Tecnología de la Programación

Pruebas de unidad: JUnit

David Cabrero Souto

Facultad de Informática
Universidade da Coruña

Curso 2007/2008

Pruebas de unidad

- Se validan los componentes (módulos) del sistema por separado
- Típicamente es un proceso de caja blanca
- P.e. En OO se realizan pruebas a nivel de clase
- Existen diversas herramientas para automatizar la ejecución de las pruebas.
  - P.e.: En Java: JUnit
  - Facilitan la integración en el proceso de desarrollo: escribir pruebas, codificar rutina, . . . compilar, ejecutar pruebas

Prueba de unidad

Pieza de código que pone a prueba la funcionalidad de un área del código pequeña y específica.

- Habitualmente, en OO, una prueba de unidad comprueba la ejecución de un método en un contexto determinado.
- P.e.: método `list.add` en una lista ordenada. Añadimos un elemento y comprobamos que la lista sigue ordenada.
- Cada módulo se prueba de forma aislada.
- Real testing checks results.
- Escrita y ejecutada por el programador.
- También documenta la intención del código.

Ventajas de las herramientas *Unit

- Ventajas de automático vs. manual
- La ejecución no para en el primer fallo
- Posibilidad de grupos de pruebas (testsuites)
- Obtención de informes
- Integración con otras herramientas de automatización de tareas (p.e.: ant)

Anatomía de una prueba de unidad

- Es una pieza de código que comprueba la ejecución de otra pieza de código.
- Las comprobaciones se realizan en base a *aserciones*.
  - Una aserción es una instrucción que comprueba que una condición es verdadera.
  - Por ejemplo:

```
int a = 2;
...
...
assertTrue(a == 2);
...
```

  - Si la condición no es cierta, la aserción falla.

Ejemplo (i)

- Método que calcula el número más grande de una lista.

```
public static int calcularMayor(int[] lista)
```

- ¿ Qué pruebas escribirías ?

Ejemplo (ii)

- El resultado es independiente de su posición en la lista.
  - $[7, 8, 9] \rightarrow 9$
  - $[9, 8, 2] \rightarrow 9$
  - $[7, 9, 1] \rightarrow 9$
- ¿ Qué pasa ocurre si está duplicado ?
  - $[9, 8, 9, 1, 5] \rightarrow 9$
- ¿ Qué ocurre si hay uno o cero elementos ?
  - $[4] \rightarrow 4$
  - $[\,] \rightarrow ?$
- ¿ Qué ocurre si son números negativos ?
  - $[-7, -8, -9] \rightarrow -7$

Ejemplo (iii)

```
public class Ejemplo {
  /**
   * Devuelve el número más grande de una lista.
   *
   * @param lista Una lista de enteros
   * @return El número más grande de la lista dada
   */
  public static int calcularMayor(int[] lista) {
    int index, max=Integer.MAX_VALUE;
    for (index=0; index < lista.length-1; index++) {
      if (lista[index] > max) {
        max = lista[index];
      }
    }
    return max;
  }
```

Ejemplo (iv)

- El código de las pruebas (JUNIT).

```
import junit.framework.*;

public class TestMayor extends TestCase {
  public TestMayor(String name) {
    super(name);
  }

  public void testOrden() {
    assertEquals(9,
      Ejemplo.calcularMayor(new int[] {8,9,7}));
  }
}
```

Ejemplo (v)

- Demo

```
java junit.swingui.TestRunner

java junit.textui.TestRunner TestMayor
```

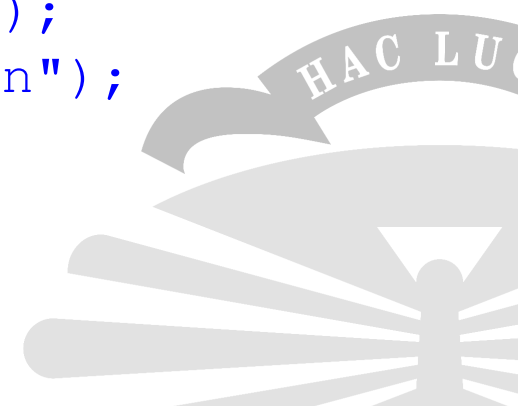
Ejemplo (vi)

- ¿ Qué hacemos con la lista vacia ? Lanzar una excepción

```
...
public static int calcularMayor(int[] lista) {
  int index, ...
  if (list.length == 0) {
    throw new RuntimeException("calcularMayor: Lista va
...
```

