

Validación y Verificación de Software

Aspectos básicos de testing sistemático:

Testing “ad hoc” | Testing sistemático |
Testing unitario | Unidad y Suite de tests |
Frameworks xUnit |
Fixtures compartidos e independencia |
Tests parametrizados por datos

Testing “ad hoc” (no sistemático)

Todo programador está habituado al testing.

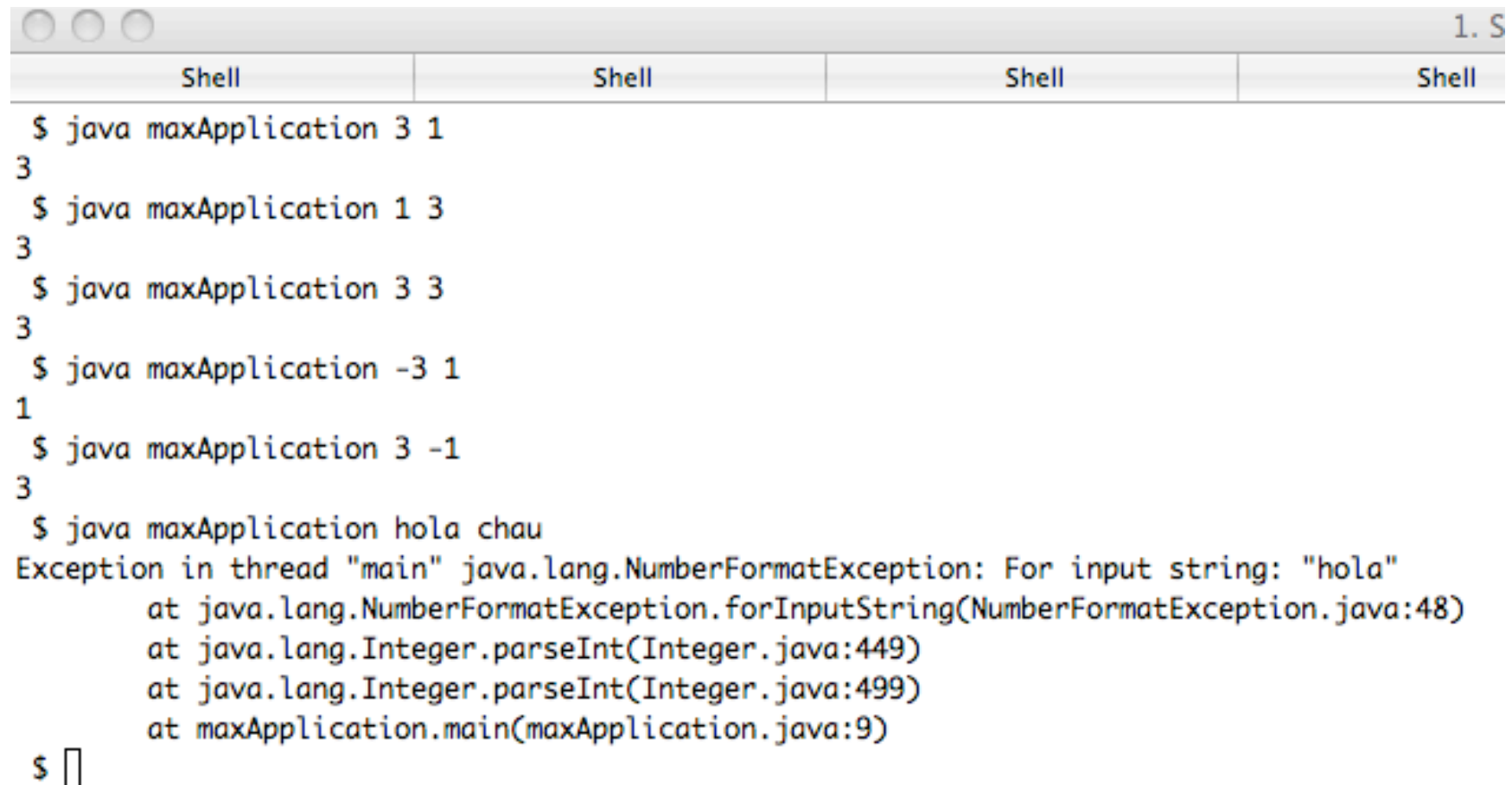
En muchos casos, la forma en la que hacemos testing es “ad hoc”, es decir, no sistemática.

