Práctica 1

PARTE 1:
INTRODUCCIÓN A LA PRUEBA UNITARIA DE SOFTWARE
Uso de las bibliotecas JUNIT para la prueba unitaria en el lenguaje Java

Tabla de contenidos

1. ¿Qué es la Prueba Unitaria?

La prueba de programas es una aproximación dinámica a la verificación de los mismos, en la cual se ejecuta el software con unos datos o casos de prueba para analizar el buen funcionamiento de los requisitos esperados del programa. Podemos encontrar una descripción exhaustiva de las técnicas de prueba unitaria en (Morell, 1992).

> El término *verificación* de programas hace referencia a cualquier tipo de proceso que trate de comprobar si un determinado programa, módulo o componente software cumple con los requisitos que se esperan de él. De esta manera, engloba muchos tipos de técnicas, incluyendo algunas que no requieren la ejecución – y ni siquiera la implementación – del programa, como las técnicas de verificación formal, así como otras que basan su comprobación en el análisis de la ejecución del programa implementado. De esta manera, podemos decir que la *prueba* es un tipo de verificación que se basa en la ejecución del código.

La prueba unitaria es la prueba de un módulo concreto dentro de un software que incluirá muchos otros módulos. Un módulo es una unidad de código que sirve como bloque de construcción para la estructura física de un sistema. El concepto de "módulo" se debe interpretar en términos del lenguaje de programación que estemos utilizando. Por ejemplo, podemos considerar que una clase Java o C++ es un módulo, pero también lo es un módulo del lenguaje Modula-2 o una unidad en TurboPascal.

En lenguajes más antiguos, podemos considerar que un subprograma o un conjunto de subprogramas relacionados es un módulo, aunque normalmente se considera que un módulo se compila de manera separada al resto de los módulos que conforman el sistema.

2. Introducción a la familia de *frameworks* xUnit para la Prueba Unitaria

En los últimos años, aparecido una metodología de desarrollo de programas denominada *Extreme Programming* (XP) (Beck 1999), que hace un énfasis especial en las prácticas y técnicas de prueba unitaria. Como resultado de ese énfasis, se han creado una serie de *frameworks* para ayudar a realizar pruebas unitarias en diferentes lenguajes. Al conjunto de esos *frameworks* se les denomina xUnit.

> Un *framework* o "marco de clases" puede definirse como "una colección de clases que proporciona un conjunto de servicios para un dominio particular, exportando un cierto número de clases y mecanismos de utilización que pueden utilizarse tal cual o adaptarse para una aplicación concreta"

Aunque en este documento nos centramos en JUNIT, que es "el XUNIT para Java", los conceptos son muy similares a los que nos encontramos en los *frameworks* XUNIT para otros lenguajes (incluyendo a C++, Delphi, Smalltalk y Visual Basic). Podemos encontrar esos *frameworks* en la siguiente dirección:

`http://www.xprogramming.com/software.htm`

Los conceptos fundamentales de los frameworks XUNIT son los casos de prueba (*test cases*), las suites de prueba (*test suites*). Los casos de prueba son clases (o módulos, más en general) con una serie de métodos que ejecutan los métodos de una determinada clase (o módulo), que es el objeto de la prueba. De esta manera, cada clase tiene asociada otra clase que sirve para probarla – excepto clases triviales para las que no merezca la pena codificar un caso de prueba.

> De hecho, esta forma de prueba unitaria no es más que la sistematización de los "programas de prueba", muchas veces de "usar y tirar" que acostumbran a escribir los programadores. La diferencia es que al construir casos de prueba con el *framework*, los programas de prueba están escritos de una manera que permite su ejecución y re-ejecución (pruebas de regresión) automatizada.

Este enfoque tiene las siguientes características:

- Las pruebas son también programas, y no simplemente especificaciones de datos de entrada y datos de salida esperados, de manera que pueden ejecutarse automáticamente, y puede quedar automatizada también la comprobación de si se ha pasado la prueba o no.
- Los casos de prueba no se descartan, sino que son un producto más del desarrollo, junto con el código fuente.
- Los casos se pueden estructurar en colecciones de manera que se pueden ejecutar las pruebas relativas sólo a una parte del sistema o a todo el sistema.
- Según se va desarrollando la aplicación, se va generando una batería de pruebas más y más completa que sirve para realizar pruebas de regresión.

