Tema 9

Pruebas Unitarias en Java

Pruebas Manuales

Es responsabilidad de un programador el probar el código que produce para asegurarse de que este libre de errores y trabaja correctamente. Idealmente, el programador debería de probar cada método utilizando los suficientes valores de entrada que permitan asegurar que el método trabaja correctamente para cualquier tipo de entrada válida y aún para las entradas inválidas. Es importante notar que en un método puede haber sentencias condicionales y repetitivas por lo que los datos de entrada deberían de cubrir todos los posibles caminos a los que puedan llevarnos las sentencias condicionales y todas las posibles alternativas de entrar y salir de las sentencias repetitivas.

Considerando el número de valores que debemos emplear para probar un método y el número de métodos que un programador debe implementar en un proyecto dado, podemos entender que en la mayoría de los casos el programador sólo realice las pruebas para un limitado número de valores, dejando la posibilidad de que el método falle para otros valores y no sea detectado.

Si a lo anterior le agregamos que muchas veces los métodos tienen que ser modificados por que los requerimientos cambian o por que no cumplen con los requerimientos, obligando a repetir el proceso de prueba con cada modificación. Para ilustrar este proceso considere que tenemos la siguiente clase:

```
package pruebas;

public class Calculadora {
    public double suma(double x, double y) {
        return x + y;
    }

    public double resta(double x, double y) {
        return x - y;
    }

    public double producto(double x, int y) {
        int signo = 1;

        if(y < 0) {
            signo = -signo;
            y = -y;
```

```
        }

        double prod = 0.0;
        while(y > 0) {
            prod += x;
            y--;
        }

        return signo * prod;
    }
}
```

Para probar los métodos de la clase podemos construir la siguiente clase de prueba.

```
package pruebas;

public class CalculadoraTest {
    private int numErrores = 0;

    public static void main(String[] args) {
        CalculadoraTest ct = new CalculadoraTest();

        try {
            ct.testSuma();
        }
        catch(Exception e) {
            ct.numErrores++;
            e.printStackTrace();
        }

        try {
            ct.testResta();
        }
        catch(Exception e) {
            ct.numErrores++;
            e.printStackTrace();
        }

        try {
            ct.testProducto();
        }
        catch(Exception e) {
            ct.numErrores++;
            e.printStackTrace();
        }

        if(ct.numErrores > 0) {
            throw new RuntimeException("Hubo " + ct.numErrores + " Error(es)");
        }
   }

    /**
     * Prueba del metodo suma() de la clase Calculadora.
     */
    public void testSuma() {
        Calculadora calculadora = new Calculadora();
```

```
        System.out.println("Prueba del método suma");

        double resultado = calculadora.suma(20.0, 30.0);
        if(resultado != 50.0) {
            throw new RuntimeException(
              "Fallo testSuma: Valor esperado 50.0, obtenido " + resultado);
        }
    }

    /**
     * Prueba del metodo resta() de la clase Calculadora.
     */
    public void testResta() {
        Calculadora calculadora = new Calculadora();

        System.out.println("Prueba del método resta");

        double resultado = calculadora.resta(50.0, 30.0);
        if(resultado != 20.0) {
            throw new RuntimeException(
              "Fallo testResta: Valor esperado 20.0, obtenido " + resultado);
        }
    }

    /**
     * Prueba del metodo producto() de la clase Calculadora.
     */
    public void testProducto() {
        Calculadora calculadora = new Calculadora();
        double resultado;

        System.out.println("Prueba del método producto");

        resultado = calculadora.producto(50.0, 0);
        if(resultado != 0.0) {
            throw new RuntimeException(
               "Fallo testProducto: Valor esperado 0.0, obtenido "
               + resultado);
        }

        resultado = calculadora.producto(50.0, 3);
        if(resultado != 150.0) {
            throw new RuntimeException(
              "Fallo testProducto: Valor esperado 150.0, obtenido "
              + resultado);
        }

        resultado = calculadora.producto(50.0, -3);
        if(resultado != -150.0) {
            throw new RuntimeException(
               "Fallo testProducto: Valor esperado -150.0, obtenido "
               + resultado);
        }
    }
}
```

La corrida del programa de prueba anterior se muestra a continuación:

```
Prueba del método suma
Prueba del método resta
Prueba del método producto
```

Si modificamos el método `resta()` de la clase `Calculadora` para similar un error en el código:

```
public double resta(double x, double y) {
   return x + y;
}
```

La corrida de la clase de prueba mostraría lo siguiente:

```
Prueba del método suma
Prueba del método resta
java.lang.RuntimeException: Fallo testResta: Valor esperado 20.0,
 obtenido 80.0
        at pruebas.CalculadoraTest.testResta(CalculadoraTest.java:60)
        at pruebas.CalculadoraTest.main(CalculadoraTest.java:30)
Exception in thread "main" java.lang.RuntimeException: Hubo 1 Error(es)
        at pruebas.CalculadoraTest.main(CalculadoraTest.java:38)
Java Result: 1
Prueba del método producto
```

El Marco de Trabajo JUnit

Dada la importancia del proceso de probar el software y el trabajo que implicar realizar las pruebas en forma manual, se buscó la forma de automatizar las pruebas, lo que dio como resultado el marco de trabajo JUnit.

> *Un marco de trabajo es una aplicación semicompleta. Provee una estructura reusable y común que puede ser compartida entre aplicaciones. Los desarrolladores incorporan el marco en sus propias aplicaciones y la extienden para que se ajuste a sus necesidades específicas.*

JUnit es un marco de trabajo para desarrollar pruebas unitarias en Java.

