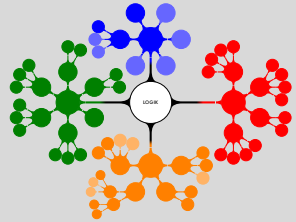
Tema 2

JUnit: Un marco de trabajo para la realización de pruebas de software en Java

Introducción Unit

Asignatura *Desarrollo de Software* fecha 5 abril 2017

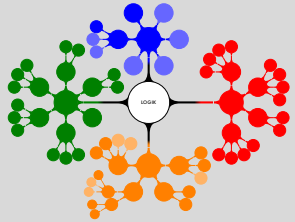
Manuel I. Capel
Departamento de Lenguajes y Sistemas Informaticos
Universidad de Granada



Desarrollo de software bajo contrato

Características

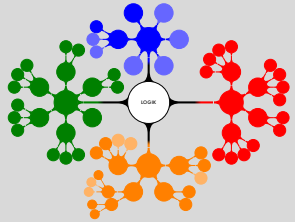
- Se trata de un paso más que permite avanzar en la dirección del *Desarrollo con Orientación a Objetos*
- Programación de los objetos de una aplicación para que se *comporten* de acuerdo con las especificaciones de los requisitos de usuario pre-establecidas
  - Cumplimiento de SLA (“service–level agreement").
  - Propicia la *composicionalidad* de objetos probados
- Se aconsejan los tests *automatizables* y *autoverificados* de los productos–software



Pruebas de Objetos

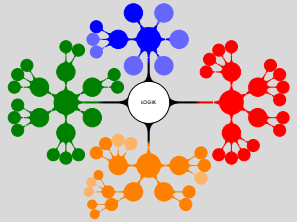
Concepto

- Se trata de probar los objetos de una aplicación de manera aislada,
  - cada objeto ha de *comportarse* correctamente de acuerdo con un *contrato* predefinido (SLA),
  - no se tiene en cuenta al sistema *alrededor* del objeto que se prueba,
  - los objetos probados individualmente se componen en sistemas correctos
- Escribir pruebas de objetos (*object testing*) consiste en escribir código para ensayar la ejecución de los objetos invocando sus métodos directamente, en lugar de hacerlo desde la aplicación



Pruebas de objetos vs. depuración de código

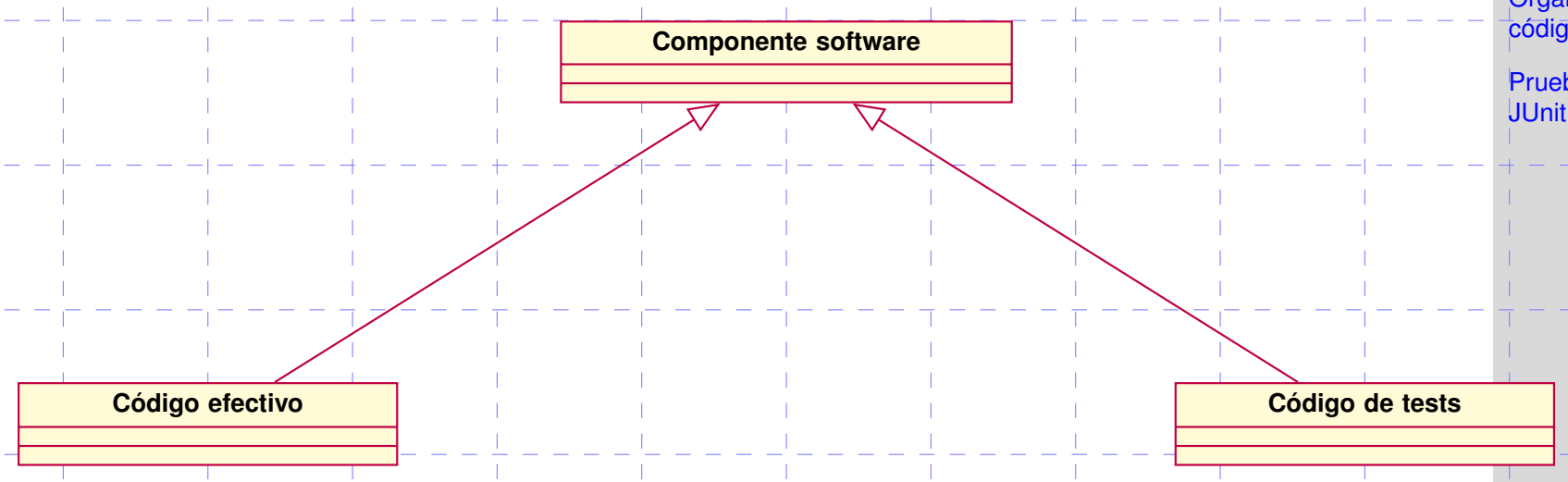
- La depuración de código perjudica la efectividad en el desarrollo de software (enlentecimiento del proceso):
  - No se presta a la automatización, ya que la depuración suele ser realizada manualmente por el programador sobre el código
  - Hay que recordar los valores correctos de las variables
  - Código específico para las pruebas, que aprovecha el profundo conocimiento de la aplicación que posee el programador
- Las pruebas de objetos propician la ejecución automática de los tests por el propio computador
- Las pruebas de software realizado de esta forma son *reproducibles*

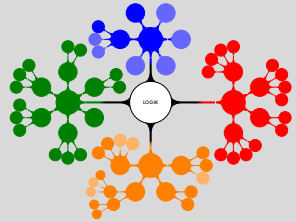


Desarrollo de software dirigido por las pruebas (TDD)

Motivación

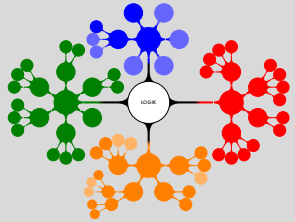
- Es mejor probar el software según se va desarrollando
- Productos del desarrollo según TDD:
  1. Código efectivo
  2. Tests





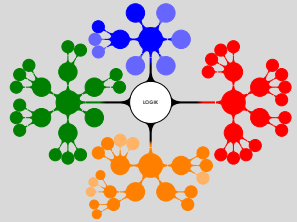
Beneficios de desarrollar según TDD

- Todo el código efectivo de la aplicación resulta cubierto por los tests
- Se consiguen componentes software altamente cohesivos y débilmente acoplados
- La *autoverificación*, según se va produciendo, propicia el *desarrollo incremental* del software
- Facilita la *refactorización* del código
- Facilita cambios posteriores, extensión, mantenimiento y extensión del software