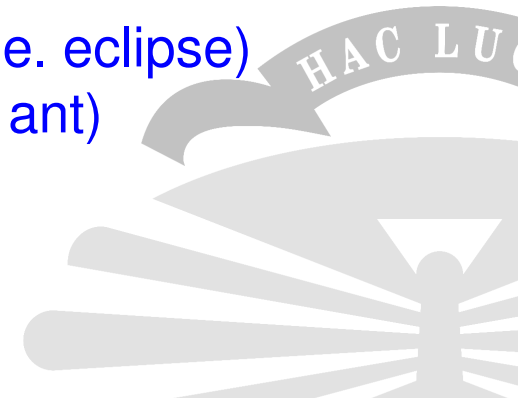
- Añadimos una nueva prueba:

```
public void testVacia() {
  try {
    Ejemplo.calcularMayor(new init[] {});
    fail("Should have throw an exception");
  } catch (RuntimeException e) {
    assertTrue(true);
  }
}
```



Introducción a JUnit

- Herramienta para automatizar las pruebas de unidad en Java.
- Basado en el uso de*aserciones* (assert) para caracterizar los resultados de la prueba.
- Las pruebas se codifican a nivel de objeto.
- Desarrollado por Kent Beck y Erich Gamma.
- Diversas interfaces de usuario:
  - CLI propio del framework de JUnit
  - GUI (swing) propio del framework de JUnit
  - Integrado en diversos entornos de desarrollo (p.e. eclipse)
  - Herramientas de automatización de tareas (p.e. ant)



Codificación de pruebas en JUnit

- Pasos básicos de una prueba
  1. Establecer las condiciones necesarias para la prueba (reservar recursos, crear objetos, ...)
  2. Llamar al método que se está validando
  3. Comprobar el resultado
  4. Limpiar (objetos, recursos, ...)
- Los resultados se comprueban mediante aserciones

```
assertXXX([String mensaje], esperado,
          obtenido)
```

- Las pruebas se codifican en una subclase de

```
junit.framework.TestCase
```

Aserciones en JUnit

Método	Comprueba que
`fail(`*msg*`)`	Nada. Falla siempre
`assertTrue(`*msg*`, b)`	b es cierto
`assertFalse(`*msg*`, b)`	b es falso
`assertEquals(`*msg*`, v1, v2)`	v1 = v2
`assertEquals(`*msg*`, v1, v2, e)`	| v1 - v2 | $\leq$ e
`assertNull(`*msg*`, obj)`	obj es null
`assertNonNull(`*msg*`, obj)`	obj no es null
`assertSame(`*msg*`, o1, o2)`	o1 y o2 son el mismo objeto
`asserNotSame(`*msg*`, o1, o2)`	o1 y o2 no son el mismo objeto

N.B.: No confundir con `assert boolean` (a partir de java 1.4)

JUnit TestCase

- Un TestCase es un cjto. de pruebas sobre la misma unidad
- Cada método `void testXXX()` implementa una prueba

```
import junit.framework.*;
public class TestEjemplo extends TestCase {

  public TestEjemplo(String nombre) {
    super(nombre);
  }

  public void testSuma() {
    assertEquals(2, 1+1);
  }

  public void testSumas() {
    assertEquals(2, 1+1);
    assertEquals(4, 2+2);
    assertEquals(-8, -12+4);
  }
}
```

Ejecutar un test

- Desde el framework

```
java junit.swingui.TestRunner

java junit.textui.TestRunner TestEjemplo
```

- Desde la propia clase TestCase

```
public static void main(String args[]) {
  String[] testCaseNames = {TestEjemplo.class.getName()};
  junit.swingui.TestRunner.main(testCaseNames);
}
```

Setup y Tear-down

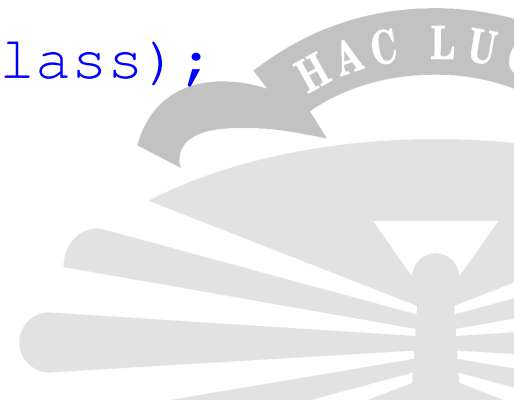
- La ejecución de cada prueba es independiente. Podemos necesitar:
  - Crear el entorno de ejecución
  - Deshacer los cambios

```
protected void setUp() { }
protected void tearDown() { }
```