> *Una prueba unitaria examina el comportamiento de una unidad de trabajo. En una aplicación Java, una unidad de trabajo es por lo general (pero no siempre) un simple método.*
>
> *Una unidad de trabajo es una tarea que no depende directamente de la terminación de otra tarea.*

Por el contrario las pruebas de integración y las pruebas de aceptación examinan cómo los componentes interactúan. Las pruebas unitarias por lo general se enfocan en

probar si un método sigue los términos de su contrato de la API. Si el contrato no puede cumplirse, el método lanza una excepción que indica que el contrato no puede cumplirse.

Un marco de trabajo para pruebas unitarias debería seguir las siguientes mejores prácticas:

- Cada prueba unitaria debe ejecutarse independientemente de las otras pruebas.
- Los errores deben detectarse y reportarse prueba por prueba.
- Debe ser fácil definir qué pruebas unitarias deben ejecutarse.

El siguiente código muestra una clase de prueba para la clase `Calculadora` usando el marco de trabajo JUnit 4:

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

public class CalculadoraTest {
    /**
     * Prueba del metodo suma() de la clase Calculadora.
     */
    @Test
    public void testSuma() {
        System.out.println("Prueba del método suma");
        Calculadora calculadora = new Calculadora();

        double result = calculadora.suma(20.0, 30.0);
        assertEquals(50.0, result, 0.0);
    }

    /**
     * Prueba del metodo resta() de la clase Calculadora.
     */
    @Test
    public void testResta() {
        System.out.println("Prueba del método resta");
        Calculadora calculadora = new Calculadora();

        double result = calculadora.resta(50.0, 30.0);
        assertEquals(20.0, result, 0.0);
    }

    /**
     * Test of producto method, of class Calculadora.
     */
    @Test
    public void testProducto() {
        System.out.println("Prueba del método producto");
        Calculadora calculadora = new Calculadora();
        double result;

        result = instance.producto(50.0, 0);
        assertEquals(0.0, result, 0.0);
```

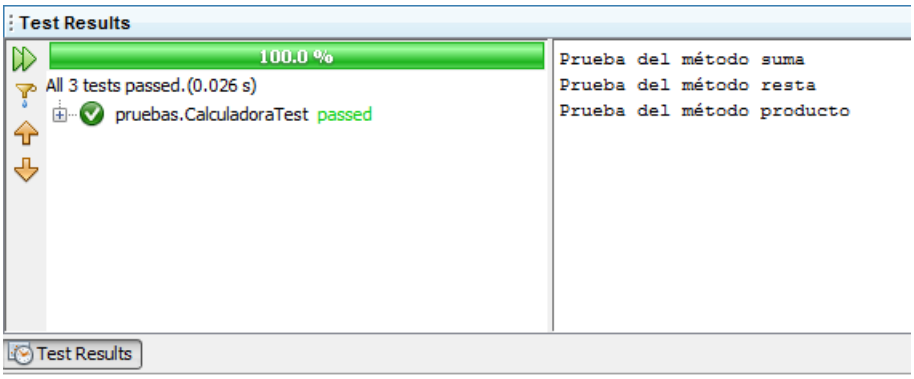
```
        result = instance.producto(50.0, 3);
        assertEquals(150.0, result, 0.0);

        result = instance.producto(50.0, -3);
        assertEquals(-150.0, result, 0.0);
    }
}
```

La anotación `@Test` le indica al compilador que trate a la clase como una clase de prueba y al método anotado como un método de prueba.

El método `assertTrue()` compara el resultado de invocar al método a probar con los argumentos dados contra el resultado esperado. Si la comparación tiene éxito no se hace nada pero si falla el método lanza una excepción.

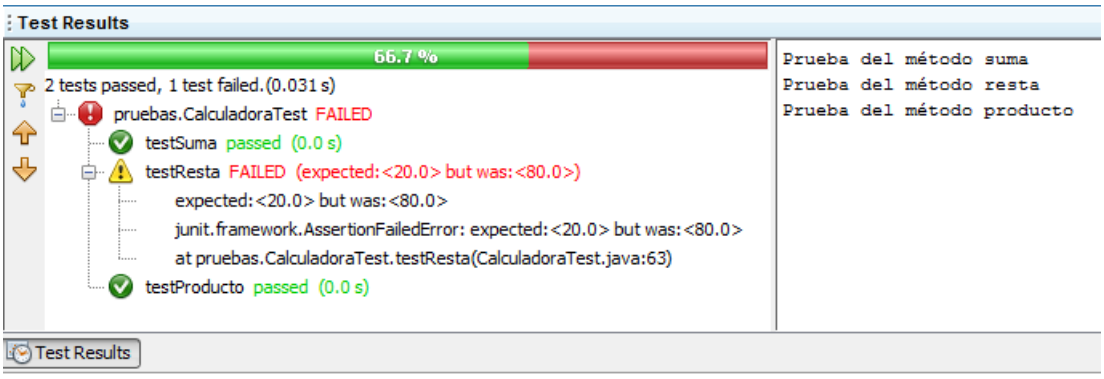
La corrida del programa de prueba anterior se muestra a continuación:



Si modificamos el método `resta()` de la clase `Calculadora` para similar un error en el código:

```
public double resta(double x, double y) {
   return x + y;
}
```

La corrida de la clase de prueba mostraría lo siguiente:



Anotaciones de JUnit 4

La tarea de programar las clases de prueba en JUnit 4 se simplifica mediante el uso de anotaciones que le indican a JUnit que realice realice una tarea. Esas anotaciones son las siguientes.

@Test

La anotación `@Test` le indica a JUnit que el método `public void` sin argumentos adjunto debe ejecutarse como un caso de prueba. Para ejecutar el el método, JUnit crea una nueva y luego invoca al método anotado. Cualquier excepción arrojada por el método será reportada por JUnit como un fracaso. Si no se lanza ninguna excepción la prueba tuvo éxito.

La sintaxis de la anotación `@Test` es:

```
public class nombreClasePrueba {

    @Test[(expected=nombreExcepcion.class, timeout=valor)]
    public void nombreMetodoPrueba() {
       ...
    }

    ...
 }
```

La anotación `@Test` soporta dos parámetros opcionales. El parámetro:

```
expected=nombreExcepcion.class
```

declara que el método debería lanzar la excepción dada por *`nombreExcepcion`*. Si el método no lanza esa excepción o lanza una de un tipo diferente, la prueba falla.

El parámetro:

```
timeout=valor
```

Hace que la prueba falle si le toma más del valor especificado por `valor`, en milisegundos.

@BeforeClass, @AfterClass

La anotación `@BeforeClass` le indica a JUnit que el método `public static void` sin argumentos adjunto debe ejecutarse una vez antes que cualquier método de la clase. Sólo puede haber un método `@BeforeClass` por clase. Los métodos `@BeforeClass` de las superclases se ejecutarán antes que los otros métodos de la clase.

La anotación `@AfterClass` le indica a JUnit que el método `public static void` adjunto debe ejecutarse una vez después que cualquier método de la clase. Sólo puede haber un método `@AfterClass` por clase. Los métodos `@AfterClass` de las superclases se ejecutarán después que los otros métodos de la clase. Todos los métodos `@AfterClass` se ejecutarán aún cuando un método `@BeforeClass` lance una excepción.

La sintaxis de las anotaciones `@BeforeClass` y `@AfterClass` son:

```
public class nombreClasePrueba {

    @BeforeClass
    public static void nombreMetodo() {
       ...
    }

    @Test
    public void nombreMetodoPrueba1() {
       ...
    }

    @Test
    public void nombreMetodoPrueba2() {
       ...
    }

    ...

    @AfterClass
    public static void nombreMetodo() {
       ...
    }
 }
```

@Before, @After

La anotación `@Before` le indica a JUnit que el método `public void` adjunto debe ejecutarse antes que cada uno de los métodos de prueba. Puede haber más de un método `@Before` por clase. Los métodos `@Before` de las superclases se ejecutarán antes que los otros métodos de la clase.

La anotación `@After` le indica a JUnit que el método `public void` adjunto debe ejecutarse después que cada uno de los métodos de prueba. Puede haber más de un método `@After` por clase. Los métodos `@After` de las superclases se ejecutarán después que los otros métodos de la clase. Todos los métodos `@After` se ejecutarán aún cuando un método `@Before` lance una excepción.

La sintaxis de las anotaciones `@BeforeClass` y `@AfterClass` son:

```
public class nombreClasePrueba {
```

```
    @Before
    public void nombreMetodo() {
       ...
    }

    @Test
    public void nombreMetodoPrueba1() {
       ...
    }

    @Test
    public void nombreMetodoPrueba2() {
       ...
    }

    ...

    @After
    public void nombreMetodo() {
       ...
    }
}
```

@Ignore

La anotación `@Ignore` le indica a JUnit que uno o varios métodos de prueba no serán ejecutados.

Hay una corriente de pensamiento en programación llamada Diseño Manejado por las Pruebas que sostienen que primero se deben implementar las pruebas y luego codificar las clases para que pasen las pruebas. En este caso podemos tener ya implementadas las pruebas de ciertos métodos que no han sido implementados. En lugar de en la prueba fallen las pruebas de esos métodos podemos indicarle a JUnit que no realice la prueba para esos métodos no implementados.

La sintaxis de la anotación `@Ignore` es:

```
public class nombreClasePrueba {

    @Ignore[("mensaje")]
    @Test[(expected=nombreExcepcion.class, timeout=valor)]
    public void nombreMetodoPrueba() {
       ...
    }

    ...
}
```

La anotación `@Ignore` puede incluir una cadena como parámetro, si se quiere registrar la causa por la que se está omitiendo el caso de prueba.

La anotación `@Ignore` también puede usarse para anotar la clase de prueba. Ninguno de los métodos de prueba de la clase se ejecutarán. Su sintaxis es:

```
@Ignore[("mensaje")]
public class nombreClasePrueba {

    @Test[(expected=nombreExcepcion.class, timeout=valor)]
    public void nombreMetodoPrueba() {
       ...
    }
    ...
 }
```

La clase `Assert` de JUnit 4

En los casos de prueba, podemos utilizar los métodos de la clase `Assert` para determinar si un método a probar pasa la prueba o no usando aserciones. Sólo las aserciones que fallan son registradas. Como los métodos de la clase `Assert` son estáticos podemos invocarlos precidiéndolos del nombre de la clase. Por ejemplo:

```
Assert.assertEquals(...)
```

Sin embargo podemos importar estáticamente a la clase:

```
import static org.junit.Assert.*;
    ...
    assertEquals(...);
```

La descripción de los diferentes métodos de la clase `Assert` se encuentran en la tabla 9.1.

Tabla 9.1 Métodos de la Clase Assert

