho entorno

```
//algoritmo de un driver
informe driver() {
  d= prepara_datos_entrada();
  esperado= resultado_esperado;
  //llamamos a SUT
  real= SUT(d);
  //comparamos el resultado
  //real con el esperado
  c= (esperado == real);
  informe= prepara_informe(c);
  return informe;
}
```



driver : conductor de la prueba. Contiene el código necesario para ejecutar el test sobre SUT

SUT : Código que queremos probar.

En este tema vamos a ver cómo implementar drivers con JUnit para realizar pruebas unitarias dinámicas !!!!

JUnit

- JUnit es un API de java que permite implementar los drivers para ejecutar los casos de prueba sobre componentes (SUT) de forma automática
- JUnit proporciona:
  - Clases y métodos para implementar las pruebas (*tests*)
    - Envío de entradas a SUT
    - Respuestas esperadas de la ejecución de SUT
    - Comprobación de si el resultado real coincide o no con el esperado
    - Generación de informes con el resultado de la ejecución de los tests
  - Clases para ejecutar los tests (*test runners*)
    - Mecanismo para "leer" las pruebas y ejecutarlas
    - Registra y sigue la pista de los resultados de los tests
- JUnit se puede utilizar para implementar drivers de pruebas unitarias y también de integración
  - En una prueba unitaria estamos interesados en probar un único SUT
  - En una prueba de integración estamos probando varias unidades

¿Cómo identificamos un driver con JUnit?

- Un driver automatiza la ejecución de un caso de prueba.
  - Junit utiliza anotaciones para facilitar la programación de dichos conductores de pruebas.
  - JUnit identifica un test (uno por cada caso de prueba) como un MÉTODO público, sin parámetros, que devuelve void, y está anotado con @Test (@org.junit.Test)

```
package mismo.paquete.claseAprobar;

public class TianguloTest {

    @Test
    public void testQueNoHaceNada() {

    }

}
```

Los tests deben estar agrupados lógicamente con el SUT correspondiente, pero estarán físicamente separados

La clase de pruebas tendrá el mismo nombre que la clase que contiene el SUT precedida (o antecedida) por "Test"

El nombre de cada método anotado con @Test estará precedido (o antecedido) por "test"

Cada método anotado con @Test implementará un driver para un ÚNICO caso de prueba !!!!

Implementación de un driver(I)

Cualquier driver sigue el mismo esquema:

```
//algoritmo de un driver
informe driver() {
  d= prepara_datos_entrada();
  esperado= resultado_esperado;
  real= SUT(d); //llamamos a SUT
  c= (esperado == real); //comparamos el resultado real con el esperado
  informe= prepara_informe(c);
  return informe;
}
```

para implementar un driver necesitamos diseñar los casos de prueba previamente

preparar datos de entrada

```
public class TrianguloTest {
  int a,b,c;
  String resultadoReal, resultadoEsperado;
  Triangulo tri= new Triangulo();

  @Test
  public void testTipo_trianguloC1() {
    a = 1;
    b = 1;
    c = 1;
    resultadoEsperado = "Equilatero";
    resultadoReal = tri.tipo_triangulo(a,b,c);
    assertEquals(resultadoEsperado, resultadoReal);
  }
}
```

Implementación de un driver(II)

- El código de pruebas está físicamente separado del código fuente del SUT

SUT

```
package ppss;

public class Triangulo {

  public String tipo_triangulo
              (int a, int b, int c) {
       ...
  }

}
```

/src/main/java
/target/classes

DRIVER

```
package ppss;

public class TrianguloTest {
   int a,b,c;
   String real, esperado;
   Triangulo tri= new Triangulo();

   @Test
   public void testTipo_trianguloC1() {
     a = 1;
     b = 1;
     c = 1;
     resultadoEsperado = "Equilatero";
     resultadoReal = tri.tipo_triangulo(a,b,c);
     assertEquals(esperado, real);
   }
}
```