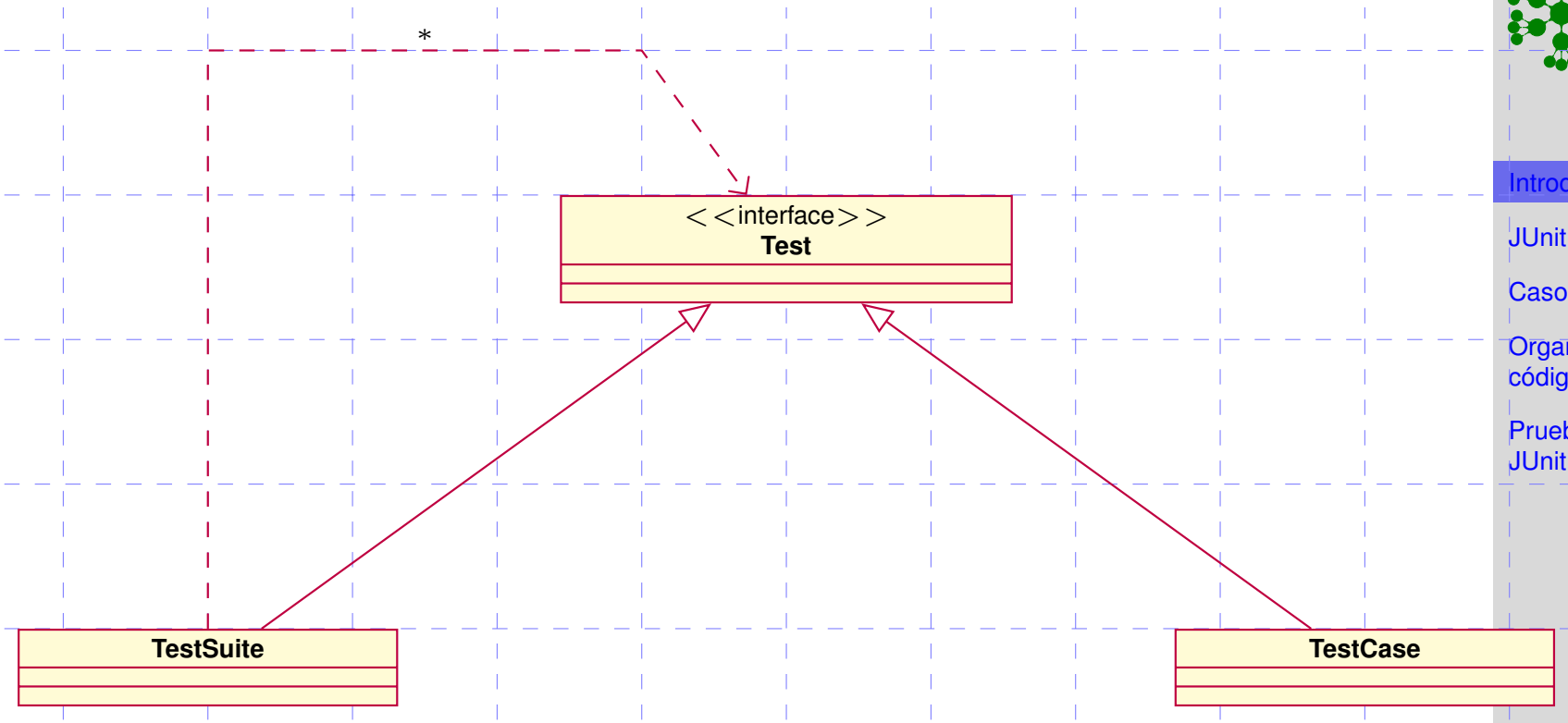


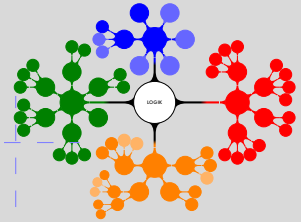
Entornos de TDD

- Proporcionan mecanismos automáticos de extracción de los tests individuales de cada clase
- Automatización de las pruebas: se deja que el propio computador las realice, en lugar de programar *“a mano"* un *driver*
- Reproducibilidad: el ejecutar la misma prueba varias veces, manteniendo las mismas condiciones sobre un objeto de código, ha de producir los mismos resultados
- Autoverificación: la propia prueba ha de indicarnos si ésta se realiza con éxito o falla
- Tratamiento diferenciado de colecciones de tests: utilización de `TestSuite`s



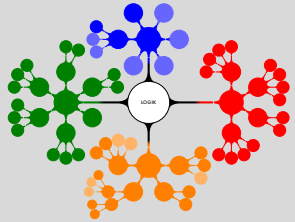
Entornos de TDD II





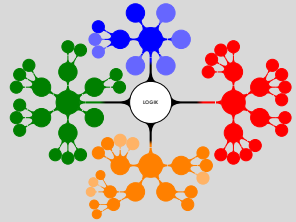
El marco de trabajo de pruebas JUnit

- JUnit ha evolucionado hacia un estándar industrial para el diseño y prueba de aplicaciones escritas en Java (ver, `http://junit.org`, `https://github.com/junit-team/junit4/wiki/Download-and-Install`)
- Agrupación y automatización de las pruebas: dentro de una clase Java
- JUnit proporcionaun *"motor de extracción"* basado en la identificación de los métodos `testXXX()` y de las clases que los contienen



Primer ejemplo

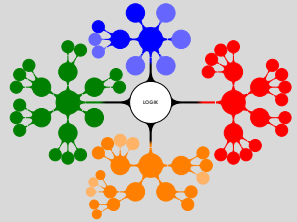
```
mis_tests.junit.primero;
import junit.framework.TestCase;

public class DineroTest extends TestCase{
  public void testAdd(){
    Dinero sumando= new Dinero(30,0);
    Dinero comparando= new Dinero(20,0);

    Dinero suma= sumando.add(comparando);
    assertEquals(5000, suma.inCents());
  }
}
```

Primer ejemplo II

Explicación

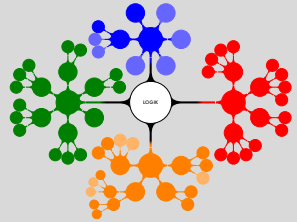
- Hay que ubicar un método que describe la prueba en una clase que extiende a la clase `TestCase` del marco de trabajo JUnit
- Programar `testAdd()` para completar la *prueba* que se incluye en la clase `DineroTest`
- JUnit buscará y ejecutará automáticamente los métodos que comiencen por `test`
- Escribimos asertos(`assertEquals()`, proporcionados) que han de asegurar que se cumple el *contrato* del objeto
- Si no falla ningún aserto cuando JUnit ejecuta el test, decimos que el código ha pasado la prueba



La clase `TestCase`

Características:

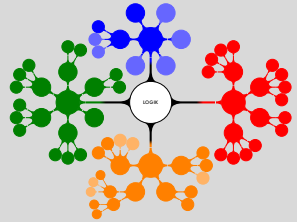
- Se encuentra en el paquete `junit.framework`
- Cada tipo de *prueba individual* que queramos realizar con un objeto se implementará como un método de una subclase de `TestCase`
- Cada uno de los *casos de prueba* que realicemos ha de ser un objeto de tal subclase
- Las partes fijas de una prueba se denominan (*"fixture"*)
- Fixture: la manera normal de agrupar elementos fijos que usa el código de las pruebas de tal manera que JUnit sólo tenga que ejecutarlos una única vez



La clase `Assert`

Extendida por `TestCase`

`assertTrue(boolean condicion)`	si cond.==false, falla; si no, *pasa*
`assertEquals(Object esperado, Object actual)`	utiliza el método `equals()`
`assertEquals(int esperado, int actual)`	utiliza el operador `==`
`assertSame(Object esperado, Object actual)`	mismo objeto en memoria
`assertNull(Object objeto)`	*pasa* si `objeto== null`; si no, falla



Métodos `assert` lógicamente contrarios

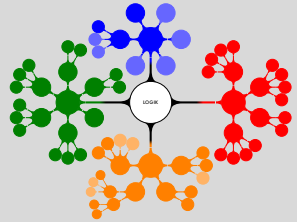
Proporcionados por JUnit:

- `assertFalse()`
- `assertNotSame()`
- `assertNotNull()`

Mensajes de fallo:

- Los métodos `assertXXX()` aceptan, como un primer parámetro optativo, un `String` para cuando el aserto falle
- `assertEquals()` posee su propio mensaje de fallo en JUnit:

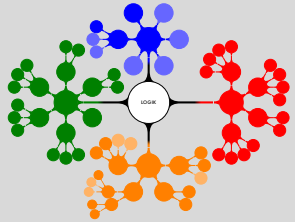
```
junit.framework.AssertionFailedError : expected < ... > butwas :< ... >
```

Operaciones aritméticas sobre un *pool* de monedas

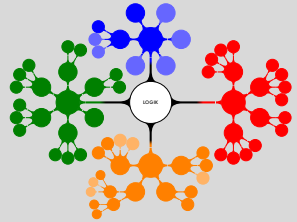
Condiciones del problema

- Se trata de una cartera de valores formada por una*bolsa* de monedas, cuyas cotizaciones cambian continuamente
- No existe una conversión de referencia para cada una de las monedas anteriores
- Hay que hacer operaciones sobre la bolsa de monedas y obtener su valoración global con respecto a diferentes cotizaciones diarias