```
public static void assertEquals(double expected, double actual, double delta)
public static void assertEquals(String message, double expected,
                                double actual, double delta)
public static void assertEquals(long expected, long actual)
public static void assertEquals(String message, long expected, long actual)
public static void assertEquals(Object expected, Object actual)
public static void assertEquals(String message, Object expected,
                                Object actual)
```

Estos métodos comparan sus parámetros `expected y actual`. Si no son iguales el método lanza una excepción del tipo `AssertionError` con el mensaje de error dado por el parámetro `message`, si existe.

Si los parámetros `expected y actual` son del tipo double, la comparación es hecha con el margen dado por el parámetro `delta`.

Si los parámetros `expected y actual` son del tipo `Object` y ambos son `null`, se consideran iguales.

Tabla 9.1 Métodos de la Clase Assert. Cont

`public static void` **`assertFalse`**`(boolean condition)`
`public static void` **`assertFalse`**`(String message, boolean condition)`
 	
Estos métodos determinan si su parámetro `condition` es falso. Si es verdadero el método lanza una excepción del tipo `AssertionError` con el mensaje de error dado por el parámetro `message`, si existe.

`public static void` **`assertTrue`**`(boolean condition)`
`public static void` **`assertTrue`**`(String message, boolean condition)`
 	
Estos métodos determinan si su parámetro `condition` es verdadero. Si es falso el método lanza una excepción del tipo `AssertionError` con el mensaje de error dado por el parámetro `message`, si existe.

`public static void` **`assertNull`**`(Object object)`
`public static void` **`assertNull`**`(String message, Object object)`
 	
Estos métodos determinan si su parámetro `object` es nulo. Si no es nulo el método lanza una excepción del tipo `AssertionError` con el mensaje de error dado por el parámetro `message`, si existe.

`public static void` **`assertNotNull`**`(Object object)`
`public static void` **`assertNotNull`**`(String message, Object object)`
 	
Estos métodos determinan si su parámetro `object` no es nulo. Si es nulo el método lanza una excepción del tipo `AssertionError` con el mensaje de error dado por el parámetro `message`, si existe.

`public static void` **`assertSame`**`(Object expected, Object actual)`
`public static void` **`assertSame`**`(String message, Object expected, Object actual)`
 	
Estos métodos determinan si los objetos dados por sus parámetros `expected y actual` son el mismo. Si no son el mismo objeto, el método lanza una excepción del tipo `AssertionError` con el mensaje de error dado por el parámetro `message`, si existe.

`public static void` **`assertNotSame`**`(Object unexpected, Object actual)`
`public static void` **`assertNotSame`**`(String message, Object unexpected, Object actual)`
 	
Estos métodos determinan si los objetos dados por sus parámetros `expected y actual` no son el mismo. Si son el mismo objeto, el método lanza una excepción del tipo `AssertionError` con el mensaje de error dado por el parámetro `message`, si existe.

`public static void` **`fail`**`()`
`public static void` **`fail`**`(String message)`
 	
Estos métodos hacen que la prueba falle lanzando una excepción del tipo `AssertionError` con el mensaje de error dado por el parámetro `message`, si existe.

Tabla 9.1 Métodos de la Clase Assert. Cont

```
public static void assertArrayEquals(byte[] expecteds, byte[] actuals)
                          throws org.junit.internal.ArrayComparisonFailure
public static void assertArrayEquals(String message, byte[] expecteds,
                                     byte[] actuals)
                          throws org.junit.internal.ArrayComparisonFailure
public static void assertArrayEquals(char[] expecteds,
                                     char[] actuals)
public static void assertArrayEquals(String message, char[] expecteds,
                                     char[] actuals)
                          throws org.junit.internal.ArrayComparisonFailure
public static void assertArrayEquals(double[] expecteds, double[] actuals,
                                     double delta)
public static void assertArrayEquals(String message, double[] expecteds,
                                     double[] actuals, double delta)
                          throws org.junit.internal.ArrayComparisonFailure
public static void assertArrayEquals(float[] expecteds, float[] actuals,
                                     float delta)
public static void assertArrayEquals(String message, float[] expecteds,
                                     float[] actuals, float delta)
                          throws org.junit.internal.ArrayComparisonFailure
public static void assertArrayEquals(int[] expecteds, int[] actuals)
public static void assertArrayEquals(String message, int[] expecteds,
                                     int[] actuals)
                          throws org.junit.internal.ArrayComparisonFailure
public static void assertArrayEquals(long[] expecteds, long[] actuals)
public static void assertArrayEquals(String message, long[] expecteds,
                                     long[] actuals)
                          throws org.junit.internal.ArrayComparisonFailure
public static void assertArrayEquals(Object[] expecteds, Object[] actuals)
public static void assertArrayEquals(String message, Object[] expecteds,
                                     Object[] actuals)
                          throws org.junit.internal.ArrayComparisonFailure
public static void assertArrayEquals(short[] expecteds, short[] actuals)
public static void assertArrayEquals(String message, short[] expecteds,
                                     short[] actuals)
                          throws org.junit.internal.ArrayComparisonFailure
```

Estos métodos comparan los arreglos dados por sus parámetros `expecteds y actuals`. Si no son iguales el método lanza una excepción del tipo `AssertionError` con el mensaje de error dado por el parámetro `message`, si existe.
Si los arreglos dados por sus parámetros `expecteds y actuals` son del tipo double, la comparación es hecha con el margen dado por el parámetro `delta`.

Pruebas Unitarias

Las pruebas unitarias deben diseñarse par que pasen cuando el método a probarse trabaje correctamente.

Pruebas Unitarias de Constructores

Normalmente no es necesario probar los constructores a menos que éstos contengan lógica, en cuyo caso si deben probarse.

Pruebas Unitarias de los Métodos de Acceso

Al igual que con los constructores, a menos que los métodos contengan lógica, no es necesario probarlos.

Pruebas Unitarias de los Métodos que Lanzan una Excepción

Si el código de un método de prueba lanza una excepción la prueba fallará. Por esa razón se debe establecer que el método de prueba relance esa excepción. La única ocación para ignorar esa regla es cuando deseamos verificar que el método bajo prueba lanza una excepción. Sin embargo si el método bajo prueba lanza la excepción, la prueba fallará en lugar de tener éxito como esperamos. Para resolver ese problema, la anotación `@Test` niene el parámetro `expected` que nos permite especificar que el método de prueba tendrá éxito si el método bajo prueba lanza la excepción. La sintaxis es:

```
@Test[(expected=nombreExcepcion.class)]
public void nombreMetodoPrueba() {
   ...
   // Aquí se invoca al método bajo prueba
   ...
}
```

Ejemplo sobre Pruebas Unitarias

Como ejemplo sobre pruebas unitarias considere las clases que simulan el mecanismo de persistencia utilizando listas para el programa sobre el amante de música y el cine, visto en el Tema 6: Colecciones. El siguiente código es el de la clase `Generos` que permite almacenar y listar los géneros de canciones y películas.