/src/test/java
/target/test-classes
/target/surefire-reports

ubicación física de SUT y DRIVER si usamos Maven

Sentencias Assert (org.junit.Assert)

http://junit.org/javadoc/latest/org/junit/Assert.html

- Junit proporciona sentencias Assert para determinar el resultado de las pruebas y emitir el informe correspondiente
- Son métodos estáticos, cuyas principales características son:
  - Se utilizan para comparar el resultado esperado con el resultado real
  - Pueden incluir un primer parámetro opcional con un mensaje que se mostrará en el informe si la aserción no se cumple
  - Todos los métodos "assert" generan una excepción de tipo AssertionError si el resultado esperado no coincide con el real
  - Los métodos "assert" devuelven "void" si el valor esperado coincide con el real
  - El orden de los parámetros para los métodos assert... es:
    - [mensaje opcional] resultado ESPERADO, resultado REAL
  - Para el subconjunto de métodos assertThat..., el orden es:
    - [mensaje opcional] resultado REAL, resultado ESPERADO
    - además, estos métodos utilizan en los parámetros objetos de tipo Matcher

Ejemplos de métodos assert...

Lista completa de métodos que se pueden utilizar:
http://junit.sourceforge.net/javadoc/org/junit/Assert.html

Statement (Son métodos ESTÁTICOS)	**Description**
fail(String)	Fails a test with the given message
assertTrue(condition); assertFalse(condition)	Asserts that a condition is true/false **
assertEquals([String message], expected, actual)	Asserts that two values are equal. **Note**: for arrays the reference is checked not the content of the arrays **
assertEquals([String message], expected, actual, tolerance)	Usage for float and double; the tolerance are the number of decimals which must be the same **
assertNull([message], object)	Checks if the object is null **
assertNotNull([message], object)	Checks if the object is not null **
assertSame([String], expected, actual)	Checks if both variables refer to the same object **
assertNotSame([String], expected, actual)	Checks that both variables refer not to the same object **
assertTrue([message], boolean condition)	Checks if the boolean condition is true. **
try {a.shouldThroughException(); fail("Failed")} catch (RuntimeException e) {asserttrue(true);}	Alternative way for checking for exceptions

** if it isn't an **AssertionError** is thrown

Método assertThat

assertThat([value], [matcher statement]);

- Permite flexibilizar la forma de expresar las aserciones con JUnit
  - El orden de los parámetros es:
    - Valor real
    - Un objeto Matcher (org.hamcrest.CoreMatchers)
  - Las sentencias matcher pueden ser:
    - negadas (not(s))
    - combinadas (either(s).or(t), both())
    - mapeadas a una colección de elementos: (everyItem(s))
- Ejemplos:
  - assertThat(x, is(3)); assertThat(x, is(not(4)));
  - assertThat(resultado, either(containsString("color")).or(containsString("colour")));
  - assertThat("albumen", both(containsString("a")).and(containsString("b")));
  - assertThat(myList, hasItem("3"));
  - assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));
  - assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }), everyItem(containsString("n")));
  - Los métodos de los objetos Matcher son métodos estáticos:
    - import static org.hamcrest.CoreMatchers.not;

JUnit y Maven

- Para poder utilizar JUnit en la implementación de los tests, necesitamos incluir la librería correspondiente en el fichero de configuración del proyecto:

pom.xml

```
<project ...>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

</project>
```

```
//código fuente de los tests en /src/test/java
package ppss;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
...
```

- El fichero junit-4.12.jar contiene la implementación (ficheros .class) de las clases del API junit que importamos para poder compilar los tests.
- El valor "test" de la etiqueta <scope> indica que la librería sólo se utiliza desde el código fuente de los tests (/src/test/java)