Ej: supongamos que queremos testear la siguiente función:

```
public class maxApplication {  
  
    public static void main(String[] args) {  
        if (args.length != 2) {  
            System.out.println("uso: maxApplication <int> <int>");  
        }  
        else {  
            int a = Integer.parseInt(args[0]);  
            int b = Integer.parseInt(args[1]);  
            if (a>b) {  
                System.out.println(a);  
            }  
            else {  
                System.out.println(b);  
            }  
        }  
    }  
}
```

Testing “ad hoc” (cont.)

En este caso, podemos testear la aplicación directamente desde la línea de comandos:



```
1. S
Shell Shell Shell Shell
$ java maxApplication 3 1
3
$ java maxApplication 1 3
3
$ java maxApplication 3 3
3
$ java maxApplication -3 1
1
$ java maxApplication 3 -1
3
$ java maxApplication hola chau
Exception in thread "main" java.lang.NumberFormatException: For input string: "hola"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:449)
    at java.lang.Integer.parseInt(Integer.java:499)
    at maxApplication.main(maxApplication.java:9)
$
```

Testing “ad hoc” (cont.)

Ej: En caso que lo que tengamos no sea una aplicación, sino una funcionalidad interna:

```
public class SampleStaticRoutines {  
    public static int max(int a, int b) {  
        if (a>b) {  
            return a;  
        }  
        else {  
            return b;  
        }  
    }  
}
```

el testing “ad hoc” se vuelve más difícil: tenemos que programar un arnés para la rutina (i.e., un “main” que la invoque).

Dificultades del Testing “ad hoc”

Es simple para testear aplicaciones,
pero no almacena los tests para pruebas futuras.

Requiere la construcción manual de “arneses” para la prueba de funcionalidades “internas” (no directamente invocables desde la interfaz de la aplicación).

Requiere la inspección humana de la salida de los tests:
se decide manualmente si el test pasó o no.

Sistematización del Testing

Las librerías de apoyo al testing ayudan a sistematizar parte de las tareas manuales mencionadas:

- permiten almacenar los tests como “scripts”,
- definen un esquema estándar para scripts de tests,
- permiten definir la salida esperada como parte del script,
- construyen automáticamente arneses para la ejecución de los tests.

Testing unitario

Es una metodología para testear unidades individuales de código (SUT: Software under test), preferentemente de forma aislada de sus dependencias