Test Suites

- Composición de varias pruebas

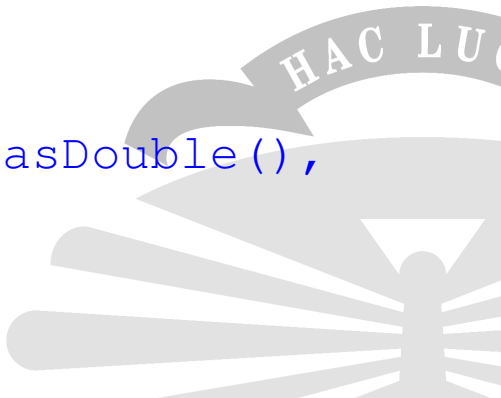
```
public static Test suite() {
   TestSuite suite = new TestSuite();
   suite.addTest(new TestEjemplo("testSuma"));
   suite.addTest(new TestEjemplo("testSumas"));
   return suite;

public static Test suite() {
   TestSuite suite = new TestSuite();
   suite.addTestSuite(TestNuevoEjemplo.class);
   suite.addTest(TestEjemplo.suite());
   return suite;
```



Aserciones a medida (i)

- Si programamos aserciones para tipos de datos específicos de nuestra aplicación, debemos evitar el copy-paste. P.e.:

```
public class ProyectTest extends TestCase {
  /**
   * Assert that the amount of money is an even
   * number of dollars (no cents)
   *
   * @param message Text message to display if the
   *                                  assertion fails
   * @param amount Money object to test
   */
  public void assertEvenDollars(String message,
                                Money amount) {
    assertEquals(message,
                 amount.asDouble() -(int)amount.asDouble(),
                 0.0,
                 0.001);
  }
```



Aserciones a medida (ii)

```
  /**
   * Assert that the amount of money is an even
   * number of dollars (no cents)
   *
   * @param amount Money object to test
   */
  public void assertEvenDollars(Money amount) {
    assertEvenDollars("", amount);
  }
}

public class TestSomething extends ProjectTest {
  ...
}
```

The Right-BICEP: Qué pruebas escribir

- Right. ¿ Los resultados son correctos ?
- B (Boundary). Condiciones en los límites
- I (Inverse). Relaciones inversas
- C (Cross-check). Comprobar resultados de varias formas
- E (Error conditions). Forzar errores
- P (Performance). Se mantiene el rendimiento en los límites esperados

Datos de las pruebas (i)

- Leer datos de ficheros, BB.DD., . . . P.e.:

```
# Test básico
9     7 8 9
9     9 8 7
9     9 8 9
# Test con números negativos
-7    -7 -8 -9
-7    -8 -7 -8
-7    -9 -7 -8
# Mezcla
7     -9 -7 -8 7 6 4
9     -1 0 9 -7 4
# Condiciones límite
1     1
0     0
2147483647     2147483647
-2147483648    -2147483648
```

Datos de las pruebas (ii)

```
String line;
BufferedReader rdr = new BufferedReader(
                              new FileReader("testdata.txt"));
while((line = rdr.readLine) != null) {
  if (line.starsWith("#")) { // Ignore comments
   cotinue;
  }
  StringTokenizer st = new StringTrokenizer(line);
  if (!st.hasMoreTokens()) {  // Blank line
    continue;
  }
  // Get the expected value
  String val = st.nextToken();
  int expected = Integer.valueOf(val).intValue();
```

Datos de las pruebas (iii)

```
// And the arguments to Largest
ArrayList arg_list = new ArrayList();
while(st.hadMoreTokens()) {
  arg_list.add(Integer.valueOf(st.nextToken()));
}
// Transfer object list into native array
int[] args = new int[argument_list.size()];
for(int i=0; i<arg_list.size(); i++) {
  args[i] = ((Integer)arg_list.get(i)).intValue();
}
// And run the assert
assertEquals(expected,
             Ejemplo.calcularMayor(args));
```

Datos en los límites

- Valores totalmente inconsistentes
  filename: “!*W#·& %¡HJYYa>/&Rb”
- Datos mal formateados
  email: “fred@udc.”
- Valores vacios o inexistentes
  0, 0.0, “”, null
- Valores ilógicos
  edad: 10.000 años
- Valores duplicados en listas sin duplicados
- Listas ordenadas que no lo están y viceversa
- Cosas en el orden no esperado
  P.e.: print antes de login

Usar las relaciones inversas

- Por ejemplo: operadores inversos

```
public void testSquareRoot() {
  double x = mySquareRoot(4.0);
  assertEquals(4.0, x * x, 0.0001);
}
```

- Otro ejemplo: insertar en B.D. y recuperar
- Peligro: repetir el mismo bug en ambas operaciones