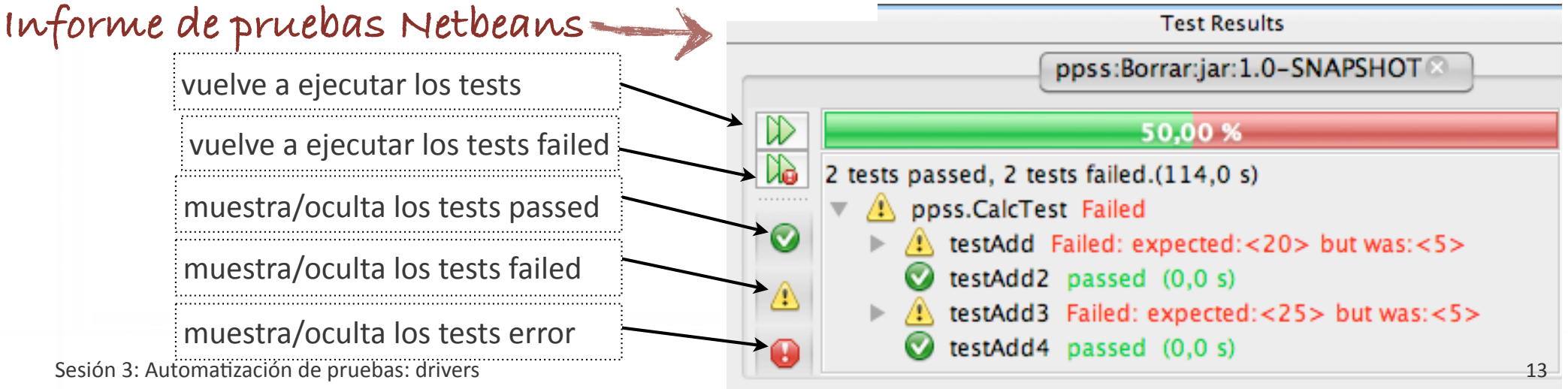
Ejecución de tests e informes JUnit

- JUnit proporciona unas clases especiales denominadas "runners" que se utilizan para ejecutar los tests
  - Un "JUnit runner" es una clase que hereda de la clase abstracta Runner:
    - Podemos indicar la clase runner que queremos usar con la anotación @RunWith
    - Por defecto se utiliza la clase BlockJunit4-ClassRunner
- Los runners se encargan también de proporcionar la información para elaborar el informe de las pruebas:
  - **Pass**: el resultado del método Assert es true
  - **Fail**: el método Assert lanza una excepción de tipo AssertionError
  - **Error**: se genera cualquier otra excepción durante la ejecución del test
  - Un test puede ignorarse (no ser ejecutado) mediante la anotación @Ignore

```
Tests run: 5, Failures: 3, Errors: 1, Skipped: 0
```

Informe de pruebas Maven

Informe de pruebas Netbeans



Fixtures

- El término fixture hace referencia a un cierto estado de un conjunto de objetos usados en nuestros tests
  - El propósito de la fixture de un test es asegurar un entorno bien conocido y fijo en el que los tests se ejecuten, de forma que los resultados sean repetibles. Ejemplos de fixtures:
    - Preparación de datos de entrada (inicialización de valores)
    - Cargar un conjunto de datos en una base de datos antes de hacer las pruebas
- JUnit proporciona anotaciones de forma que cada test pueda presentar una fixture antes o después de cada test, o bien podamos establecer una única fixture para todos los métodos anotados con test de una clase
  - @BeforeClass, @AfterClass: ejecutan una única vez antes (después) de cualquier método anotado con test de una clase. Esta anotación se utiliza con un método estático, sin parámetros, y que devuelve void
  - @Before, @After: se ejecutan antes (después) de cada método anotado con test de una clase. Se puede anotar cualquier método público, sin parámetros, y que devuelva void

Ejemplo de uso de fixtures

```
public class OutputTest {
  private File output;

  @Before public void createOutputFile() {
     output = new File(...);
  }
  @After public void deleteOutputFile() {
     output.delete();
  }
  @BeforeClass public static void initialState() {
     //initial code
  }
  @AfterClass public static void finalState() {
     //final code
  }
  @Test public void test1WithFile() {
     // code for test case objective
  }
  @Test public void test2WithFile() {
     // code for test case objective
  }
}
```

Orden de ejecución:

1. initialState()
2. createOutputFile()
3. test1WithFile()
4. deleteOutputFile()
5. createOutputFile()
6. test2WithFile()
7. deleteOutputFile()
8. finalState()

Asumiendo que test1WithFile() se ejecute antes que test2WithFile!!!

