LPS: JUnit y Ant

Federico Peinado
www.federicopeinado.es

Depto. de Ingeniería del Software e Inteligencia Artificial
disia.fdi.ucm.es

Facultad de Informática
www.fdi.ucm.es

Universidad Complutense de Madrid
www.ucm.es

Pruebas formales

- Un paso más allá de la depuración o los registros (*logs*) de errores de la aplicación
- Son más líneas de código que usan nuestro código fuente, sometiéndolo a una especie de interrogatorio
  - ¿Hace lo que quiero?
  - ¿Hace todo/sólo/siempre lo que quiero?
- Cuantas más pruebas supere nuestro código fuente y más duras sean, más confianza tendremos en que funciona bien
- Las pruebas no se escriben al final, ya que además sirven para especificar el funcionamiento de nuestro código
  1. Defines una nueva clase o método
     (con documentación completa pero implementación trivial -tipo *mock/stub*-)
  2. Definir las pruebas que dicha clase o método debería superar
  3. Escribir el código de las pruebas
  4. Escribir el código de la nueva clase o método
     (implementación completa)
  5. Ejecutar las pruebas
     (si fallan, cambiar el código y volver a ejecutarlas tras cada cambio)

Propiedades deseables

- Automáticas
  - Muy sencillo ejecutarlas y comprobar resultados
- Reproducibles
  - En cualquier orden de ejecución producen siempre los mismos resultados (son deterministas solas y en conjunto)
- Independientes
  - El cambio en el código de una prueba no debe afectar a los resultados de las demás pruebas
- Completas
  - Probar todo lo que se pueda, sea obvio o insólito
- Profesionales
  - Seguir mismo criterio de calidad que en el código fuente para el código de las pruebas

JUnit

JUnit

- Armazón software que sirve para realizar pruebas formales automáticas cómodamente http://www.junit.org/
  - Usaremos la versión actual, 4.8.2
- Identifica el código de prueba y lo ejecuta, ofreciendo métodos para verificar si se el resultado cumplen ciertas condiciones
- Puede ejecutarse de varias formas
  - Directamente desde la consola
  - Desde Eclipse, dibujando barras verdes o rojas dependiendo de si se superaron o no todas las pruebas

Previo: Anotaciones en Java

- Información adicional sobre un programa Java que no afecta a cómo este se ejecuta
  - Disponible desde Java 1.5
  - Informan a las herramientas que quieran trabajar sobre nuestro código
  - Asisten a la compilación y otras funciones básicas
  - También pueden usarse en ejecución (`@Retention`)

```
@Author(
  name = "Benjamin Franklin",
  date = "3/27/2003"
)
class MyClass() {…}
```

  - Pueden llevar valores (con nombre, si hay varios)

Previo: Anotaciones del compilador

- `@Deprecated` avisa de que un elemento del programa está obsoleto
  - El compilador emite una advertencia (*warning*) al programador que lo use (similar a la etiqueta `@deprecated` de Javadoc)
- `@Override` anuncia que un elemento pretende sobreescribir a otro de una superclase
  - El compilador emite un error si la sobreescritura anunciada no se produce
- `@SuppressWarnings` solicita al compilador que no emita advertencias de cierto tipo

Previo: Definir y usar nuevas anotaciones

- Pueden definirse nuevas anotaciones como si fuesen una especie de interfaz
  - Se admiten valores por defecto

```
@interface ClassPreamble {
  String author();
  String date();
  int currentRevision() default 1;
  String lastModified() default "N/A";
  String lastModifiedBy() default "N/A";
  String[] reviewers(); // Se usan arrays
}
```

- Herramienta **`apt`** del compilador de Java

Organización de las pruebas

- Queremos separar el código de pruebas del código a probar, pero manteniendo una relación lógica
- Organización propuesta
  - Directorio **`test`** aparte, al mismo nivel que **`src`**
  - Subpaquetes **`test`** para cada paquete del código fuente
    - Se replica la estructura de paquetes de la práctica a probar
    - Cada clase a probar tendrá su clase prueba asociada
- Las pruebas realizan comprobaciones
  - Positivas: que el método funciona cuando debe
  - Negativas: que el método "falla" (devuelve *null*, lanza una excepción, etc.) cuando debe
- Todo el código de pruebas deberá ser automático
  - Sin intervención del usuario en ningún momento
  - No escriben ni muestran ninguna información por pantalla

Cómo programar las pruebas

- Inicializar lo que haga falta para ejecutarlas
  - Crear objetos, inicializar variables, etc.
- Llamar al método que se quiere probar
  - El método que prueba *xyz* se llamará *testXyz*
- Verificar que el método funciona/no funciona cuando debe, con un método de prueba
  - El método de prueba está anotado con `@Test`
  - Las verificaciones se hacen con métodos de **`Assert`**
- Limpiar lo que haga falta tras la ejecución

Assert y fail

- Dos valores u objetos son iguales

```
assertEquals([String message],expected,actual)
```

- Dos reales son iguales (con cierto nivel de tolerancia)

```
assertEquals([String
  message],expected,actual,tolerance)
```