Esta última característica es especialmente importante desde el punto de vista de la calidad del software resultante. Cuando introducimos un cambio en una clase que ya ha sido probada, existe la posibilidad de que ese cambio genere un error en otra parte de la clase como efecto colateral. Para garantizar que el cambio no ha generado este tipo de efectos, debemos re-ejecutar todas las pruebas que se habían hecho anteriormente sobre la clase. A este proceso se le denomina prueba de regresión.

La estructuración de los casos de prueba se realiza mediante suites. Un suite agrupa un conjunto de casos de prueba sobre clases que están funcionalmente relacionadas. Por ejemplo, podemos agrupar en una suite todos los casos de prueba de clases relacionadas con la matriculación de alumnos, y en otra todas las relacionadas con la contratación del profesorado. Ambas suites pueden a su ver estructurarse formando un árbol. De esta manera, las pruebas de nuestro

sistema se estructuran en forma de árbol, siendo las hojas los casos de prueba, y permitiendo ejecutar cualquier subárbol a partir de cualquier nodo intermedio.

Por último, cabe destacar que en la metodología XP es el mismo desarrollador de la clase el que codifica el caso de prueba unitaria, y lo hace como actividad inmediata al desarrollo de la clase e incluso paralela, de forma que garantizamos que se hacen pruebas sobre todo el código que se escribe.

Instalación y prueba de JUnit

El framework JUnit es un conjunto de clases Java que se encuentran dentro de un fichero con extensión "JAR". Para descargar una versión de JUnit puede acudirse a www.junit.org.

Para probarlo:

1. Descarga el fichero junit.jar al directorio donde estén las clases de prueba.
2. Crea un fichero PalindromaTest.java con el contenido del fichero que encontrarás más abajo.
3. Para compilar los casos de prueba, hay que añadir junit.jar al CLASSPATH. Puede hacerse en una misma sentencia, por ejemplo:

   javac -classpath .;./junit.jar PalindromaTest.java
4. Para ejecutar las pruebas, se puede utilizar uno de los programas que proporciona Junit, por ejemplo:

   java -classpath .;./junit.jar junit.swingui.TestRunner

Clase de prueba de Junit

```
import junit.framework.TestCase;
public class PalindromaTest extends TestCase{
  public PalindromaTest(String nombre){
        super(nombre);
  }
  public void testEsPalidroma(){
        String s1= new String
                    ("Dabale  arroz a la zorra el abad").
                          toUpperCase();
        String s2= new String(" Dabale arroz a la zorra el abad
");
        assertTrue (new Palindroma(s1).esPalindroma());
        assertTrue (!new Palindroma(s2).esPalindroma());
  }
}
```

3. Uso de JUNIT para la Prueba Unitaria en Java

3.1. ¿Qué es JUNIT?

JUnit es un *framework* para escribir casos de prueba cuya ejecución y comprobación puede realizarse de manera automatizada. JUnit es un conjunto de clases Java, y sirve para la prueba de clases en este mismo lenguaje.

JUnit es un software de "fuente abierto", y puede obtenerse (tanto su código fuente como el *bytecode* Java y la documentación en formato *javadoc*) de la dirección Web:

http://sourceforge.net/projects/junit/

También se puede obtener de http://www.junit.org/, donde además se recopila una gran cantidad de artículos y software relacionado. Por ejemplo, el artículo "JUnit Test Infected: Programmers Love Writing Tests" nos introduce en los conceptos básicos de JUnit. Puede encontrarse en:

http://junit.sourceforge.net/doc/testinfected/testing.htm

En el resto de esta sección vamos a introducir algunas de las técnicas esenciales de prueba unitaria con JUnit. Hay que tener en cuenta que el *framework* es amplio y no vamos a tratar más que un subconjunto, el resto puede consultarse en la documentación del mismo.

3.2. Un primer ejemplo de uso de JUNIT

Vamos a considerar una clase muy sencilla como primer ejemplo. Probablemente, esta clase no requeriría un caso de prueba por su extremada simplicidad, pero la utilizaremos por motivos pedagógicos.