```
/*
 * Generos.java
 *
 * Creada el 30 de julio de 2008, 11:36 AM
 */
package dao;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
```

```
import objetosNegocio.Genero;
import excepciones.DAOException;

/**
 * Esta clase permite almacenar, actualizar, eliminar y consultar
 * generos de canciones y películas del programa AmanteMusica.
 * Los datos se almacenan en listas.
 *
 * @author mdomitsu
 */
public class Generos {
  protected List<Genero> generos;

  /**
   * Crea una lista del tipo ArrayList para almacenar los géneros
   */
  public Generos() {
    // Crea la lista para almacenar los géneros
    generos = new ArrayList<Genero>();
  }

  /**
   * Regresa el genero dla lista que coincida con el genero del parametro.
   * Las claves del genero de la lista y del parametro deben coincidir
   * @param genero Genero con la clave del genero a regresar
   * @return El genero cuya clave es igual a la clave del genero
   * dado por el parámetro, si se encuentra, null en caso contrario
   */
  public Genero obten(Genero genero) {
    // Busca un genero con la misma clave.
    int pos = generos.indexOf(genero);

    // Si lo encontró, regrésalo
    if(pos >= 0) return generos.get(pos);

    // si no, regresa null
    return null;
  }

  /**
   * Agrega un genero a la lista de generos
   * @param genero Genero a agregar.
   */
  public void agrega(Genero genero) {
    // Se agrega el genero
    generos.add(genero);
  }

  /**
   * Actualiza el genero de la lista que coincida con el genero del
   * parametro. Las claves del genero de la lista y del parametro deben
   * coincidir
   * @param genero Género a actualizar.
   * @throws DAOException Si el género no existe.
   */
  public void actualiza(Genero genero) throws DAOException {
    // Busca un genero con la misma clave.
```

```
    int pos = generos.indexOf(genero);

    // Si no lo encontro, no se actualiza
    if(pos < 0) throw new DAOException("Género inexistente");

    // Si lo hay, actualizalo
    generos.set(pos, genero);
  }

  /**
   * Elimina el genero dla lista que coincida con el genero del parametro.
   * Las claves del genero dla lista y del parametro deben coincidir
   * @param genero Genero a eliminar.
   * @throws DAOException Si el género no existe.
   */
  public void elimina(Genero genero) throws DAOException {
    // Si no lo encuentra, no se borra
    if(!generos.remove(genero))
        throw new DAOException("Género inexistente");
  }

  /**
   * Regresa una lista de todos los géneros.
   * @return ULista de todos los géneros
   */
  public List<Genero> lista() {
    return generos;
  }

  /**
   * Crea la lista de los géneros del mismo medio que el parámetro
   * @param tipoMedio Tipo del medio de los géneros a listar
   * @return Lista de los géneros del mismo tipo de medio que el parámetro
   */
  public List<Genero> listaMedio(char tipoMedio) {
    List<Genero> lista = new ArrayList<Genero>();

    // Recorre la lista
    for(Iterator<Genero> iterador = generos.iterator(); iterador.hasNext(); )
{
      Genero genero = iterador.next();

      // Si es el medio especificado
      if(tipoMedio == genero.getTipoMedio())
        // Agrega el medio a la lista
        lista.add(genero);
    }

    return lista;
  }
}
```

La clase de prueba para la clase `Generos` es la siguiente:

```
/*
 * GenerosTest.java
 */
package dao;

import excepciones.DAOException;
import java.util.ArrayList;
import java.util.List;
import objetosNegocio.Genero;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 * Clase de prueba de la clase Generos.
 *
 * @author mdomitsu
 */
public class GenerosTest {

    public GenerosTest() {
    }

    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }

    /**
     * Prueba del metodo obten(), de la clase Generos.
     */
    @Test
    public void testObten() {
        Generos instance = new Generos();
        Genero result;
        Genero genero1 = new Genero("GC001", "Balada", 'C');
        Genero genero2 = new Genero("GC002", "Bossanova", 'C');

        System.out.println("Prueba del método obten()");

        result = instance.obten(new Genero("GC001"));
        assertNull(result);

        instance.agrega(genero1);
        instance.agrega(genero2);

        result = instance.obten(new Genero("GC003"));
        assertNull(result);

        result = instance.obten(new Genero("GC001"));
        assertEquals(genero1, result);
```

```
        result = instance.obten(new Genero("GC002"));
        assertEquals(genero2, result);
    }

    /**
     * Prueba del metodo agrega(), de la clase Generos.
     */
    @Test
    public void testAgrega() {
        Generos instance = new Generos();
        Genero result;
        Genero genero1 = new Genero("GC001", "Balada", 'C');
        Genero genero2 = new Genero("GC002", "Bossanova", 'C');

        System.out.println("Prueba del método agrega()");

        instance.agrega(genero1);
        result = instance.obten(new Genero("GC001"));
        assertEquals(genero1, result);

        instance.agrega(genero2);
        result = instance.obten(new Genero("GC002"));
        assertEquals(genero2, result);
    }

    /**
     * Prueba del metodo actualiza(), de la clase Generos.
     */
    @Test(expected=DAOException.class)
    public void testActualiza() throws Exception {
        Generos instance = new Generos();

        System.out.println("Prueba del método actualiza()");

        instance.actualiza(new Genero("GC001", "Balada", 'C'));
    }

     /**
     * Prueba del metodo actualiza(), de la clase Generos.
     */
    @Test
    public void testActualiza2() throws Exception {
        Generos instance = new Generos();
        Genero result;
        Genero genero1 = new Genero("GC001", "Balada", 'C');
        Genero genero2 = new Genero("GC002", "Bossanova", 'C');