Cross-check

- Usar diversas formas de realizar la operación
- P.e.: existen dos implementaciones de un algoritmo, la menos eficiente se sabe que es correcta

```
public void testSquareRoot() {
  double n = 3880900.0;
  double root1 = mySquareRoot(n);
  double root2 = Math.sqrt(n);
  assertEquals(root1, root2, 0.0001);
}
```

Forzar errores

- Errores del entorno de ejecución
  - Quedarse sin memoria
  - Quedarse sin espacio en disco
  - Hora del sistema erronea
  - Errores y caidas de la red
  - Carga del sistema
  - Paleta de color limitada, escasa resolución gráfica o exceso de resolución

Cuestiones de rendimiento

- Examinar la evolución del rendimiento

```
Timer timer = new Timer();
...
timer.start();
someOperation(smallList);
timer.end();
assertTrue(time.elapsedTime() < 1.0);

timer.start();
someOperation(mediumList);
timer.end();
assertTrue(time.elapsedTime() < 2.0);

timer.start();
someOperation(bigList);
timer.end();
assertTrue(time.elapsedTime() < 3.0);
```

Mock Objects (i)

- Las pruebas deben ser independientes. Pero algunas operaciones interactúan con otras partes del sistema . . .
- En el cine usan dobles, nosotros usamos *mock objects*:

  Sustituto de un objeto real, empleado en tareas de depuración

- Los Mock Objects son útiles cuando el objeto real:
  - tiene un comportamiento no determinista
  - es difícil de configurar
  - tiene algún comportamiento difícil de provocar (p.e. error de red)
  - es lento
  - implica una interface con el usuario
  - no nos indica cómo fue usado (p.e. asegurar que se llamó a una función de callback)
  - todavía no existe

Mock Objects (ii)

- Cuando usamos mocks:
  - Definimos la interface del objeto
  - Implementamos la interface para el objeto real
  - Implementamos la interface para el mock object

```
public interface Environment {
  public long getTime();
}

public class SystemEnvironment implements Environment {
  public long getTime() { return System.currentTimeMillis(); }
}

public class MockSystemEnvironment implements Environment {
  public long getTime() { return _currentTime; }
  public void setTime(long aTime) { _currentTime = aTime; }
  private long _currentTime;
}
```

Mock Objects (iii)

- Se expone la interface, imdependientemente de que se use:
  - el objeto real (sistema en producción)
  - un mock object (pruebas)

```
public class Checker {
  public Checker(Environment env) {
    this._env = env;
  }
  public void reminder() {
    Calendar cal = Calendar.getInstance();
    cal.setTimeInMillis(_env.getTime());
    int hour = cal.get(Calendar.HOUR_OF_DAY);
    if (hour >= 17) {
      _env.playWavFile("quit_whistle.wav");
    }
  }
  private Environment _env;
}
```

Mock Objects (iv)

```
public class MockSystemEnvironment implements Environment {
  public long getTime() {
    return _currentTime;
  }
  public void setTime(long aTime) {
    _currentTime = aTime;
  }
  public void playWavFile(String filename) {
    _playedWav = true;
  }
  public boolean wavWasPlayed() {
    return _playedWav;
  }
  public void resetWav() {
    _playedWav = false;
  }
  private long _currentTime;
  private boolean _playedWar;
}
```

Mock Objects (v)

- En las pruebas usamos mock objects

```
public void testQuittingTime() {
  MockSystemEnvironment env = new MockSystemEnvironment();
  // Set up a target test time
  Calendar cal = Calendar.getInstance();
  cal.set(Calendar.YEAR, 2008);
  cal.set(Calendar.MONTH, 4);
  cal.set(Calendar.DAY_OF_MONTH, 14);
  cal.set(Calendar.HOUR_OF_DAY, 16);
  cal.set(Calendar.MINUTE, 55);
  long t1 = cal.getTimeInMills();
  env.setTime(t1);
  Checker checker = new Checker(env);
  // Run the checker
  checker.reminder();
  assertFalse(env.wavWasPlayed());
```

Mock Objects (vi)

```
    // Advance time by 5 minutes and run again
    t1 += 5 *60 * 1000;
    env.setTime(t1);
    checker.reminder();
    assertTrue(env.wavWasPlayed());
    // Reset the flag, advance by 2 hours and try again
    env.resetWav();
    t1 += 2 * 60 * 60 * 1000;
    env.setTime(t1);
    checker.reminder();
    assertTrue(env.wavWasPlayed());
}
```

Frameworks de Mock Objects

- Algunos mock objects son laboriosos de implementar
  P.e.: contendedor de servlets
- Existen herramientas y frameworks
  - `http://www.mockobjects.com`
  - `http://www.easymock.org`