Supongamos que queremos crear una clase para manipular números complejos. Una primera versión de la clase (eliminamos los comentarios por brevedad) podría ser la siguiente:

```
public class Complejo{
        private float _parteReal;
        private float _parteImaginaria;

        public Complejo(float parteReal, float parteImaginaria){
         _parteReal = parteReal; _parteImaginaria = parteImaginaria;
        }
        public float getParteReal(){
                return _parteReal;
        }

        public float getParteImaginaria(){
                return _parteImaginaria;
        }

        public Complejo sumar(Complejo c){
                return new Complejo( this.getParteReal() +
                                     c.getParteReal(),
                                     this.getParteImaginaria() +
                                     c.getParteImaginaria()  );
        }
}
```

Se momento sólo tenemos una funcionalidad relevante, la de sumar dos complejos, obteniendo un tercero. Ahora utilizaremos JUnit para construir un caso de prueba.

En JUnit, los casos de prueba son clases que derivan de la clase TestCase, e implementan métodos sin parámetros de nombre testXXX, donde XXX es una descripción de lo que está probando ese método.

```
import junit.framework.TestCase;
public class ComplejoTest extends TestCase{
    public ComplejoTest(String nombre){
          super(nombre);
    }
    public void testSumaComplejos(){
          // 1.
          Complejo c1 = new Complejo(3, 5);
          Complejo c2 = new Complejo(1, -1);
          // 2.
          Complejo resultado = c1.sumar(c2);
          // 3.
          assertTrue(resultado.getParteReal()==4);
          assertTrue(resultado.getParteImaginaria()==4);
    }
}
```

Como convención se suele poner como nombre de la clase que implementa el caso de prueba el mismo que la clase probada con el sufijo Test. Lo único necesario para compilar y ejecutar los casos de prueba es tener indicar dónde se encuentra el fichero junit.jar que contiene las clases de JUnit, bien mediante la variable de entorno CLASSPATH o especificándolo en los comandos, por ejemplo: `javac -classpath .;.\junit.jar ComplejoTest.java`

En el código anterior debemos notar lo siguiente:

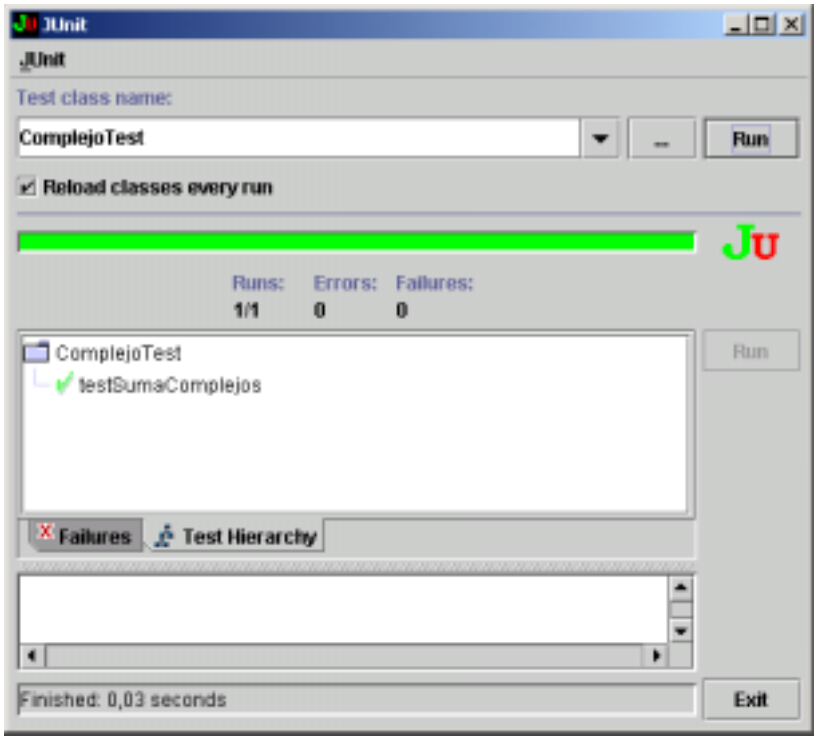
- El constructor que hemos puesto en esa clase es obligatorio, y sirve para dar un nombre significativo al caso de prueba, para identificarlo en su posterior ejecución.
- Tenemos un método de prueba que realiza los siguientes pasos:
- Crea los objetos implicados en la prueba, en nuestro caso, c1 y c2. A estos objetos se les suele denominar *fixture*.
- Código que ejecuta operaciones sobre los objetos en el *fixture*.
- Código que verifica que los resultados obtenidos son los esperados. Para ello, utilizamos *aserciones* que se encuentran en la clase Assert, que es superclase de TestCase. Por ejemplo, assertTrue toma una expresión booleana como parámetro y produce una excepción en caso de que no se evalúe a true, produciendo así un **fallo del caso de prueba**.