        System.out.println("Prueba del método actualiza()");

        instance.agrega(genero1);
        instance.agrega(genero2);

        Genero genero3 = new Genero("GC001", "Romantica", 'C');
        Genero genero4 = new Genero("GC002", "Samba", 'C');

        instance.actualiza(genero3);
```

```
        result = instance.obten(new Genero("GC001"));
        assertEquals(genero3, result);

        instance.actualiza(genero4);
        result = instance.obten(new Genero("GC002"));
        assertEquals(genero4, result);
    }

    /**
     * Prueba del metodo elimina(), de la clase Generos.
     */
    @Test(expected=DAOException.class)
    public void testElimina() throws Exception {
        Generos instance = new Generos();
        System.out.println("Prueba del método elimina()");

        instance.elimina(new Genero("GC001", "Balada", 'C'));
    }

    /**
     * Prueba del metodo elimina(), de la clase Generos.
     */
    @Test
    public void testElimina2() throws Exception {
        Generos instance = new Generos();
        Genero result;
        Genero genero1 = new Genero("GC001", "Balada", 'C');
        Genero genero2 = new Genero("GC002", "Bossanova", 'C');

        System.out.println("Prueba del método actualiza()");

        instance.agrega(genero1);
        instance.agrega(genero2);

        instance.elimina(genero2);
        result = instance.obten(new Genero("GC002"));
        assertNull(result);

        instance.elimina(genero1);
        result = instance.obten(new Genero("GC001"));
        assertNull(result);
    }

    /**
     * Prueba del metodo lista(), de la clase Generos.
     */
    @Test
    public void testLista() {
        Generos instance = new Generos();
        List<Genero> generos = new ArrayList<Genero>();
        List<Genero> result;

        System.out.println("Prueba del método lista()");

        result = instance.lista();
        assertEquals(generos, result);
```

```
        Genero genero1 = new Genero("GC001", "Balada", 'C');
        Genero genero2 = new Genero("GC002", "Bossanova", 'C');
        Genero genero3 = new Genero("GC001", "Balada", 'C');
        Genero genero4 = new Genero("GC002", "Bossanova", 'C');

        instance.agrega(genero1);
        instance.agrega(genero2);
        generos.add(genero3);
        generos.add(genero4);

        result = instance.lista();
        assertEquals(generos, result);
    }

    /**
     * Prueba del metodo listaMedio(), de la clase Generos.
     */
    @Test
    public void testListaMedio() {
        Generos instance = new Generos();
        List<Genero> generos = new ArrayList<Genero>();
        List<Genero> result;
        System.out.println("Prueba del método listaMedio()");

        result = instance.lista();
        assertEquals(generos, result);

        Genero genero1 = new Genero("GC001", "Balada", 'C');
        Genero genero2 = new Genero("GP003", "Comedia", 'P');
        Genero genero3 = new Genero("GC001", "Balada", 'C');
        Genero genero4 = new Genero("GP003", "Comedia", 'P');

        instance.agrega(genero1);
        instance.agrega(genero2);
        generos.add(genero3);
        result = instance.listaMedio('C');
        assertEquals(generos, result);

        generos = new ArrayList<Genero>();
        generos.add(genero4);
        result = instance.listaMedio('P');
        assertEquals(generos, result);
    }
}
```

Pruebas Unitarias Parametrizadas

En ocaciones se requiere ejecutar una prueba repetidamente para un conjunto de valores. JUnit 4 nos permite implementar y ejecutar una prueba que se ejecute varias veces una para cada valor establecido. El procedimiento para crear una clase de prueba parametrizada es el siguiente:

1. Anotar la clase de prueba con la anotación `@RunWith`, cuya sintaxis es:

2. Agrégale a la clase un atributo para cada parámetro requerido por la prueba, incluyendo el resultado.

3. Crea un constructor que inicialice los atributos a los valores de los parámetros.

4. Crea un método estático que sirva para alimentar las pruebas del tipo `Collection` y agréguele la anotación `@Parameters`.

5. Crea el método de prueba parametrizado:

La sintaxis de la clase de prueba parametrizada es la siguiente:

```
// Anota la clase de prueba
@RunWith(Parameterized.class)
public class NombreClasePrueba {
   // Atributos para los parámetros
   tipo nombreAtributo1;
   [tipo nombreAtributo2]...
   tipo resultado;

   // Constructor que inicializa los parámetros
   public NombreClasePrueba(tipo nomParametro1,
                            [tipo nombreParametro2],
                            ...
                            tipo nomParametroRes) {
      nombreAtributo1 = nomParametro1;
      nombreAtributo2 = nomParametro2;
      ...
      resultado = nomParametroRes;
   }

   // Método alimentador
   @Parameters
   public static Collection data() {
      return Arrays.asList(new Object[][]{
             {valorParametro11, valorParametro21, ...,
              valorParametroRes1},
             {valorParametro12, valorParametro22, ... ...,
              valorParametroRes2},
             ...,
      });
   }

   // método de prueba parametrizado
   @Test
    public void nombreMetodoPrueba() {
        System.out.println("Prueba parametrizada ... ");
        NombreClaseAProbar instance = new NombreClaseAProbar();

        assertEquals(resultado, instance.suma(nombreAtributo1,
                                              nombreAtributo2
                                              ...);
    }
}
```

El siguiente código es un ejemplo de una clase de prueba parametrizada:

```
package pruebas;

import java.util.Arrays;
import java.util.Collection;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;

@RunWith(Parameterized.class)
public class CalculadoraTest {
    double parametro1;
    double parametro2;
    double resultado;

    public CalculadoraTest(double parametro1, double parametro2,
                           double resultado) {
        this.parametro1 = parametro1;
        this.parametro2 = parametro2;
        this.resultado = resultado;
    }