Pruebas de excepciones

- Si queremos probar si nuestro SUT lanza una determinada excepción, usaremos el parámetro "expected" en la anotación @Test

```
@Test(expected=ExceptedTypeOfException.class)
public void testException()  {
     exceptionCausingMethod();
}
```

la ejecución de este método debe lanzar la excepción

Si no se lanza ninguna excepción, o bien si tiene lugar una excepción no esperada, el test fallará

- Si queremos probar el mensaje de la excepción, o limitar el ámbito en el que se espera dicha excepción, la capturaremos y utilizaremos la aserción fail( ) si no se lanza la excepción

```
@Test public void testException() {
  try {
       exceptionCausingMethod();

       // If this point is reached, the expected
       // exception was not thrown

       fail("Exception should have occurred");
  } catch ( ExceptedTypeOfException exc ) {
      String expected = "A suitable error message";
      String actual = exc.getMessage();
      Assert.assertEquals( expected, actual );
  }
}
```

Maven y pruebas unitarias

- El proceso de construcción de Maven y las pruebas unitarias: ¿qué secuencia de acciones son necesarias para ejecutar pruebas unitarias con Maven?
  - Fase compile: se compilan los fuentes del proyecto (/src/main/java)
    - goal **compiler:compile**
    - artefactos generados en /target/classes
  - Fase test-compile: se compilan los tests unitarios (/src/test/java)
    - goal **compiler:test-compile**
    - artefactos generados en /target/test-classes
  - Fase test: se ejecutan los tests unitarios
    - goal **surefire:test**
    - artefactos generados en /target/surefire-reports
      - Los informes se generan en formato *.txt y *.xml
    - La goal surefire:test detiene la construcción del proyecto si algún test tiene como resultado "failure"
- mvn test
  - Ejecuta los tests unitarios: métodos anotados con @Test de las clases **/Test*.java, **/*Test.java, o **/*TestCase.java

Default lifecycle

validate
initialize
generate-sources
process-sources
generate-resources
process-resources
compile
process-classes
generate-test-sources
process-test-sources
generate-test-resource
process-test-resources
test-compile
process-test-classes
test
prepare-package
package
pre-integration-test
integration-test
post-integration-test
verify
install
deploy

Organización de los tests: Category

- Las "Categorías" permiten hacer agrupaciones de casos de prueba, utilizando la anotación @Category
  - Para definir las categorías podemos utilizar tanto clases como interfaces (normalmente se utilizan interfaces)
  - Podemos anotar como categorías tanto clases como métodos
- Ejemplos de definiciones de Categorías:
  - para identificar tests que solo funcionan para diferentes navegadores (Firefox, InternetExplorer, AnyBrowser...)

```
public interface Firefox{}
public interface InternetExplorer{}
public interface AnyBrowser{}
           extends Firefox, InternetExplorer;
```

  - para discriminar tests unitarios de tests de integración

```
public interface SlowTests {}
public interface IntegrationTests
            extends SlowTests {}
public interface PerformanceTests
            extends SlowTests {}
public interface UnitTests {}
```

CATEGORÍA

```
public interface IntegrationTests {}

@Category(IntegrationTests.class)
public class AccountIntegrationTest {

  @Test
  public void thisTestWillTakeSomeTime() {...}

  @Test
  public void thisTestWillTakeEvenLonger() {...}
}

public class OtherIntegrationTest {

  @Test
  @Category(IntegrationTests.class)
  public void thisTestIsImportant() {
    ...}

  @Test
  public void thisTestIsCritical() {...}
}
```

anotamos una clase

anotamos un método

Ejecución de categorías con Maven

- Podemos ejecutar los tests unitarios en función de las categorías a las que pertenezcan configurando convenientemente el plugin surefire
  - Por defecto, la goal surefire:test ejecuta **todos** los tests
  - Para ejecutar sólo ciertas categorías de tests, tenemos que configurar el parámetro "groups" de dicho plugin en el pom.xml

```
<plugins>
  [...]
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.19.1</version>
    <configuration>
       <groups>com.mycompany.SlowTests</groups>
    </configuration>
  </plugin>
  [...]
</plugins>
```