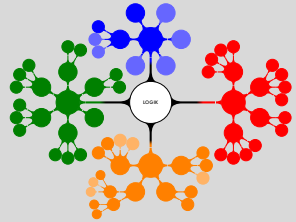
Representación como una clase Java

```
class Dinero {
    private int fCantidad;
    private String fMoneda;

    public Dinero(int cantidad, String moneda) {
        fCantidad= cantidad;
        fMoneda= moneda;
    }

    public int cantidad() {
        return fCantidad;
    }

    public String moneda() {
        return fMoneda;
    }
    public Dinero add(Dinero m){
      return new Dinero(cantidad()+m.cantidad(),moneda());
    }
}
```

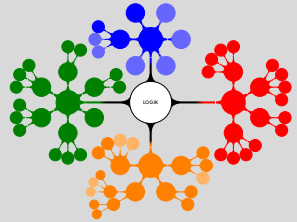
Representación de la clase `DineroTest`

- Inicialmente se ubica en el mismo paquete de la clase que queremos probar
- `DineroTest` tendrá, por tanto, acceso a todos los métodos que protege tal paquete



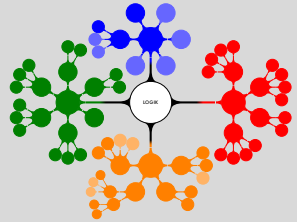
Representación de la clase `DineroTest` II

```
public class DineroTest extends TestCase {
    private Dinero f12CHF;
    private Dinero f14CHF;

    protected void setUp() {
        f12CHF= new Dinero(12, "CHF");
        f14CHF= new Dinero(14, "CHF");
    }
   //Hay que redefinir el metodo equals() de la clase
       Object
   public boolean equals(Object unObjeto) {
   // hay que redefinir este metodo para que determine si
   // el objeto que se le pasa, convertido a una instancia
   // de "Dinero" es del mismo tipo de moneda y tiene el
   // mismo valor nominal que "this".

   }
    return false;
}
}
```

Otros métodos a añadir a `DineroTest`

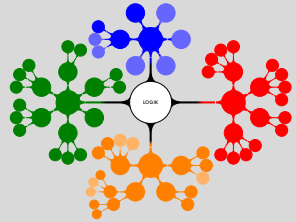
```
public void testEquals() {
  //Comprobar que: los "dineros" estan creados (! null),
  //que todo "dinero" es igual a si mismo (p.reflexiva),
  //que los "dineros" se crean bien
  //y que tomados 2 a 2, los dineros son diferentes.
  //Utilizar los metodos: assertTrue(), assertEquals() y
      equals()

}

public void testSimpleAdd() {
  //Para comprobar que funciona: por ejemplo, el resultado
  //de ingresar 14CHF con 12CFH
  //tiene que ser un dinero 26CHF.
  //Utilizar los metodos assertTrue() y equals()

}
```

Test de los asertos que contienen `equals()`

Prueba de un método *equals()* propio:

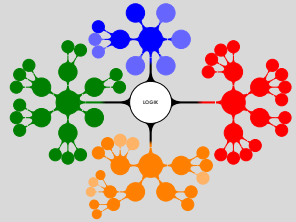
1. Propiedad reflexiva
2. Propiedad simétrica
3. Propiedad transitiva



Test de los asertos que contienen `equals()` II

Propiedad reflexiva

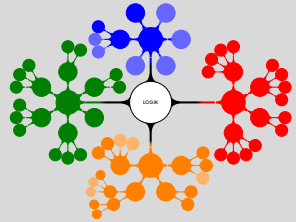
```
public class DineroEqualsTest extends TestCase{
  private Dinero a;
  protected void instanciar(){
    a= new Dinero(100,0);
  }
  public void testReflexiva(){
    assertEquals(a,a);
  }
}
```

Test de los asertos que contienen `equals()` III

Propiedad simétrica

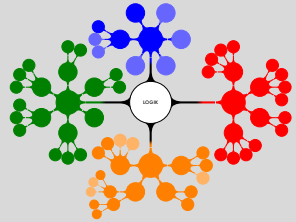
```
public class DineroEqualsTest extends TestCase{
  private Dinero a;
  private Dinero b;
  private Dinero c;
  protected void instanciar(){
    a= new Dinero(100,0); b= new Dinero(100,0);
    c= new Dinero(200,0);
  }
  public void testSimetrica(){
    assertEquals(a,b);
    assertEquals(b,a);
    assertFalse(a.equals(c));
    assertFalse(c.equals(a));
  }
}
```

Ejecución de los *casos de prueba*

Ejecuciones de las pruebas

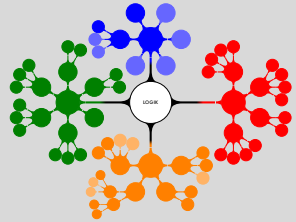
Tipo	Modo	Mecanismo
Individualmente	Estático	redefinición de `runTest()`
Individualmente	Dinámico	“reflexión" de Java
Dentro de *suite*	Estático	redefinición de `suite()`
Dentro de *suite*	Dinámico	“reflexión" de Java



Prueba de `DineroTest`: individual estática

Utilizando una clase interna y anónima

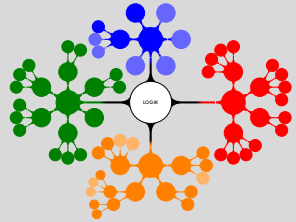
```
TestCase test= new DineroTest("simple add")
//Hay que darle un nombre al caso de prueba
{
    public void runTest() {
        testSimpleAdd();
    }
};
```

Prueba de `DineroTest`: individual dinámica

En este caso utilizaremos directamente el mecanismo de “reflexión" y no es necesario redefinir ningún método

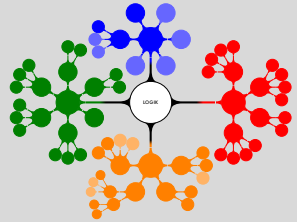
```
TestCase test= new DineroTest("testSimpleAdd");
```

Prueba de `DineroTest`: en `suite` estática

Instanciando una `TestSuite` y añadiendole los tests posteriormente

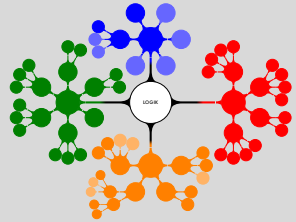
```
TestSuite suite= new TestSuite();
suite.addTest(
    new DineroTest("equals de dinero") {
        protected void runTest() { testEquals(); }
    });
suite.addTest(
    new DineroTest("add() simple") {
        protected void runTest() { testSimpleAdd(); }
    }
);
```

Prueba de `DineroTest`: en suite dinámica

En este caso utilizaremos directamente el mecanismo de “reflexión" cuando añadimos los métodos de prueba a la suite

```
    TestSuite suite= new TestSuite();
    suite.addTest(new DineroTest("testEquals"));
    suite.addTest(new DineroTest("testSimpleAdd"));
```

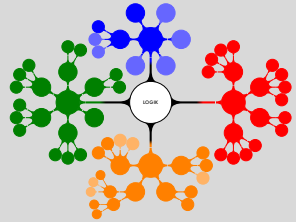



Prueba de `DineroTest`: en suite dinámica II

Desde la versión 2.0 de JUnit

Sólo hay que pasar la clase con los tests a un `TestSuite` para que éste extraiga los tests automáticamente

```
TestSuite suite= new TestSuite(DineroTest.class);
```

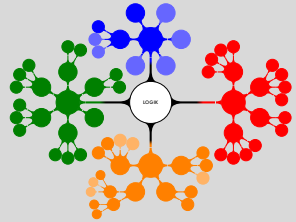



Tratamiento de los fallos en tests

Fallos de asertos en JUnit

- Cuando un aserto de JUnit falla, el método que implementa el aserto *lanza*(“throw" de Java) un `AssertionFailedError`:

```
static public assertTrue(boolean condition){
  if (!condition)
     throw new AssertionFailedError();
}
```

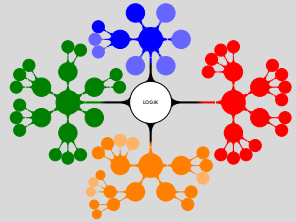
Tratamiento de los fallos en tests II

Fallos y *errores*

- Cuando un un método `testXXX()` contiene un aserto que falla, JUnit lo cuenta como un *test* que no ha pasado
- pero si el método *lanza* la excepción y no la captura, entonces JUnit lo cuenta como un *error*