    @Parameters
    public static Collection data() {
        return Arrays.asList(new Object[][]{
                {0, 0, 0},
                {5, 5, 10},
                {10, 10, 20},
        });
    }

   @Test
    public void testSumaParametrizada() {
        System.out.println("Prueba parametrizada del método suma ");
        Calculadora instance = new Calculadora();

        assertEquals(resultado, instance.suma(parametro1, parametro2), 0.0);
    }
}
```

Codigo que No Debe Probarse Usando Pruebas Unitarias

No todo código debe o puede probarse usando pruebas unitarias. No quiere decir que no debe probarse, sino que debe probarse con otro tipo de pruebas como de integración o funcionales. No debe usarse pruebas unitarias a código que:

- Se comunica con base de datos.
- Se comunica usando una red.
- Emplea un sistema de archivos.
- No puede probarse simultáneamente que otro código.
- Se requiere hacer algo especial al ambiente para ejecutar la prueba, como modificar archivos de configuración.

Este tipo de código se dice que es una dependencia externa y sobre el cual no se tiene control.

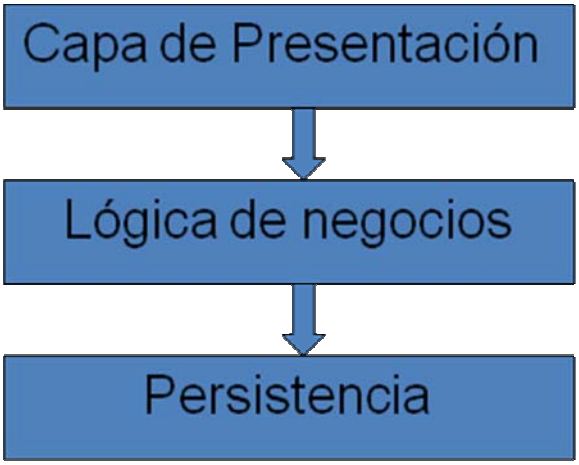
Stubs

Suponga que deseamos probar una clase que depende de otra clase sobre la cual no tengamos control o de una clase que aún no ha sido escrita. Debido a eso, nuestra prueba no puede controlar lo que esa dependencia externa le regresa al código bajo prueba.

En estos casos podemos substituir esa dependencia externa por un componente de software, llamado stub, que simule el comportamiento de la dependencia externa pero en una forma controlada.

Un stub es un reemplazo controlable para una dependencia externa en el sistema bajo prueba. El uso del stub nos permite probar el código sin tener que lidear con las dependencias externas.

Como ejemplo del uso de stubs para para realizar las pruebas, considere un sistema contruido en capas, como se muestra en la siguiente figura:



En un diseño en capas, el código de cada capa interactua con la capa inferior a través de una interfaz fien definida. Los métodos de una capa sólo pueden invocar a los métodos de una capa inferior y no a los de una capa superior. En este sistema la capa superior: la capa de presentación contiene el código que representa la interfaz del

usuario de la aplicación. La capa intermedia: la capa de la lógica de los negocios implementa las reglas de los negocios de la aplicación y la capa inferior: la capa de persistencia encapsula el mecanismo empleado para persistir (almacenar permanentemente) los datos. Ese mecanismo podría emplear archivos o una base de datos.

Para este sistema, el código de las capas de presentación y de la capa de persistencia no pueden probarse usando pruebas unitarias debido a que son dependencias externas. Para probar el código de la capa de la logica de negocios debemos crear un stub que actue como el mecanismo de persistencia pero que no tenga dependencias externas. Por ejemplo podríamos simular el almacenamiento de datos utilizando listas o arreglos.

Como ejemplo sobre pruebas unitarias que usan stubs considere la siguiente clase que actúa como fachada a las clases que implementan el mecanismo de persistencia para el problema del amante de la música y el cine. Como los métodos del mecanismo de persistencia no pueden probarse usando pruebas unitarias, para probar los métodos de la fachada debemos simular el mecanismo de persistencia. En este caso usaremos las clases vistas en el Tema 6: Colecciones como stubs.

```
/*
 * PersistenciaListas.java
 *
 * Created on 15 de septiembre de 2007, 12:21 PM
 */

package persistencia;

import java.util.List;

import objetosServicio.*;
import objetosNegocio.*;
import excepciones.*;
import interfaces.IPersistencia;
import dao.*;

/**
 * Esta clase implementa la interfaz IPersistencia que simula el mecanismo
 * de persistencia usando listas del programa AmanteMusica
 *
 * @author mdomitsu
 */
public class PersistenciaListas implements IPersistencia {
    private Generos catalogoGeneros;
    private Canciones catalogoCanciones;
    private Peliculas catalogoPeliculas;

    /**
     * Este constructor crea los objetos con los arreglos para
     * almacenar los géneros, canciones y películas
     */
    public PersistenciaListas() {
```

```
      // Crea un objeto del tipo Generos para accesar a la tabla canciones
      catalogoGeneros = new Generos();

      // Crea un objeto del tipo Canciones para accesar a la tabla canciones
      catalogoCanciones = new Canciones();

      // Crea un objeto del tipo Peliculas para accesar a la tabla peliculas
      catalogoPeliculas = new Peliculas();
    }

...