  - También podemos configurar el parámetro "groups" desde línea de comando:
    - mvn test -Dgroups="com.mycompany.SlowTests"

Tests con parámetros

- La implementación de los drivers para automatizar las pruebas siempre sigue el mismo algoritmo:

```
public class TrianguloTest {
   ...
    @Test
    public void testTipo_trianguloC1() {
       a = 1; b = 1;  c = 1;
       resultadoEsperado = "Equilatero";
       resultadoReal = tri.tipo_triangulo(a,b,c);
       assertEquals(resultadoEsperado, resultadoReal);
    }

    @Test
    public void testTipo_trianguloC2() {
       a = 1;  b = 1; c = 11;
       resultadoEsperado = "No es un triangulo";
       resultadoReal = tri.tipo_triangulo(a,b,c);
       assertEquals(resultadoEsperado, resultadoReal);
    }

    @Test
    public void testTipo_trianguloC3() {
       a = 1;  b = 2; c = 0;
       resultadoEsperado = "Valor c fuera del rango permitido";
       resultadoReal = tri.tipo_triangulo(a,b,c);
       assertEquals(resultadoEsperado, resultadoReal);
    }

    ...
}
```

Si queremos añadir un nuevo test, podemos utilizar uno cualquiera de ellos y cambiar los valores de entrada y del resultado esperado

Supongamos que necesitamos añadir un nuevo parámetro al método tipo_triangulo()...

tendríamos que modificar TODOS los tests!!!