Diferencias:

- Un aserto que falla normalmente indica que el código producido es incorrecto, pero un error significa que hay un problema bien en el propio test o en su entorno
- Si optamos por los errores, JUnit no puede llegar a ninguna conclusión y no puede saber si el test se completaría con éxito o no



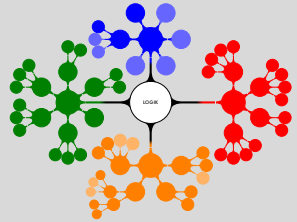
Test de lanzamiento de una excepción

Problema:

Queremos verificar que un método lanza la excepción adecuada si se dan las circunstancias apropiadas

Fallos de tests con JUnit

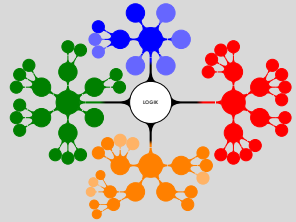
- Un test no pasa la prueba (con *fallo* o *error*) si falla un aserto o se lanza una excepción
- Un test debe fallar sólo si la línea de código incorrecta no lanza una excepción
- Debe capturar sólo el tipo de excepción adecuado al fallo
- Cualquier otra excepción ha de lanzarsela al marco de trabajo JUnit



Test de lanzamiento de una excepción II

Receta:

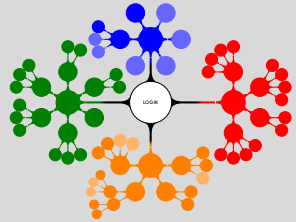
- Identificar la línea de código que puede lanzar una excepción y colocarla dentro de un bloque `try`
- En la línea siguiente programar una sentencia `fail()` para señalar: “si hemos llegado aquí, no se lanzó la excepción esperada"
- Añadir un bloque `catch` para capturar la excepción esperada
- Declarar que el método–test que programamos lanza una excepción (`throws Exception`)



Programación de un test para el lanzamiento de una excepción

Patrón de implementación

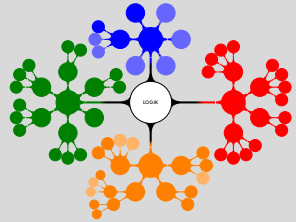
```
public void testConstructorAcabaEnNada() throws Exception{
   try{
      Fraction unoSobreCero= new Fraction(1, 0);
      fail("Fraccion creada 1/0! : no esta definida");
   }
   catch(IllegalArgumentException esperada){
      assertEquals("denominador", esperada.getMessage());
   }
}
```

Funcionamiento de la prueba de excepciones

Comportamiento del método–test

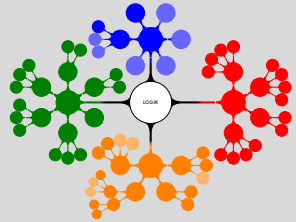
1. Lanza una excepción distinta de la *esperada*, entonces JUnit informa que se ha producido un error (no un *fallo*)
2. Un error indica que se ha producido un problema en el entorno o en el propio test, más que un problema en el *código producido*
3. Es conveniente capturar las excepciones (como la excepción *esperada* del ejemplo), ya que las excepciones se usan para mostrar caminos de ejecución de código *aberrantes*
4. No es necesario escribir asertos sobre las excepciones esperadas (y capturadas)



Tratamiento alternativo de las excepciones

Captura de todas las excepciones

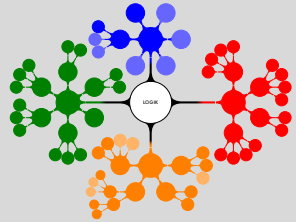
- Se quita responsabilidad en el lado del invocante del método respecto del mensaje de fallo
- Se capturan todas las excepciones que pueda generar el método–test
- Dado que no se puede conocer en compilación qué clase de excepción hay que esperar, utilizamos una *técnica reflectiva* con `Class.isInstanceof(Object)`
- Este enfoque evita replicar el código del algoritmo de captura de excepciones en múltiples tests



Tratamiento alternativo de las excepciones II

Captura de todas las excepciones

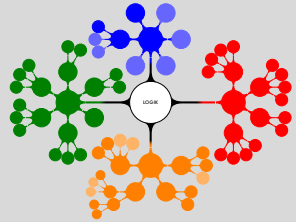
```
public void testParaExcepcion(){
  assertThrows(MiExcepcion.class,
               new ExceptionalClosure(){
   public Object execute(Object entrada) throws Exception{
               return hacerAlgoQuePuedaLanzarMiExcepcion()
                    ;}
               });
}
```

Tratamiento alternativo de las excepciones III

Captura de todas las excepciones II

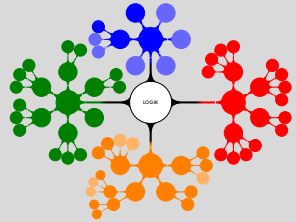
```
public static void assertThrows(
  Class claseExcepcionEsperada, ExceptionalClosure
      clausura){
   String nombreClaseExcepcionEsperada=
    claseExcepcionEsperada.getName();
   try{
      clausura.execute(null);
      fail("El bloque no lanzo una excepcion->"
   +nombreClaseExcepcionEsperada }
   catch(Exception e){
     assertTrue(
       "Capturada excepcion de tipo<"
       +e.getClass().getName()
       +">, esperada una del tipo <"
       +nombreClaseExcepcionEsperada
       +">",
       claseExcepcionEsperada.isInstance(e));
   }
}
```

Tratamiento alternativo de las excepciones IV

Captura de todas las excepciones III

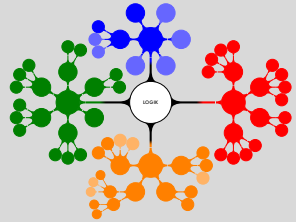
- Una *clausura*(“closure") es simplemenete un envoltorio de un bloque de código
- Se pueden encontrar interfaces `Closure` en Jakarate Commons Project o Diasparsoft Toolkit, que ha de definir un método `execute()` capaz de lanzar una excepción
- `assertThrows()` se puede incluir en una superclase de asertos adaptada a nuestra aplicación para utilizarla siempre que la necesitemos



¿Dónde situar los tests?

Algunas directrices

- Existe bastante diferencia –en cuanto a facilidad de uso– respecto de colocar (o no) el *código producido* y el *código de prueba* en el mismo paquete Java
- No existe una manera única de organizar el código de prueba en clases: no siempre resulta práctico tener 1 clase–test por cada clase–producción
- ¿Cuando se han de mover los métodos–test desarrollados a una nueva *fixture*? Normalmente, suele hacerse evidente cuando se ha desarrollado un número grande de tests para una aplicación



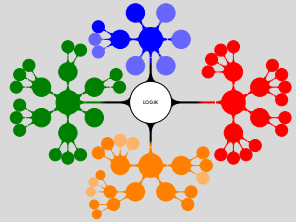
Creación de un árbol de código separado para las pruebas

Problema

Necesitamos distribuir el código producido de forma separada del código de pruebas

Receta

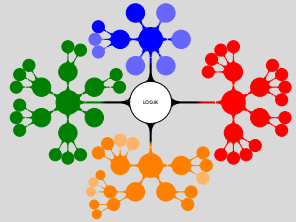
- Diseñar 2 estructuras de subdirectorios distintas
- Seleccionar 2 ubicaciones distintas para construir (`build`): crearse 2 subdirectorios distintos en el área denominada *workspace*
- Al ejecutar los tests, la dirección de las `clases/producidas` y de las `clases/pruebas` han de encontrarse en el `classpath` del ejecutor Eclipse



Creación de un árbol de código separado para las pruebas II

Receta

- Para distribuir el código producido: empaquetar sólo el contenido del subdirectorio `clases/producidas` junto con la documentación



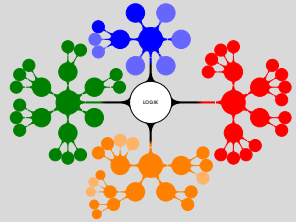
¿Cómo separar una *test fixture*?