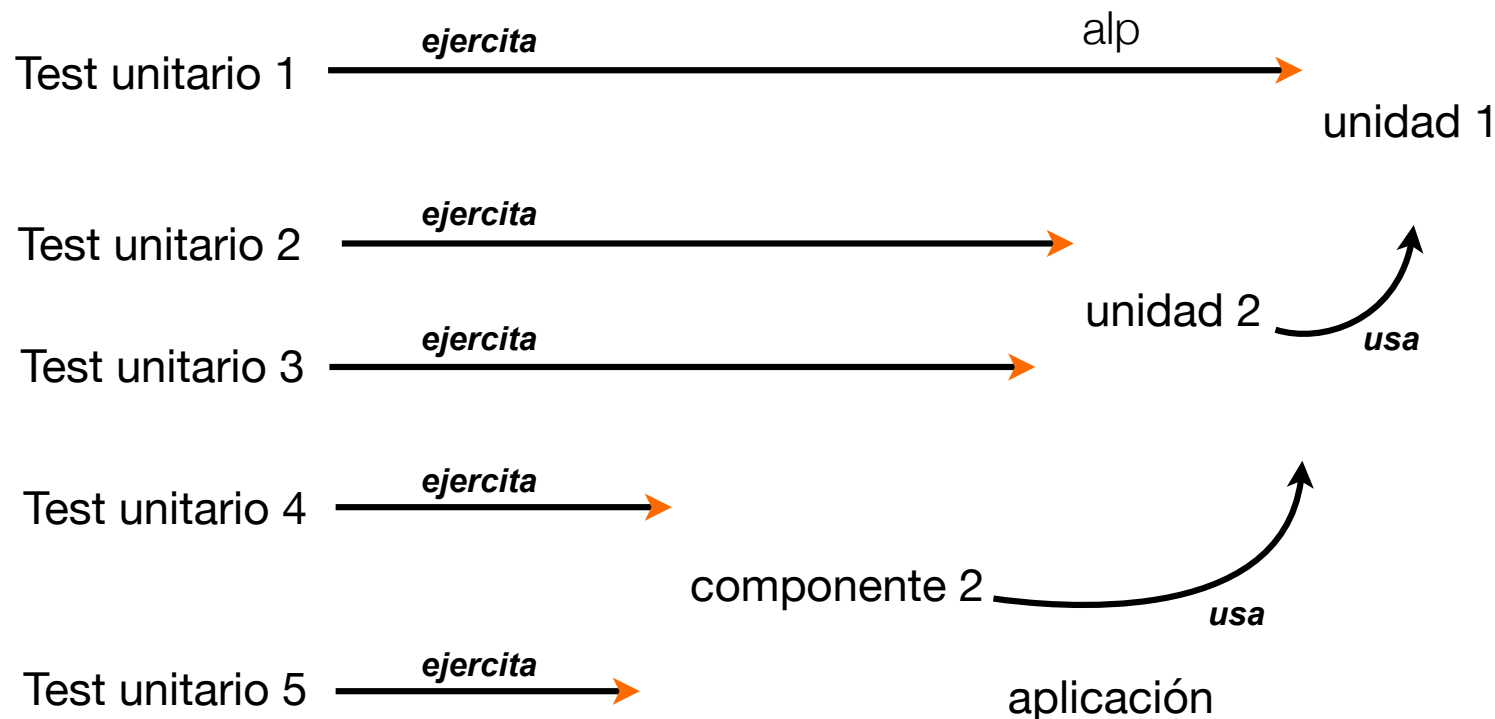
Usualmente los tests son creados por los propios programadores durante el proceso de codificación.

El uso de esta metodología:

- facilita los cambios,
- sirve como documentación de código,
- contribuye al diseño del sistema.

Testing unitario (cont.)

Normalmente las unidades son las partes mas pequeñas de un sistema: funciones o procedimientos, clases, etc.
pero la granularidad es variable.



Testing unitario con JUnit

JUnit es una librería de apoyo al testing unitario para Java:

- Define la estructura básica de un test.

- Permite organizar tests en suites.

- Ofrece entornos para la ejecución de tests y suites.

- Reporta información detallada sobre las pruebas, en especial sobre las pruebas que fallan.

Estructura de un test JUnit

arrange: se preparan los datos para alimentar al programa.

```
import org.junit.*;
import static org.junit.Assert.*;

public class SampleStaticRoutinesTest {
```

act: se ejecuta el programa con los datos contruidos.

```
    @Test
    public void testMax01() {
        int a = 1;
        int b = 3;
        int res = SampleStaticRoutines.max(a,b);
        assertTrue(res == 3);
    }
}
```

assert: se evalúa si los resultados obtenidos se corresponden con lo esperado.

Algunas aserciones JUnit

`assertTrue(<expr>)`: verifica que `<expr>` evalúe a true.

`assertFalse(<expr>)`: verifica que `<expr>` evalúe a false.

`assertEquals(<expr1>, <expr2>)`: verifica que `<expr1>` y `<expr2>` evalúen al mismo valor.

`assertArrayEquals(<array1>, <array2>)`: verifica que `<array1>` y `<array2>` sean iguales, elemento a elemento.

`assertNotNull(<object>)`: verifica que `<object>` no sea null.

`assertNull(<object>)`: verifica que `<object>` sea null.

`assertNotSame(<object1>, <object2>)`: verifica que `<object1>` y `<object2>` no sean el mismo objeto.