- La referencia a un objeto es *null*

```
assertNull([String message], java.lang.Object object)
```

- La referencia a un objeto no es *null*

```
assertNotNull([String message], java.lang.Object
  object)
```

- Dos referencias apuntan al mismo objeto

```
assertSame([String message], expected, actual)
```

- Una determinada condición es cierta/falsa

```
assertTrue/assertFalse ([String message], condition)
```

- *Fallo forzado* (se ha llegado a donde no se debería llegar)

```
fail([String message])
```

Clase de prueba

```
package lps.pr1.testprofesor;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
// import de nuestras clases

public class ParserTest {

    private Parser _mi;

    @SuppressWarnings("deprecation")
    @Before
    public void setUp() throws Exception {
   String stest = "HLP\n" +
                  "HELP\n" +
    "help\n" +
    "LOOK\n" +
    "lOoK\n" +
    "GO\n" +
    "GO nrt\n" +
    "go East\n";

    _mi = new Parser(new java.io.StringBufferInputStream(stest));
    }
/** CONTINÚA */
```

Clase de prueba

```
@Test
   public void testNextCommand() {
  // 0. Wrong command
  assertNull("ERROR: Wrong command (HLP) interpreted as correct",
          _mi.nextCommand());

  // 1. HELP command
  Command c = null;
  assertNotNull("ERROR: Correct command (HELP) interpreted as wrong",
             _mi.nextCommand());

  assertNotNull("ERROR: Correct command (help) interpreted as wrong",
             c = _mi.nextCommand());

  assertEquals("ERROR: Wrong Verb after parsing HELP:",
            Verb.HELP,c.getVerb());
   }
(...)
}
```

Prueba de excepciones

- También se puede probar que un método lanza excepciones cuando debe (es tolerante a fallos)

```
public void testComprobarExcepcion() {
    try {
        Tablero t = new Tablero(-1,12);
        fail("Debería haber saltado una excepción");
    } catch (Exception e){
        //Comprobar si los datos de la excepción
        //son correctos usando asserts
    }
    try {
        Tablero t = new Tablero(10,32);
        fail("Debería haber saltado una excepción");
    } catch (Exception e){
        //Comprobar si los datos de la excepción
        //son correctos usando asserts
    }
}
```

Métodos setUp y tearDown

- Las pruebas de una clase suelen seguir esta estructura
  1. Se establece un estado inicial en el objeto a probar
  2. Se ejecuta un método de prueba, que llama al correspondiente método del código fuente y verifica su funcionamiento
  3. Se limpia el estado del objeto, antes de probar otro método
- Para los pasos 1 y 3 se usan métodos con las anotaciones `@Before` y `@After`, respectivamente

```
@Before
  protected void setUp()
@After
  protected void tearDown()
```

- Se implementan en cada clase de prueba, y JUnit los ejecuta antes y después de llamar a cada uno de los métodos de prueba marcados con `@Test`

Tests de API

- Se utiliza introspección para comprobar que las APIs de las clases y sus métodos son correctas

```
package lps.pr1.testsPr1;

import junit.framework.TestCase;
import java.lang.reflect.Method;

public class RoomTestAPI extends TestCase {

public RoomTestAPI(String name) {
   super(name);
}

Class getClassRoom() {
try {
    Class c;
    c = Class.forName("lps.pr1.Room");
    return c;
} catch (ClassNotFoundException ex) {
   fail("Clase/Enum/Interface Room no encontrado\n");
}
return null;
}
```

Tests de API

```
public void testEsClaseRoom() {
     Class c = getClassRoom();
     assertTrue("lps.pr1.Room no es una clase.\n",
               !c.isAnnotation() && !c.isEnum() && !c.isInterface());
}

public void testRoomHasMethod_isClosed() {

     Class c = getClassRoom();

     try {
        java.lang.reflect.Method m;
        // Para garantizar que puede saltar la excepción ClassNotFoundException
        // (así la generación de este código resulte más sencilla)
        Class.forName("java.lang.Object");
        m = c.getMethod("isClosed", Class.forName("lps.pr1.Parser$Direction"));
        assertEquals("El método isClosed no devuelve boolean.\n",  m.getReturnType(),
          Boolean.TYPE);
     } catch (NoSuchMethodException no) {
        fail("La clase Room no tiene el método isClosed.\n");
     } catch (ClassNotFoundException ex) {
        fail("La clase de algún parámetro del método isClosed no existe.\n");
     }
}
```

Baterías de pruebas

- JUnit ejecuta automáticamente todos los métodos de una clase de pruebas que lleven la anotación `@Test`
- Puede interesar tener una clase que invoque un conjunto de pruebas específicas o definidas en otras clases (las llamadas *test suites* o baterías de pruebas)
- Las baterías de pruebas pueden incluir
  - Clases de pruebas
  - Otras baterías de pruebas a su vez

Baterías de pruebas

```
package lps.pr1.testprofesor;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

import junit.framework.Test;
import junit.framework.TestSuite;

@RunWith(Suite.class)
@Suite.SuiteClasses( { AllTestAPI.class,
            RoomTest.class,
            MapTest.class,
            ParserTest.class } )

public class AllTests {
    // Add new classes to the SuiteClasses array
}
```