Test fixture

Se trata de un conjunto de objetos que están siendo probados por métodos.
Una *configuración* de objetos cuyo comportamiento se puede predecir.

Comportamiento básico de un caso de test

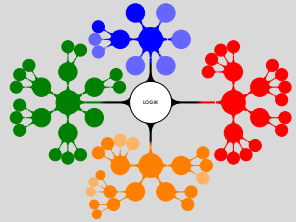
1. Crear algunos objetos (contribuyen a extender la *fixture*)
2. Invocar algunos métodos
3. Comprobar los resultados



¿Cómo separar una *test fixture*? II

¿Dónde se puede producir la mayor replicación de código entre tests distintos?

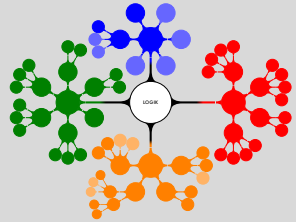
En la creación de objetos. Diferentes tests llaman al mismo constructor con diferentes parámetros.



¿Cómo separar una *test fixture*?III

Receta

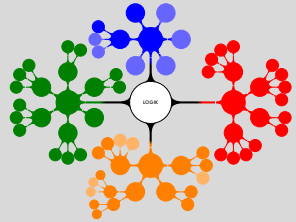
- Identificar el código duplicado de la *fixture* en tests similares
- Mover el código a un método común denominado `setUp()`
- Convertir a las variables dentro del método `setUp()` en *variables de instancia* de la clase–test, de tal forma que los métodos–test y el método `setUp()` hagan alusión a las mismas variables



¿Cómo separar una *test fixture*? IV

Receta

- Cada prueba se refiere a su propio conjunto de variables de instancia, no hay problema acerca de que los valores de dichas variables sean alterados incorrectamente por otra prueba
- JUnit proporciona soporte directo para definir `test-fixtures` con los métodos: `setUp()` y `tearDown()` que se encuentran en `junit.framework.TestCase`

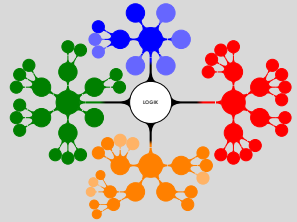


Utilización de `setUp()` y `tearDown()`

Subclasificación de `TestCase`

`setUp()` y `tearDown()` se pueden entonces redefinir en cada caso de test

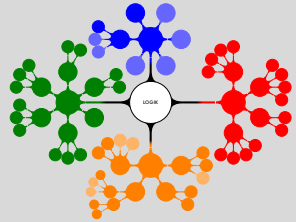
```
public void ejecutarEsquematico() throws Throwable{
   setUp();
   try{
      ejecutarTest();
   }
   finally{
     tearDown();
   }//garantiza que se llama, incluso si falla el test
}
```

Separar una jerarquía de test fixtures

Problema

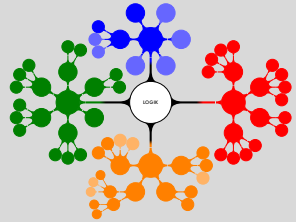
Tenemos varias fixtures que comparten objetos comunes, que están duplicados en la distintas clases de pruebas que implementan nuestra fixture



Separar una jerarquía de test fixtures II

Receta

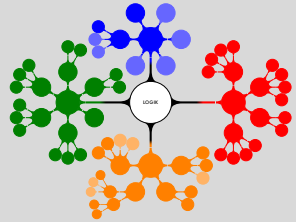
1. Seleccionar 2 casos de prueba que tengan código de una fixture en común
2. Crear una subclase (`BaseFixture`) de `TestCase` que se convertirá en la superclase de las 2 clases–test anteriores
3. Declarar a `BaseFixture` como una clase abstracta
4. Hacer que las 2 clases–test hereden de `BaseFixture`, en lugare de `TestCase`



Separar una jerarquía de test fixtures III

Receta (...)

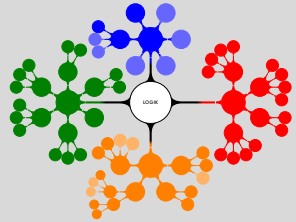
5. Copiar la fixture duplicada en `BaseFixture`; hacer los cambios necesarios en la visibilidad (`protected`, ahora) y encapsular utilizando métodos `get()`
6. Eliminar las variables de los métodos `setUp()` de las 2 clases–test
7. Añadir `super.setUp()` a los métodos `setUp()` de las 2 clases



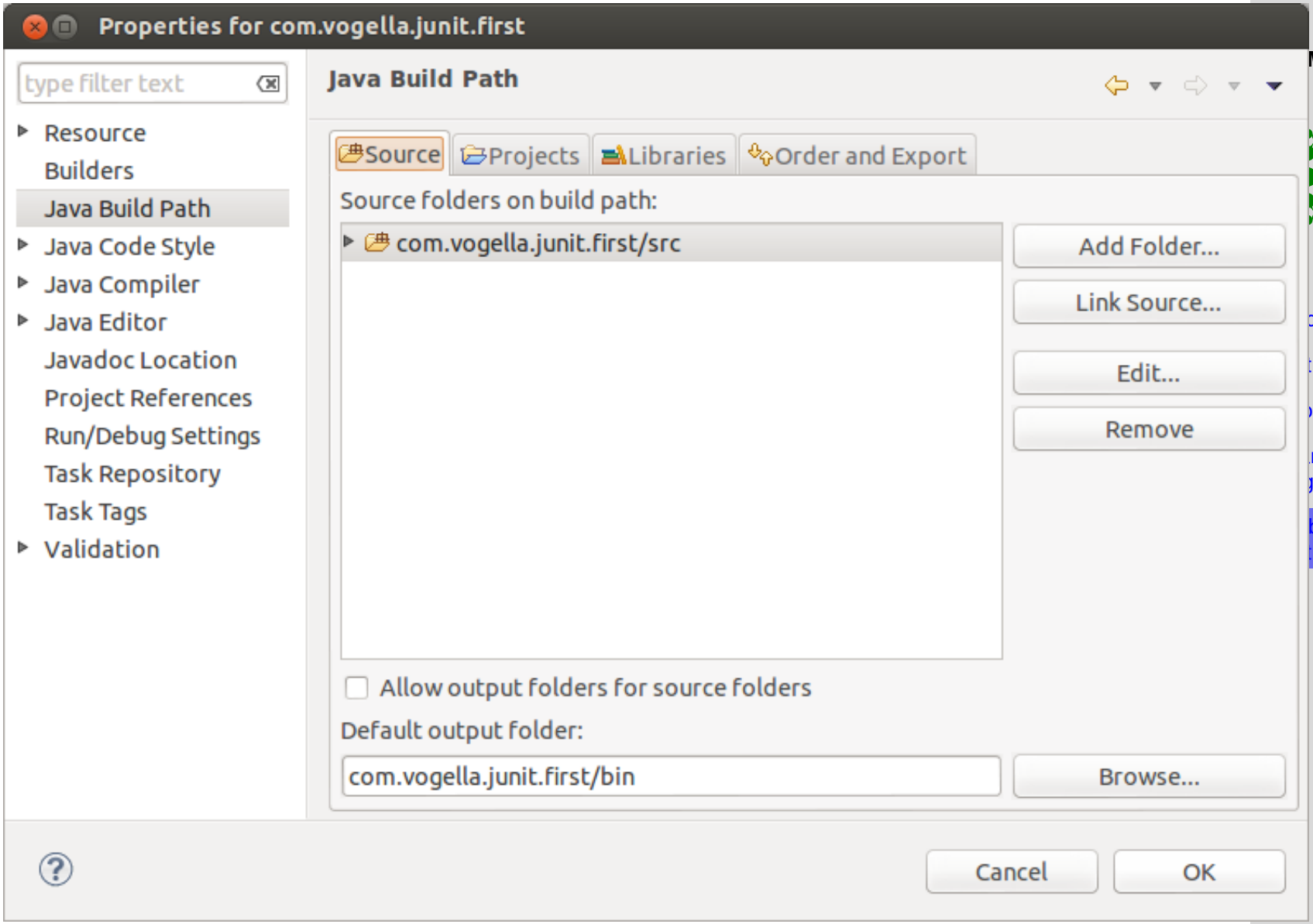
Utilización de JUnit con Eclipse

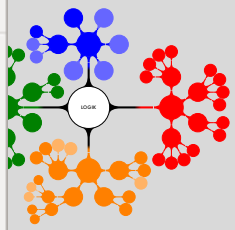
Preparación del *proyecto*

- Crear un nuevo proyecto llamado `mis_tests.junit.primero;`
- Crear un nuevo subdirectorio para el *fuente del test*:
- Picar con el botón derecho sobre el proyecto
- Seleccionar *Properties*
- Escoger *Java Build Path*
- Seleccionar la pestaña *Source*