Matchers hamcrest y assertThat

Las versiones más recientes de JUnit proveen una nueva forma de escribir aserciones para los test

Único método para describir resultados esperados:
`assertThat(<object>, <matcher>)`

Un “matcher” hamcrest es una descripción declarativa de una característica deseable de un objeto. Por ejemplo:

```
assertThat(list, containsInAnyOrder(2, 4, 5));
```

```
assertThat(list, everyItem(greaterThan(1)));
```

Matchers hamcrest: Ejemplos

El String str es igual a “b” o a “c”:

```
assertThat(str, anyOf(equalTo("b"), equalTo("c")));
```

“HoLa” es igual a “hola” si se ignoran mayúsculas:

```
assertThat("HoLa", equalToIgnoringCase("hola"));
```

El objeto obj no es null:

```
assertThat(obj, not(isNull));
```

El objeto obj es de tipo Student (o cualquier subtipo de Student):

```
assertThat(obj, instanceOf(Student.class));
```

El arreglo arr contiene el elemento 5:

```
assertThat(arr, hasItemInArray(5));
```

Matchers hamcrest: Reporte de errores

Falla en `assertTrue(s.equals("b") || s.equals("c"))`:

```
java.lang.AssertionError
    at org.junit.Assert.fail(Assert.java:86)
    at org.junit.Assert.assertTrue(Assert.java:41)
    at org.junit.Assert.assertTrue(Assert.java:52)
    at prueba.TestHamcrest.test1(TestHamcrest.java:46)
```

Falla en `assertThat(s, anyOf(equalTo("b"), equalTo("c")))`:

```
java.lang.AssertionError:
    Expected: ("b" or "c")
    but: was "a"
    at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)
    at org.junit.Assert.assertThat(Assert.java:956)
```

Características deseables de los tests unitarios

Los tests unitarios (SUT es un procedimiento) deben ser:

- Automáticos

- Fácilmente ejecutables por cualquier persona involucrada en el proyecto

- Ejecutables con sólo “presionar un botón”

- Repetibles (con resultados consistentes)

- Fáciles de implementar

- Independientes entre si

- Rápidos y eficientes en el uso de recursos

Test “negativos” en JUnit

Los tests negativos son aquellos en los ejercitamos el software bajo test en circunstancias fuera de las esperadas por el mismo (i.e., que violan la “precondición” del software bajo test).

En muchos casos, se espera que los tests negativos lancen excepciones. En JUnit esto se debe indicar explícitamente:

```
@Test(expected= IllegalArgumentException.class)
public void testMax11() {
    Comparable<Integer> a = new Integer(2);
    Comparable<String> b = new String("hola");
    Comparable res = SampleStaticRoutines.max(a, b);
}
```


Timeouts

JUnit podemos exigir que un test finalice antes de un tiempo predeterminado:

```
@Test(timeout=1000)
public void test1() {
    long res = SampleStaticRoutines.fibonacciEfi(45);
    assertEquals(res, 1134903170);
}
```

Suites de tests

Cuando el SUT está compuesto por varias clases, o simplemente varios métodos de una clase, resulta claro que debemos organizar los tests.

Los tests se organizan en test suites.

- una test suite es simplemente un conjunto de tests.

En JUnit, la organización es un poco más elaborada:

- un Test Case es un conjunto de tests

- una Test Suite es una colección de test cases

Suites de tests en JUnit

Además, podemos agrupar suites para construir nuevas suites:

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
    UnitTests1.class,
    UnitTests2.class,
    ModuleTests.class,
    IntegrationTests.class
})
```

Suites de tests y datos compartidos

En muchos casos, los tests de una suite comparten los datos que manipulan.

En estos contextos, suele ser útil organizar los tests definiendo procesos comunes de fixture (o arrangement, o set up) para todos los tests.

Similarmente, se pueden definir procesos comunes de destrucción (o tear down), que se ejecuten luego de cada test.

Suites de test y datos compartidos: un ejemplo

Consideremos un ejemplo de testing de una clase Minefield, que representa el estado de un campo minado en el juego “Buscaminas”:

```
public class Minefield {  
    private Mine[][] field;  
  
    ...  
  
    public int minedNeighbours(int x, int y) {  
        ...  
    }  
}  
  
public class Mine {  
  
    private boolean isMined;  
    private boolean isMarked;  
    private boolean isOpened;  
  
    ...  
}
```

Para poder testear minedNeighbours, debemos crear el minefield, y ubicar minas en lugares específicos.

Suites de tests y datos compartidos: un ejemplo

El proceso setUp en el siguiente ejemplo crea el escenario adecuado para la ejecución de tests de minedNeighbours.

Se ejecuta antes de cada test.