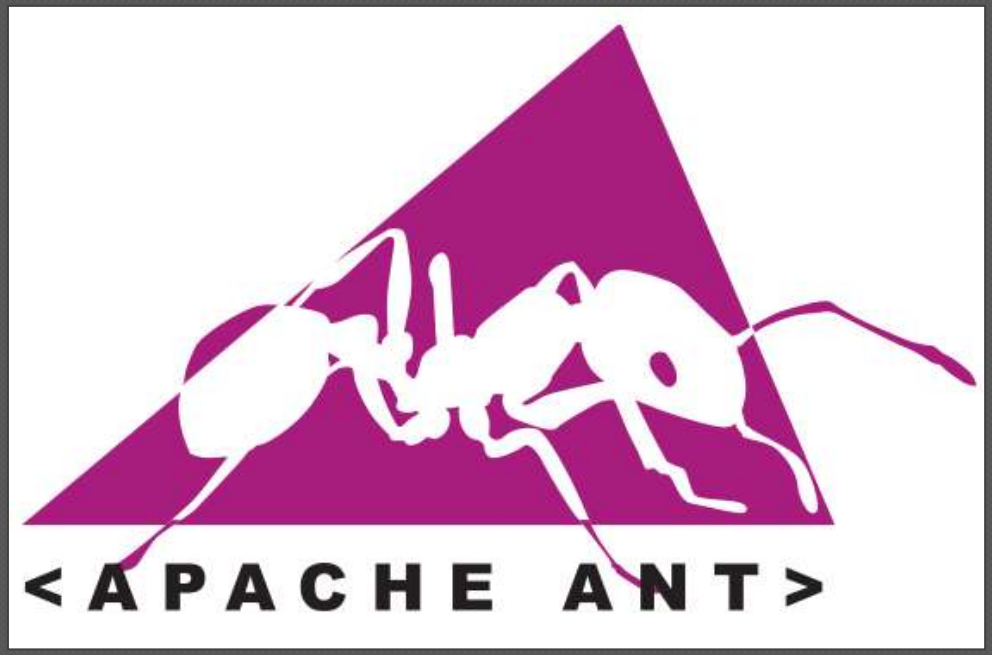
Uso de JUnit en Eclipse

- Debemos integrarlo manualmente en Eclipse
  - Descargar **junit-4.8.2.jar** e incluirlo en un directorio `lib` dentro de nuestro proyecto
  - Dentro de *Project* > *Properties* > *Java Build Path* > *Libraries* debemos añadir dicho fichero `jar`
  - Lo mismo se hace con el `jar` de las pruebas formales que proporciona el profesor para la corrección
- Una vez integrado y con alguna clase de prueba implementada hay que seleccionarla y hacer click derecho > *Run as JUnit Test*

Más documentación sobre JUnit

- JUnit Test Infected: Programmers Love Writing Tests http://junit.sourceforge.net/doc/testinfected/testing.htm
- Tutoriales JUnit http://open.ncsu.edu/se/tutorials/junit/
- Tahchiev, P., Leme, F., Massol, V., Gregory, G.: JUnit in Action. Second Edition (2010)
- Rainsberger, J.B.: JUnit Recipes: Practical Methods for Programmer Testing (2004)
- Hunt, A., Thomas, D.: Pragmatic Unit Testing in Java with JUnit (2003)

Ant



Ant

- Herramienta de construcción (*build tool*) mediante encadenamiento de tareas
  http://ant.apache.org/
  - Implementada (y extensible) en Java
  - Similar a Make, aunque multiplataforma
    - Independiente del S.O. subyacente, aunque puede acceder a funciones específicas con **`<exec>`**
  - Usa XML para los ficheros de configuración

Fichero de configuración

- Fichero (*Buildfile*) con las tareas de construcción a realizar para un proyecto (y con al menos un objetivo)

```
<project name="MyProject" default="dist" basedir=".">
  <description>
  simple example build file
  </description>

<!-- set global properties for this build -->
<property name="src" location="src"/>
<property name="build" location="build"/>
<property name="dist" location="dist"/>

<target name="init">
  <!-- Create the time stamp -->
  <tstamp/>
  <!-- Create the build directory structure used by compile -->
  <mkdir dir="${build}"/>
</target>
```

Fichero de configuración

```
<target name="compile" depends="init" description="compile the source
   " >
   <!-- Compile the java code from ${src} into ${build} -->
   <javac srcdir="${src}" destdir="${build}"/>
</target>

<target name="dist" depends="compile" description="generate the
   distribution" >
   <!-- Create the distribution directory -->
   <mkdir dir="${dist}/lib"/>
   <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar
   file -->
   <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
   basedir="${build}"/>
</target>

<target name="clean" description="clean up" >
   <!-- Delete the ${build} and ${dist} directory trees -->
   <delete dir="${build}"/>
   <delete dir="${dist}"/>
</target>

</project>
```

Críticas, dudas, sugerencias…

Federico Peinado

www.federicopeinado.es