Ventana *Properties* de proyecto

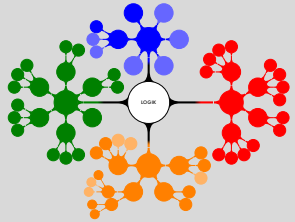




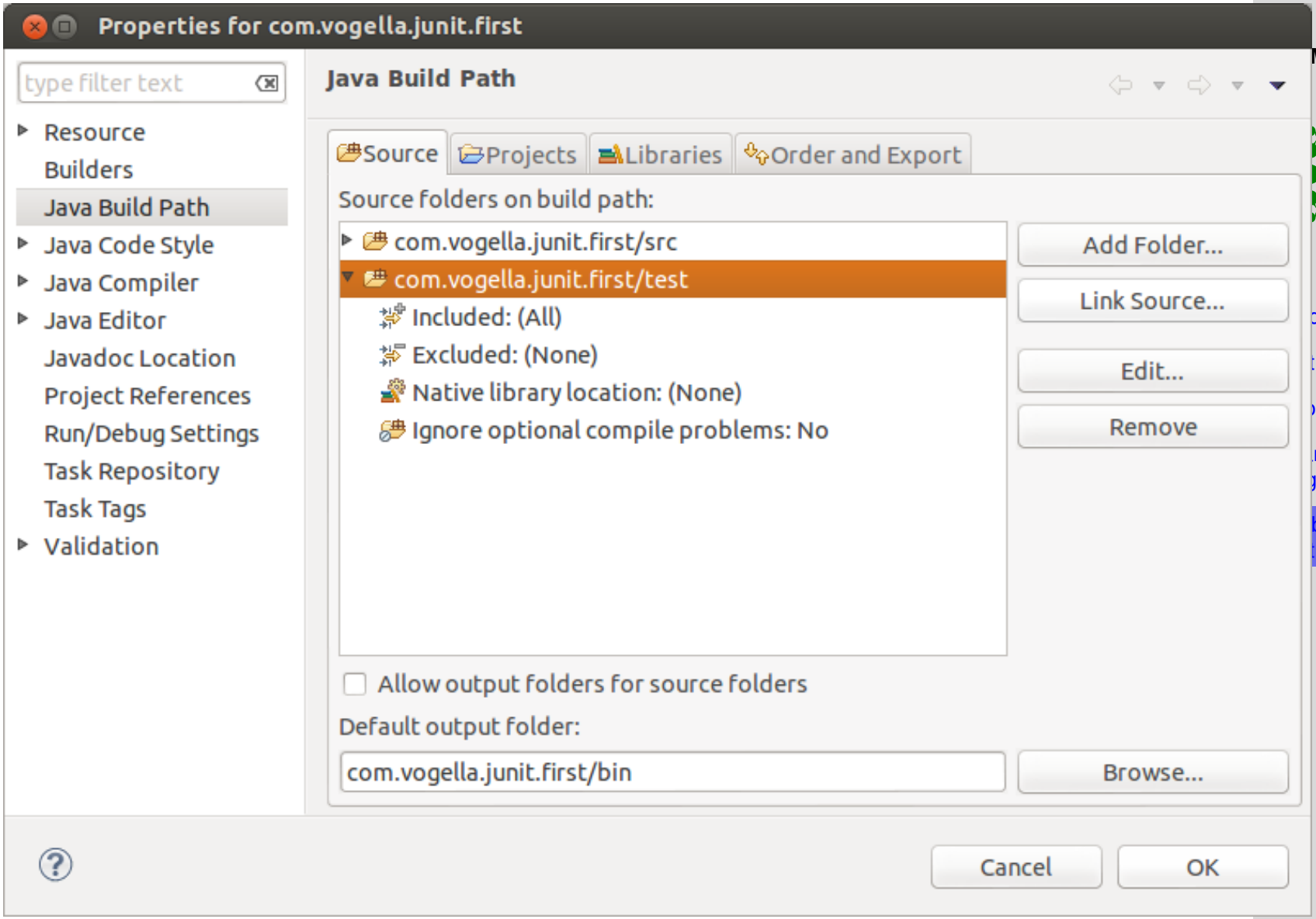
Subdirectorio `test`

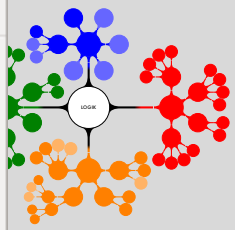
Añadir el subdirectorio

- Presionar el botón *Add Folder*
- después, presionar el botón *Create New Folder*,
- por último, crear el subdirectorio `test`
- De forma alternativa, se puede también añadir un nuevo subdirectorio fuente después de picar con el botón derecho en un proyecto y seleccionando la opción *New* $\rightarrow$ *Source Folder* en la ventana que aparece



Creación del subdirectorio `test`



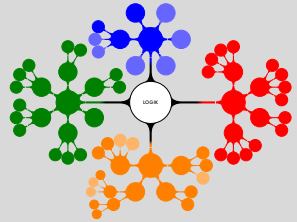


Creación de la clase de la prueba

Ubicación en subdirectorio `src`

En el subdirectorio `src`, crear el paquete `mis_tests.junit.primero;` y la siguiente clase:

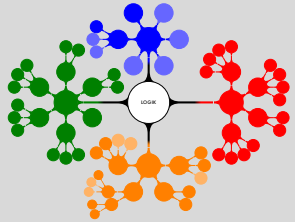
```
package mis\_tests.junit.primero;
public class MiClase {
  public int multiplicar(int x, int y){
          //lo siguiente no es mas que un ejemplo
          if(x>999){
                  throw new IllegalArgumentException("X␣
                      debe␣ser␣menor␣que␣1000");
          }
          return x/y;
   }
}
```