```
@Test
public void testNoOfMinedNeighbours01() {
    int number = testingField.minedNeighbours(1, 0);
    org.junit.Assert.assertTrue(number == 1);
}

@Test
public void testNoOfMinedNeighbours02() {
    int number = testingField.minedNeighbours(7, 7);
    org.junit.Assert.assertTrue(number == 0);
}
```

```
public class MinefieldTest {

    private Minefield testingField;

    @Before
    public void setUp() throws Exception {
        if (testingField == null) {
            testingField = new Minefield();
        }
        for (int i=0; i<8; i++) {
            for (int j=0; j<8; j++) {
                testingField.removeMine(i, j);
                testingField.unmark(i, j);
                testingField.close(i, j);
            }
        }
        testingField.putMine(0, 0);
        testingField.putMine(3, 4);
        testingField.putMine(4, 3);
        testingField.putMine(2, 2);
        testingField.putMine(0, 7);
        testingField.putMine(7, 7);
        testingField.putMine(5, 1);
        testingField.putMine(4, 7);
    }

    ...
}
```

Sobre setUp y tearDown

Es importante que no haya dependencias entre tests diferentes.

No hay ninguna garantía sobre el orden en el que se ejecutan los tests

Las rutinas setUp (@Before) y tearDown (@After) ayudan a garantizar la ausencia de dependencias entre diferentes tests.

El método anotado con @Before se ejecuta antes de cada test de la suite.

El método anotado con @After se ejecuta después de cada test de la suite,

aún si el test lanza excepciones.

Fixtures globales

En JUnit, podemos inicializar (destruir) datos/recursos globales a los tests de una misma clase (suite) anotando un método con `@BeforeClass` (`@AfterClass`)

Para leer datos de configuración globales, inicializar recursos caros en tiempo de cómputo, etc.

Usar con cuidado para evitar comprometer la independencia de los tests

```
@BeforeClass
public static void setUpClass() {
    myExpensiveManagedResource = new ExpensiveManagedResource();
}

@AfterClass
public static void tearDownClass() throws IOException {
    myExpensiveManagedResource.close();
}
```


Tests “parametrizados” por los datos que manipulan

Ya hemos visto varios casos en los cuales varios tests poseen exactamente la misma estructura, pero difieren en los datos que se utilizan para realizar el arrange del test.

Para estos casos, JUnit ofrece una forma conveniente de organizar los tests, separando su estructura de los datos que manipulan.

La clave es:

- definir datos para la operación de los tests como atributos de clase,

- definir un generador de parámetros, que se usará para instanciar los datos para los tests.

largest tiene un error comienza de 1
para que fallen algunos tests

Test parametrizados: un ejemplo

Consideremos el método largest, que calcula el máximo de un arreglo. El siguiente test parametrizado lo prueba con varios datos diferentes:

```
@RunWith(Parameterized.class)
public class SampleStaticRoutinesLargestTest {
```

```
    private Integer [] array;
    private Integer res;
```

```
    public SampleStaticRoutinesLargestTest(Object [] array, Object res) {
        this.array = (Integer[]) array;
        this.res = (Integer) res;
    }
```

```
    @Parameters
    public static Collection<Object[]> firstValues() {
        return Arrays.asList(new Object[][] {
            {new Integer [] { 1,2,3 }, 3 },
            {new Integer [] { 2,1,3 }, 3 },
            {new Integer [] { 3,1,2 }, 3 },});
    }
```

```
    @Test
    public void testFirst() {
        int max = SampleStaticRoutines.largest((Integer[]) array);
        org.junit.Assert.assertTrue(res == max);
    }
}
```

Indica que la suite está formada por tests parametrizados.

Indica que éste es el método que produce los parámetros (se pasan al constructor).

Estructura genérica de los tests.

Teorías JUnit

Un test unitario captura el comportamiento del SUT para una entrada particular; una teoría captura algún aspecto del comportamiento del SUT para un número potencialmente infinito de entradas

En otras palabras, las teorías permiten describir propiedades que el código debe satisfacer.

Ej.: el siguiente test indica que el resultado de multiplicar U\$S 5 por 2 debe ser igual a U\$S 10