Tests con parámetros. Implementación

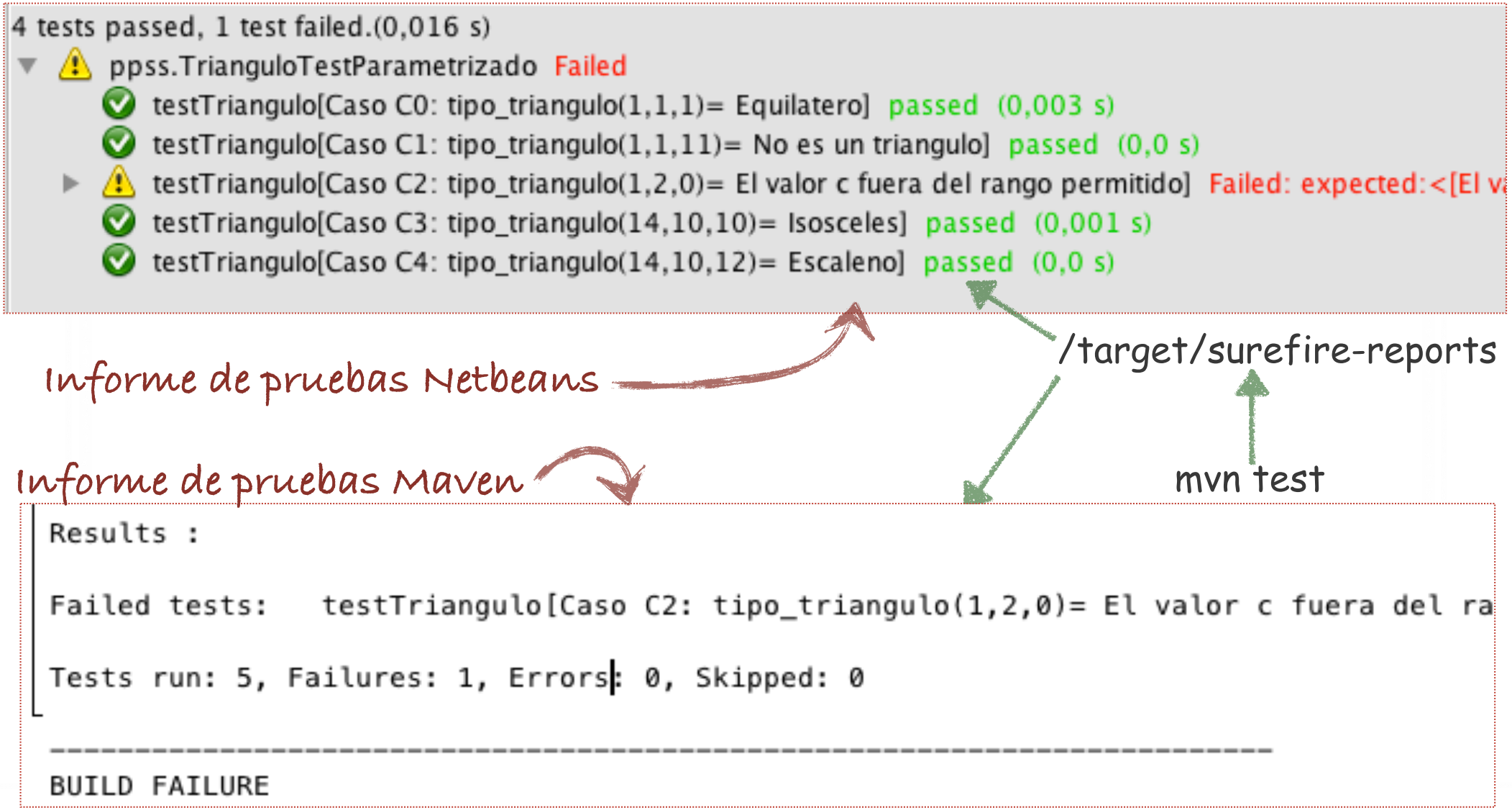
- Usaremos las anotaciones @RunWith y @Parameters para implementar tests parametrizados
- Cuando se ejecute una clase que contenga tests parametrizados se crearán instancias para cada método anotado con @Test con los parámetros necesarios de forma automática

```
import org.junit.Assert;  import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

import java.util.Arrays; import java.util.Collection

@RunWith(Parameterized.class)
public class TrianguloTestParametrizado {

  @Parameterized.Parameters(name =
                    "Caso C{index}: tipo_triangulo({0},{1},{2})= {3}")
  public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][]{
           {1, 1, 1, "Equilatero"}, //C0
           {1, 1, 11, "No es un triangulo"}, //C1
           {1, 2, 0, "El valor c fuera del rango permitido"}, //C2
           {14, 10, 10, "Isosceles"}, //C3
           {14, 10, 12, "Escaleno"} //C4
        });
    }

  private String esperado;
  private int lado1, lado2, lado3;
  private Triangulo tri = new Triangulo();

  public TrianguloTestParametrizado(int lado1, int lado2, int lado3,
                                    String esperado) {
     this.lado1 = lado1;
     this.lado2 = lado2;
     this.lado3 = lado3;
     this.esperado = esperado;
  }

  @Test
  public void testTriangulo() {
     Assert.assertEquals(esperado, tri.tipo_triangulo(lado1, lado2, lado3));
  }
}
```

podemos asociar nombres en el informe generado

Tests con parámetros. Ejecución

- Si indicamos un nombre en la anotación @Parameters, se mostrará dicho nombre en el informe de ejecución de los tests