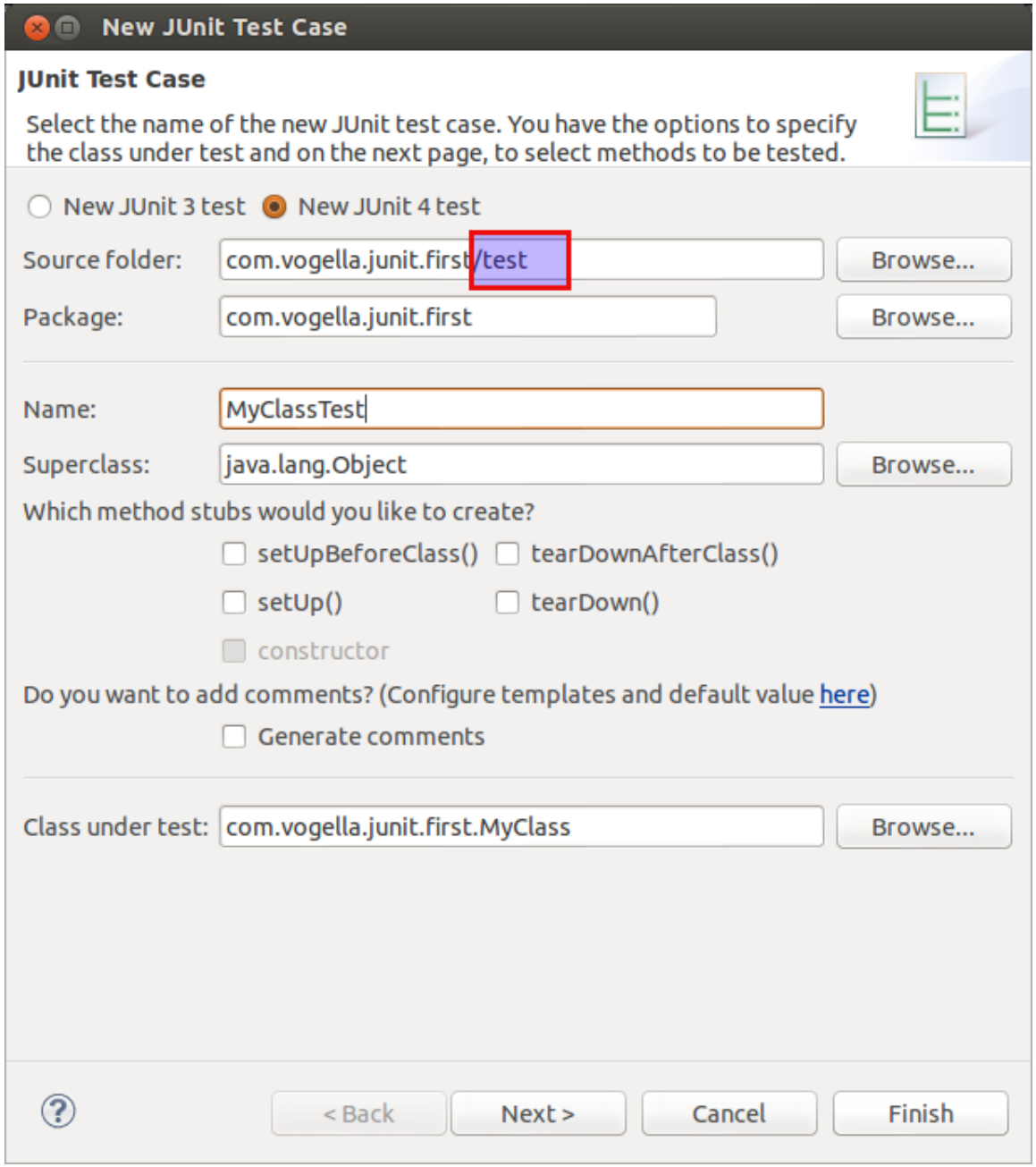
Creación de un test con JUnit

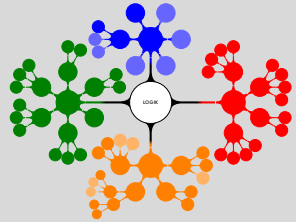
Utilización del asistente

- picar con el botón derecho sobre la nueva clase en la vista *Package Explorer* y seleccionar *New* → *JUnit Test Case*
- En el siguiente asistente, seleccionar el botón *New JUnit 4 test* y seleccionar el subdirectorio *fuente* del test, de tal forma que la clase–test se cree en este subdirectorio