```
@Test
public void multiplyByAnInteger() {
    Dollar d = new Dollar(5);
    d.times(2);
    assertThat(d.getAmount(), is(10));
}
```

Teorías JUnit

La teoría a continuación describe la propiedad (de Dollar y sus métodos times, divideBy y getAmount):

“Para cualquier par de valores amount y m, si m es distinto de 0, multiplicar y dividir (amount) por m resulta en el valor original”

```
@Theory
public void multiplyIsInverseOfDivide(int amount, int m) {

    assumeThat(m, not(0));
    System.out.println(amount + "," + m);
    Dollar d = new Dollar(amount);
    d.times(m);
    d.divideBy(m);
    assertThat(d.getAmount(), is(amount));
}
```

Teorías JUnit

Usualmente una teoría está compuesta por tres partes:

Precondición: Descripción de las entradas para las cuales la teoría es válida

Se usa el método `assumeThat(<obj>, <matcher>)`

Parámetros que violan la precondición son ignorados

Pieza de código a testear (método, módulo, etc..)

Postcondición: Propiedad que se espera que el código satisfaga, para valores cualesquiera de los parámetros que cumplan con la precondición

Teorías: Parámetros y valores

Para correr una teoría debemos proveer valores para los parámetros, que JUnit utilizará para testear el SUT

```
@RunWith(Theories.class)
public class DollarTheoryTest {

    @DataPoint
    public static int one = 1;
    @DataPoint
    public static int two = 2;
    @DataPoints
    public static int[] many = {15, 24, 47};

    @Theory
    public void multiplyIsInverseOfDivide(int amount, int m) {
        assumeThat(m, not(0));
        Dollar d = new Dollar(amount);
        d.times(m);
        d.divideBy(m);
        assertThat(d.getAmount(), is(amount));
    }
}
```

JUnit correrá un test para cada combinación posible de valores que coincidan con el tipo de los parámetros (en el ejemplo, pares de enteros tomados de {1, 2, 15, 24, 47})

Teorías: Parámetros y valores

```
@Theory
public void multiplyIsInverseOfDivide(@TestedOn(ints={0,5,10})int amount, @TestedOn(ints={1,3,10})int m)
    assumeThat(m, not(0));
    Dollar d =new Dollar(amount);
    d.times(m);
    d.divideBy(m);
    assertThat(d.getAmount(), is(amount));
}
```

Teorías JUnit: Generadores de parámetros definidos por el usuario

JUnit permite definir generadores de valores de parámetros personalizados para las teorías

Primero, debemos definir una interface con el nombre que queremos darle al generador (SimpleIntGen):

```
@Retention(RetentionPolicy.RUNTIME)
@ParametersSuppliedBy(SimpleIntGenSupplier.class)
public @interface SimpleIntGen {

}
```

La anotación @ParametersSuppliedBy indica cuál será la clase encargada de generar valores (SimpleIntGenSupplier.class)

Teorías JUnit: Generadores de parámetros definidos por el usuario

SimpleIntGenSupplier clase generadora, redefine getValueSources de ParameterSupplier

PotentialAssignment.forValue(<String>, <Object>) construye cada uno de los valores que retornará el generador (PotentialAssignment); su primer parámetro es una simple etiqueta, y el segundo es el valor a retornar

```
3
4 public class SimpleIntGenSupplier extends ParameterSupplier {
5
6     @Override
7     public List<PotentialAssignment> getValueSources(ParameterSignature sig) {
8         List<PotentialAssignment> values = new ArrayList<PotentialAssignment>();
9         Random rand = new Random();
10
11         for (int i=0; i<10; i++) {
12             int x = rand.nextInt();
13             values.add(PotentialAssignment.forValue(Integer.toString(x), x));
14         }
15         return values;
16     }
17 }
```

Teorías JUnit: Generadores de parámetros definidos por el usuario

Finalmente, debemos etiquetar los parámetros de una teoría con el nombre del generador:

```
@Theory
public void multiplyIsInverseOfDivide3(@SimpleIntGen int amount, @SimpleIntGen int m) {
    assumeThat(m, not(0));
    Dollar d = new Dollar(amount);
    d.times(m);
    d.divideBy(m);
    assertThat(d.getAmount(), is(amount));
}
}
```

En este ejemplo, JUnit generará 10 valores aleatorios para amount, 10 para m, e invocará a la teoría con todas las posibles combinaciones entre ellos

Teorías JUnit: Generadores de parámetros definidos por el usuario

Veamos el ejemplo completo:

```
@Retention(RetentionPolicy.RUNTIME)
@ParametersSuppliedBy(SimpleIntGenSupplier.class)
public @interface SimpleIntGen {

}
```

```
3
4 public class SimpleIntGenSupplier extends ParameterSupplier {
5
6     @Override
7     public List<PotentialAssignment> getValueSources(ParameterSignature sig) {
8         List<PotentialAssignment> values = new ArrayList<PotentialAssignment>();
9         Random rand = new Random();
10
11         for (int i=0; i<10; i++) {
```

```
    @Theory
    public void multiplyIsInverseOfDivide3(@SimpleIntGen int amount, @SimpleIntGen int m) { lg(x),x ));
        assumeThat(m, not(0));
        Dollar d =new Dollar(amount);
        d.times(m);
        d.divideBy(m);
        assertThat(d.getAmount(), is(amount));
    }
}
```

Teorías JUnit: Generadores de parámetros definidos por el usuario

Los generadores pueden tomar parámetros, para personalizar su funcionamiento al invocarlos con diferentes valores para los mismos

Para agregar parámetros se deben agregar métodos (sin implementación) a la interfaz que define el generador:

```
@Retention(RetentionPolicy.RUNTIME)
@ParametersSuppliedBy(SimpleIntBetweenSupplier.class)
public @interface SimpleIntBetween {
    int min();
    int max();
}
```

IntBetween generará enteros en el rango (min...max)

Teorías JUnit: Generadores de parámetros definidos por el usuario

Los parámetros del generador pueden accederse en el método `getValueSources` usando el objeto `sig`

El ejemplo produce todos los enteros entre `first()` y `last()`

```
5 public class SimpleIntBetweenSupplier extends ParameterSupplier {  
6  
7     @Override  
8     public List<PotentialAssignment> getValueSources(ParameterSignature sig) {  
9         List<PotentialAssignment> values = new ArrayList<PotentialAssignment>();  
10        SimpleIntBetween annotation = sig.getAnnotation(SimpleIntBetween.class);  
11        int min = annotation.min();  
12        int max = annotation.max();  
13        Random rand = new Random();  
14        for (int i=0; i<10; i++) {  
15            int x = rand.nextInt(max)+min;  
16            values.add(PotentialAssignment.forValue(Integer.toString(x),x ));  
17        }  
18        return values;  
19    }  
20 }
```

Teorías JUnit: Generadores de parámetros definidos por el usuario

Finalmente, la invocación al generador @IntBetween requiere de valores para first y last, como se muestra a continuación:

```
@Theory
public void multiplyIsInverseOfDivide4(@SimpleIntBetween(min=1, max=50) int amount,
```

Teorías JUnit: Generadores de parámetros definidos por el usuario

Ejemplo completo:

```
@Retention(RetentionPolicy.RUNTIME)
@ParametersSuppliedBy(SimpleIntBetweenSupplier.class)
public @interface SimpleIntBetween {
    int min();
    int max();
}

public class SimpleIntBetweenSupplier extends ParameterSupplier {

    @Override
    public List<PotentialAssignment> getValueSources(ParameterSignature sig) {
        List<PotentialAssignment> values = new ArrayList<PotentialAssignment>();
        SimpleIntBetween annotation = sig.getAnnotation(SimpleIntBetween.class);
        int min = annotation.min();
        int max = annotation.max();
        Random rand = new Random();
        for (int i=0; i<10; i++) {
            int x = rand.nextInt(max)+min;
            values.add(PotentialAssignment.forValue(Integer.toString(x),x ));
        }
        return values;
    }
}

@Theory
public void multiplyIsInverseOfDivide4(@SimpleIntBetween(min=1, max=50) int amount,
```

Lectura recomendada

Documentación de hamcrest:

<https://code.google.com/archive/p/hamcrest/wikis/Tutorial.wiki>

<http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/Matchers.html>

[http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/
CoreMatchers.html](http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/CoreMatchers.html)

“JUnit Usage and Idioms” de la documentación de JUnit:

<https://github.com/junit-team/junit4/wiki>