Para terminar nuestro ejemplo, sólo nos queda ver cómo ejecutar el caso de prueba para comprobar si el código es o no correcto. Para ello, debemos utilizar un **ejecutor de pruebas** (*test runner*). JUnit proporciona algunos como junit.textui.TestRunner que muestra los resultados en la consola modo texto o junit.swingui.TestRunner que muestra una aplicación de interfaz gráfica con los resultados de la prueba.

Por ejemplo, si ejecutamos:

```
java -classpath .;.\junit.jar junit.swingui.TestRunner
```

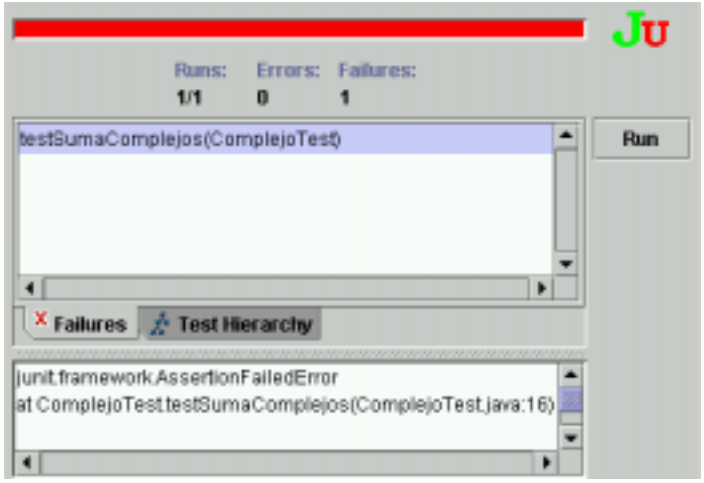
Nos aparecerá una ventana, en la cual tendremos que escribir el nombre del caso de prueba que queremos ejecutar (o seleccionarlo de una lista). A continuación, pulsaremos el botón “Run” y se nos mostrará el resultado de la ejecución de la prueba.



En la anterior interfaz es importante destacar:

- La ficha “Test Hierarchy” que nos permite seleccionar uno de los métodos del caso de prueba (en nuestro caso sólo había uno).
- La ficha “Failures” nos da información sobre los fallos del método de prueba seleccionado, si es que hubo algún fallo.
- Se proporciona un resumen de los métodos que se han ejecutado correctamente y los que han fallado.

Si la prueba hubiese fallado porque hubiéramos cometido un error de programación, se nos mostraría el lugar del caso de prueba en el que falla la aserción, como se puede ver en la siguiente figura.



En ocasiones, la invocación a un ejecutor de pruebas se incluye en el propio caso de prueba escribiendo un método main. Por ejemplo, podríamos añadir el siguiente método a la clase ComplejoTest.

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

De modo que se puede invocar al ejecutor mediante la clase que implementa el caso de prueba:

```
   java -classpath .;.\junit.jar ComplejoTest
```

Nótese que se pasa el nombre del caso de prueba obtenido mediante la clase ComplejoTest.