Asistente para crear una clase–test

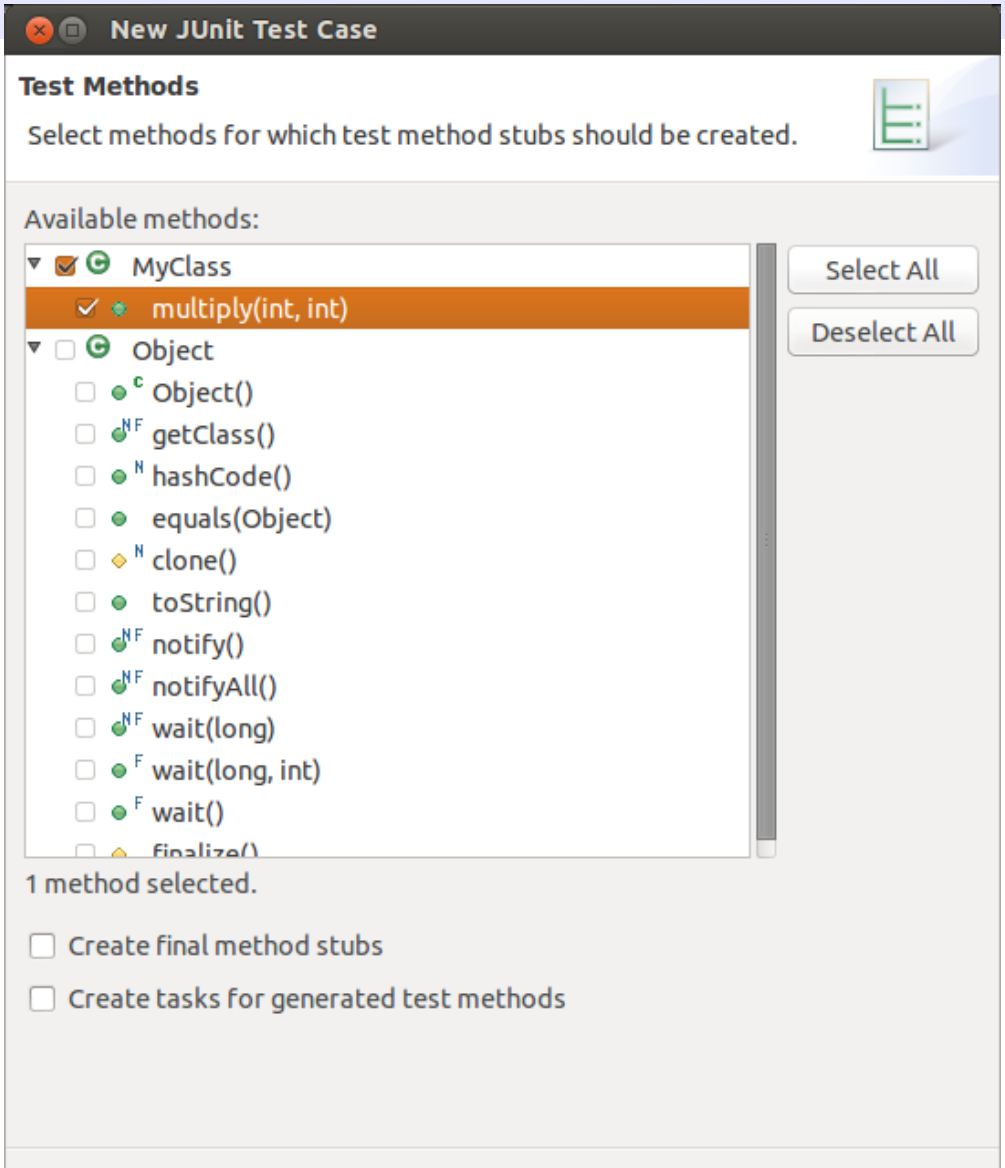


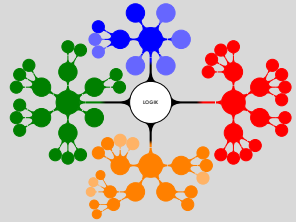


Seleccionar métodos a probar

Siguiente ventana del asistente

Picar en el botón *Next* y seleccionar los métodos que se vayan a probar

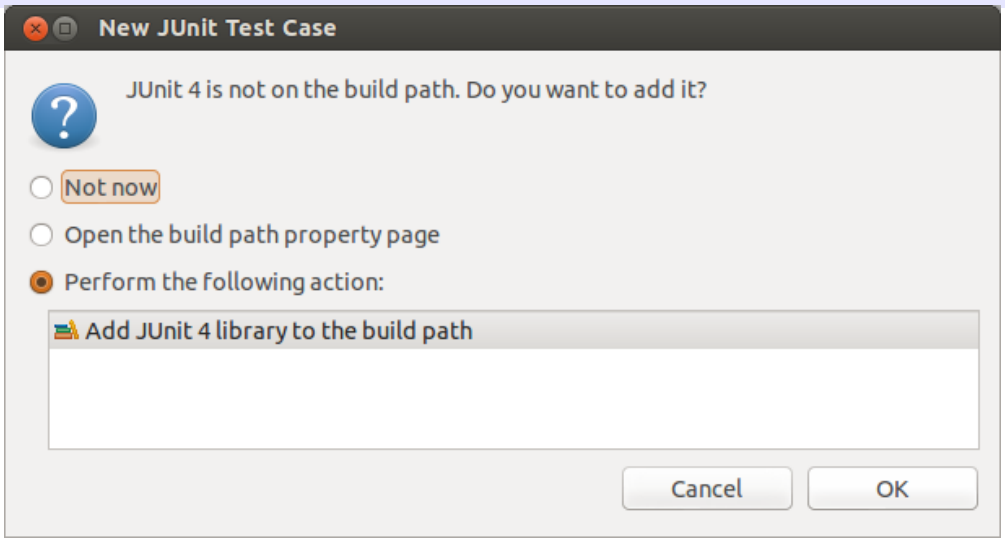


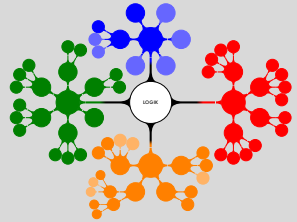


Seleccionar métodos a probar II

Siguiente ventana del asistente

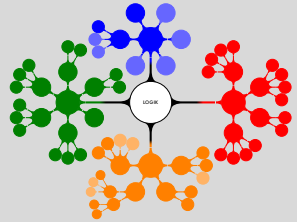
Si la biblioteca *JUnit* no es parte del `classpath` de nuestro proyecto, Eclipse nos demandará autorización para añadirlo:





Programación de la *clase–test*

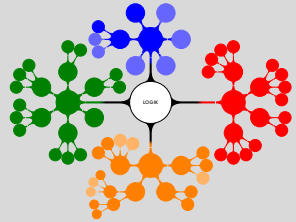
```
package mis_tests.junit.primero;import org.junit.
    AfterClass;
import static org.junit.Assert.assertEquals;
import org.junit.BeforeClass; import org.junit.Test;
public class MiClaseTest {
    @BeforeClass
    public static  void testInstanciar(){ }
    @AfterClass
    public static void testLimpiar(){
    //Destruye los datos utilizados en los tests de unit}
    @Test(expected = IllegalArgumentException.class)
    public void testExcepcionLanzada(){
        MiClase comprobador = new MiClase();
        comprobador.multiplicar(1000, 5);}
    @Test
    public void testMultiplicar() {
        MiClase comprobador = new MiClase();
        assertEquals("10 x 5 debe ser 50", 50,
                comprobador.multiplicar(10, 5)); } }
```

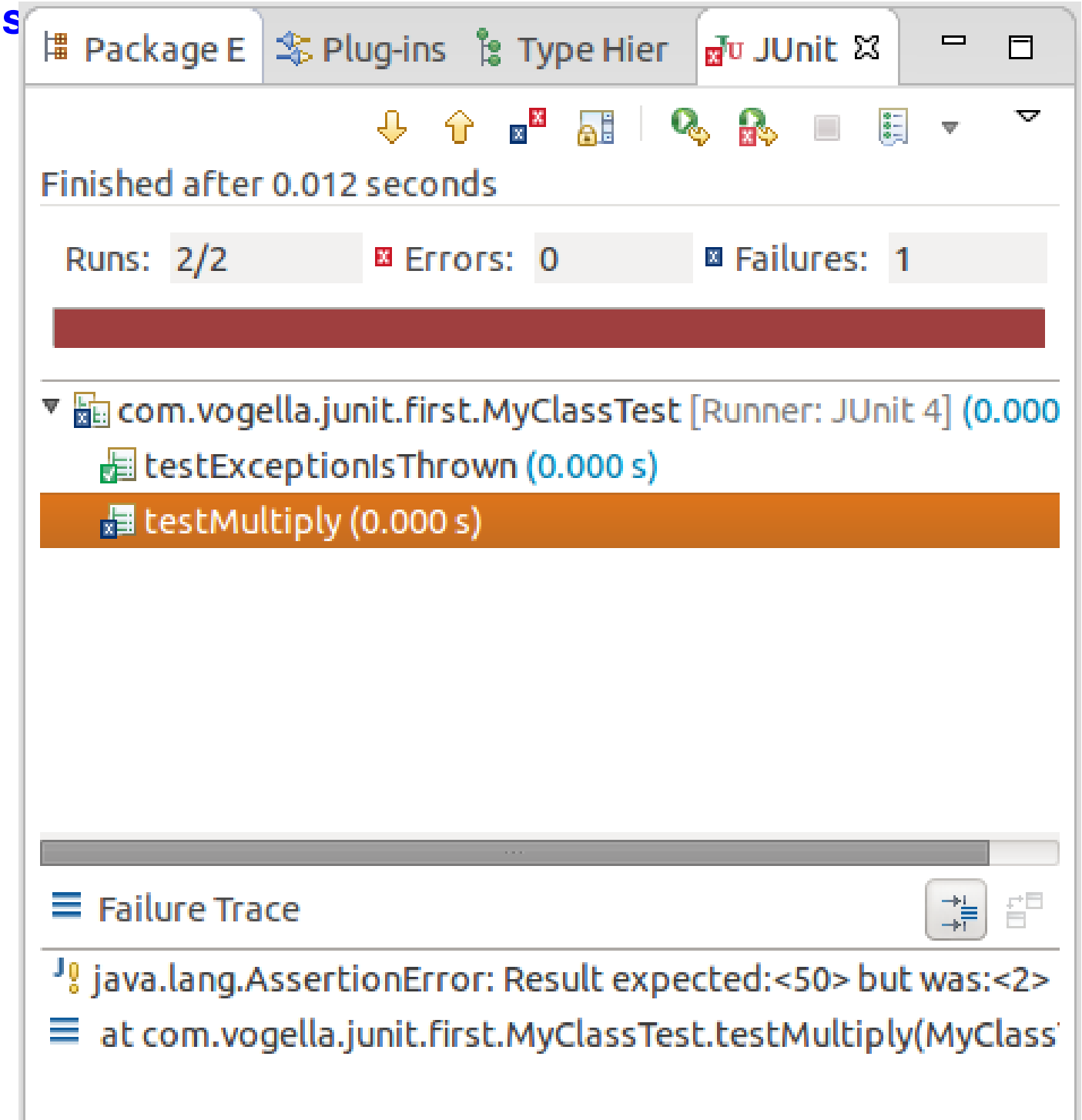
Ejecución del test en Eclipse

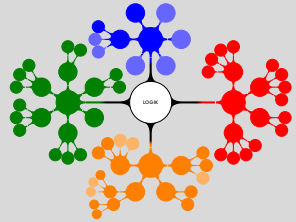
tests utilizando JUnit

- Picar con el botón derecho sobre la clase–test recién creada y seleccionar *Run-As* $\rightarrow$ *JUnit Test*
- Ejecutar *JUnit test* en Eclipse
- El resultado de las pruebas se mostrará en la vista *JUnit*



Asis

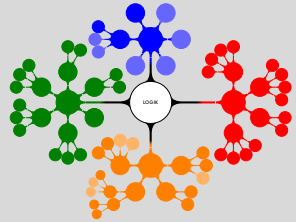




Ejecución del test en Eclipse II

Resultados de nuestro test–ejemplo

- Uno de los tests debe tener éxito
- y el otro ha de mostrar un error: que es indicado mediante una barra anaranjada
- El segundo test falla ya que nuestra clase *multiplicador* no está funcionando adecuadamente, ya que realiza una división en lugar de una multiplicación
- Corrigiendo el error y re–ejecutando el test se conseguirá obtener una barra verde, indicando que el código *ha pasado* sin error



Para ampliar


JUnit.org (2012).
JUnit Homepage.
http://www.junit.org/.


Kent, B. and Gamma, E. (2008).
JUnit Test Infected: Programmers Love Writing Tests.
http://junit.sourceforge.net/doc/testinfected/testing.htm.


Massol, V. and Husted, T. (2004).
JUnit in Action.
Manning Publications, Greenwich, Connecticut (USA).


Rainsberger, J. (2005).
JUnit Recipes: Practical Methods for Programmer Testing.
Manning Publications, Greenwich, Connecticut (USA).


Vogel, L. (2010).
JUnit Tutorial.
http://www.vogella.com/articles/JUnit/article.html.