  /**
   * Obtiene un género del catálogo de géneros
   * @param genero Género a obtener
   * @return El género si existe, null en caso contrario
   * @throws PersistenciaException Si no se puede acceder al
   * catálogo de géneros.
   */
  public Genero obten(Genero genero) throws PersistenciaException {
    // Obten el género
    return catalogoGeneros.obten(genero);
  }

  /**
   * Agrega un género al catálogo de géneros. No se permiten géneros
   * con claves repetidas
   * @param genero Género a agregar
   * @throws PersistenciaException Si no se puede agregar el género al
   * catálogo de géneros.
   */
  public void agrega(Genero genero) throws PersistenciaException {
    Genero generoBuscado;

    // Busca el género en el arreglo con la misma clave.
    generoBuscado = catalogoGeneros.obten(genero);

    // Si lo hay, no se agrega al arreglo
    if(generoBuscado != null)
      throw new PersistenciaException("Género repetido");

    // Agrega el nuevo género al catálogo
      catalogoGeneros.agrega(genero);
  }

  /**
   * Actualiza un género del catálogo de géneros
   * @param genero Género a actualizar
   * @throws PersistenciaException Si no se puede actualizar el género del
   * catálogo de géneros.
   */
  public void actualiza(Genero genero) throws PersistenciaException {
    // Actualiza el género del catálogo
    try {
      catalogoGeneros.actualiza(genero);
    }
    catch(DAOException pae) {
```

```
      throw new PersistenciaException("No se puede actualizar el género",
                                      pae);
    }
  }

  /**
   * Elimina un género del catálogo de géneros
   * @param genero Género a eliminar
   * @throws PersistenciaException Si no se puede eliminar el género del
   * catálogo de géneros.
   */
  public void elimina(Genero genero) throws PersistenciaException {
    // Elimina el género del catálogo
    try {
      catalogoGeneros.elimina(genero);
    }
    catch(DAOException pae) {
      throw new PersistenciaException("No se puede eliminar el género", pae);
    }
  }
  ...

  /**
   * Obtiene una lista de todos los géneros
   * @return Lista con la lista de todos los géneros.
   */
  public List<Genero> consultaGeneros() {
    // Regresa el vector con la lista de géneros
    return catalogoGeneros.lista();
  }

  /**
   * Obtiene una lista de los géneros de canciones
   * @return Lista con la lista de los géneros canciones
   */
  public List<Genero> consultaGenerosCanciones() {
    // Regresa el vector con la lista de géneros de canciones
    return catalogoGeneros.listaMedio('C');
  }

  /**
   * Obtiene una lista de los géneros de películas
   * @return Lista con la lista de los géneros películas
   */
  public List<Genero> consultaGenerosPeliculas() {
    // Regresa el vector con la lista de géneros de películas
    return catalogoGeneros.listaMedio('P');
  }
}
```

La clase de prueba para la clase que implementa la fachada es la siguiente:

```
/*
 * PersistenciaListasTest.java
 */
```

```
package persistencia;

import excepciones.PersistenciaException;
import objetosNegocio.Cancion;
import objetosNegocio.Genero;
import objetosNegocio.Pelicula;
import objetosServicio.Fecha;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 * Esta clase de prueba contiene los metodos para las pruebas unitarias
 * de la clase PersistenciaListas que simula el mecanismo de persistencia
 * usando listas del programa AmanteMusica.
 *
 * @author mdomitsu
 */
public class PersistenciaListasTest {

    public PersistenciaListasTest() {
    }

    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }

    ...

    /**
     * Prueba del metodo agrega(genero) de la clase PersistenciaListas.
     */
    @Test(expected=PersistenciaException.class)
    public void testAgrega_Genero() throws Exception {
        PersistenciaListas instance = new PersistenciaListas();

        System.out.println("Prueba del metodo agrega(genero)");

        Genero genero = new Genero("GC001", "Balada", 'C');
        instance.agrega(genero);
        instance.agrega(genero);
    }

    /**
     * Prueba del metodo agrega(genero) de la clase PersistenciaListas.
     */
    @Test
    public void testAgrega2_Genero() throws Exception {
        PersistenciaListas instance = new PersistenciaListas();
        Genero result;

        System.out.println("Prueba del metodo agrega(genero)");
```

```
        Genero genero = new Genero("GC001", "Balada", 'C');
        instance.agrega(genero);

        result = instance.obten(new Genero("GC001"));
        assertEquals(genero, result);
    }

    /**
     * Prueba del metodo actualiza(genero) de la clase PersistenciaListas.
     */
    @Test(expected=PersistenciaException.class)
    public void testActualiza_Genero() throws Exception {
        PersistenciaListas instance = new PersistenciaListas();

        System.out.println("Prueba del metodo actualiza(genero)");

        Genero genero = new Genero("GC001", "Balada", 'C');

        instance.actualiza(genero);
    }

    /**
     * Prueba del metodo actualiza(genero) de la clase PersistenciaListas.
     */
    @Test
    public void testActualiza2_Genero() throws Exception {
        PersistenciaListas instance = new PersistenciaListas();
        Genero result;

        System.out.println("Prueba del metodo actualiza(genero)");

        Genero genero = new Genero("GC001", "Balada", 'C');
        instance.agrega(genero);

        Genero genero2 = new Genero("GC001", "Bolero", 'C');
        instance.actualiza(genero2);

        result = instance.obten(new Genero("GC001"));
        assertEquals(genero2, result);
    }

    /**
     * Prueba del metodo elimina(genero) de la clase PersistenciaListas.
     */
    @Test(expected=PersistenciaException.class)
    public void testElimina_Genero() throws Exception {
        PersistenciaListas instance = new PersistenciaListas();

        System.out.println("Prueba del metodo elimina(genero)");

        Genero genero = new Genero("GC001", "Balada", 'C');

        instance.elimina(genero);
     }

    /**
```

```
     * Prueba del metodo elimina(genero) de la clase PersistenciaListas.
     */
    @Test
    public void testElimina2_Genero() throws Exception {
        PersistenciaListas instance = new PersistenciaListas();
        Genero result;

        System.out.println("Prueba del metodo elimina(genero)");

        Genero genero = new Genero("GC001", "Balada", 'C');
        instance.agrega(genero);
        instance.elimina(genero);

        result = instance.obten(new Genero("GC001"));
        assertNull(result);
    }
}
```