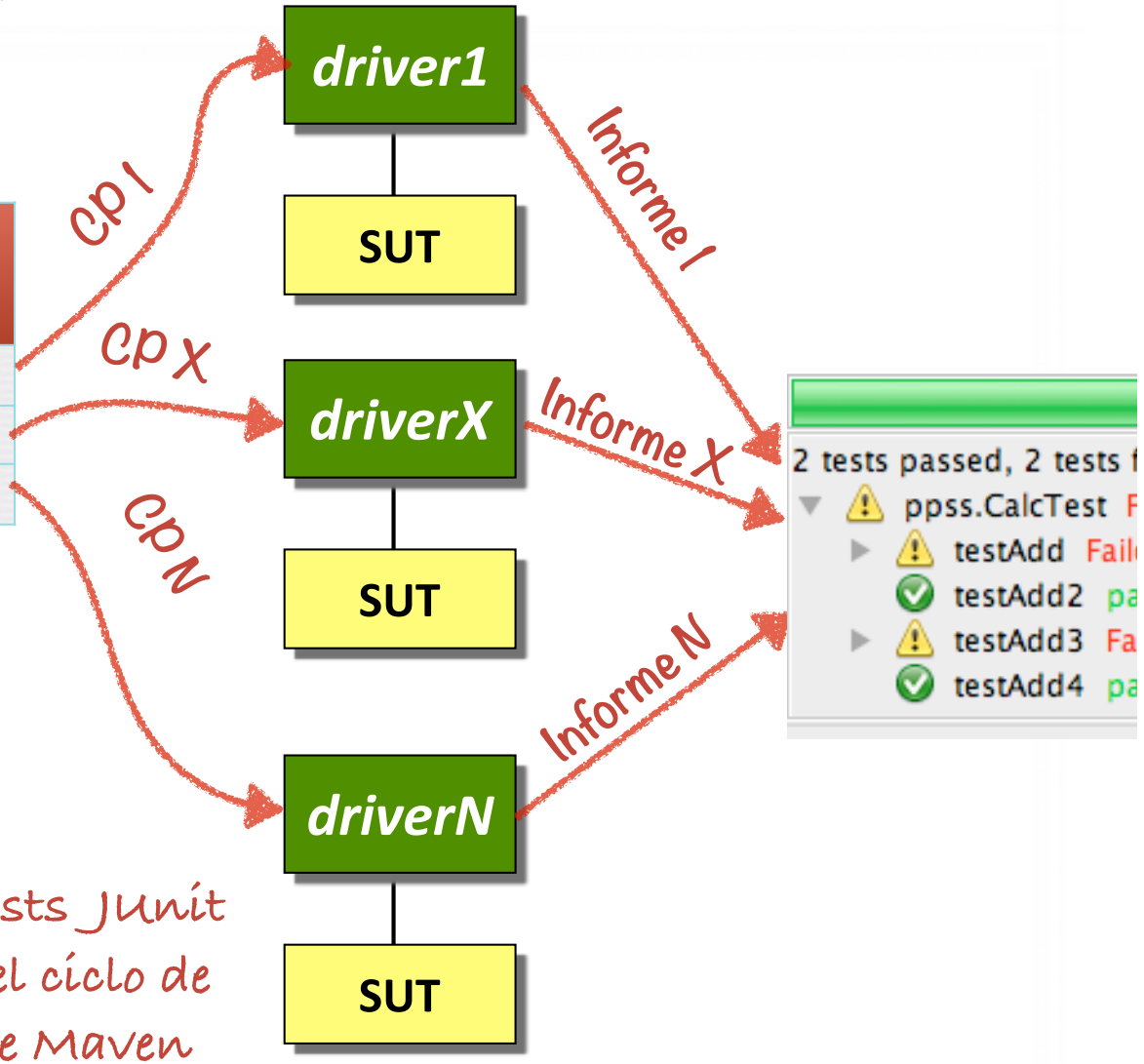


Y ahora vamos al laboratorio…

Vamos a automatizar el diseño de casos de prueba que hemos obtenido en prácticas anteriores

tabla de casos de prueba

Dato Entrada 1	Dato Entrada 2	Dato Entrada k	Resultado Esperado	Resultado Real
d1=…	d2=…	dk=…	r1	
d1=…	d2=…	dk=	rM	



Implementaremos los drivers utilizando el API JUnit

Ejecutaremos los tests JUnit integrándolos en el ciclo de vida por defecto de Maven

Referencias bibliográficas

- Software testing and quality assurance. Kshirasagar Naik & Priyadarshi Tripathy. Wiley. 2008
  - Capítulo 3: Unit Testing
- JUnit (http://junit.org)
  - Assertions
  - Exception Testing
  - Matchers and assertThat
  - Ignoring Tests
  - Parameterized Tests
  - Test Fixtures
  - Categories