3.3. Inicialización y Terminación

Supongamos ahora que añadimos una operación de división a la clase Complejo anterior.

```
    public Complejo dividir(Complejo c){
         // (a + bi) / (c + di)  = ((ac+bd) + (bc-ad)i ) /(c^2 + d^2)
            float ddor = c.getParteReal()*c.getParteReal()
                              + c.getParteImaginaria()*c.getParteImaginaria();
            // Java no lanza excepcion por división por cero en floats:
            if (ddor==0) throw new ArithmeticException("Division por cero");
            float nuevaReal = this.getParteReal()*c.getParteReal() +
                           this.getParteImaginaria()*c.getParteImaginaria();
            float nuevaImag = this.getParteImaginaria()*c.getParteReal()
                                 - this.getParteReal()*c.getParteImaginaria();
            return new Complejo(nuevaReal / ddor, nuevaImag / ddor);
    }
```

Ahora tendríamos que añadir el correspondiente método para probarla. En este caso, haremos dos métodos, uno para probar una división normal y otro para dividir. El método de prueba podría ser el siguiente:

```
  public void testDivisionComplejos(){
          // 1.
          Complejo c1 = new Complejo(3, 5);
          Complejo c2 = new Complejo(1, -1);
          // 2.
          Complejo resultado = c1.dividir(c2);
          // 3.
          assertTrue(resultado.getParteReal()==-1);
          assertTrue(resultado.getParteImaginaria()==4);
  }
```

Como vemos, la creación de los objetos se repite. En estas situaciones, se puede sobrescribir el método setUp de TestCase para poner allí el código común. Esté método será invocado cuando comiencen a ejecutarse los métodos de prueba de la clase.

Después de esa reestructuración tendríamos:

```
public class ComplejoTest2 extends TestCase{

//...
    private Complejo c1;
    private Complejo c2;
    public void setUp(){
           c1 = new Complejo(3, 5);
           c2 = new Complejo(1, -1);
    }
    public void testSumaComplejos(){
          Complejo resultado = c1.sumar(c2);
          assertTrue(resultado.getParteReal()==4);
          assertTrue(resultado.getParteImaginaria()==4);
    }

    public void testDivisionComplejos(){
          Complejo resultado = c1.dividir(c2);
          assertTrue(resultado.getParteReal()==-1);
          assertTrue(resultado.getParteImaginaria()==4);
    }
//...
}
```

Existe también un método tearDown en TestCase que podría rescribirse para librerar el *fixture* creado en setUp. En nuestro ejemplo no es necesario, ya que el recolector de basura se encargará de liberar los objetos c1 y c2. Podría ser necesario, por ejemplo, cuando la inicialización de los datos de prueba implicase insertar registros en una base de datos. En ese caso, el método tearDown debería tener el código para borrarlos, ya que la ejecución de las pruebas no debe "dejar rastro".

3.4. Pruebas que producen lanzamiento de excepciones.

Vamos a ver un caso particular de prueba cuya ejecución correcta debe terminar en una excepción. Este es el caso de una división por cero en la división de complejos (nótese que en ocasiones es necesario codificar más de un método de prueba para un mismo método de la clase).

Codificamos el método de prueba correspondiente de la siguiente manera.

```
public void testDivisionPorCeroComplejos(){
         Complejo c3 = new Complejo(0, 0);
         try{
           Complejo resultado = c1.dividir(c3);
           fail("Deberia lanzar una excepcion");
         } catch (ArithmeticException e){
         }
}
```

La idea es que el caso de prueba tiene éxito si se lanza la excepción, mientras que en caso de que no se lance, habrá que provocar el fallo del caso de prueba. Esto se realiza con el método fail heredado de TestCase, que, lógicamente, se pondrá detrás de la sentencia que debe producir la excepción.

3.5. Suites de Pruebas

Podemos agrupar un conjunto de métodos de prueba en una suite. Para hacerlo en JUnit, se debe incorporar a la clase un método suite() de la clase TestCase que devuelva una instancia de la clase TestSuite. Las suites pueden estructurarse en árbol para dividir las pruebas en diferentes categorías.

Siguiendo el ejemplo anterior, podríamos modificar la clase ComplejoTest para crear una suite con dos sub-suites y los correspondientes casos de prueba dentro de cada una de estas últimas.

```
import junit.framework.TestCase;
import junit.framework.TestSuite;
import junit.framework.Test;

public class ComplejoTest extends TestCase{
    //...
    public static Test suite(){
           TestSuite suite = new TestSuite("suiteRaiz");

           TestSuite suite1 = new TestSuite("suiteSuma");
                suite1.addTest(new ComplejoTest("testSumaComplejos"));
           TestSuite suite2 = new TestSuite("suiteDivision");
            suite2.addTest(new ComplejoTest("testDivisionComplejos"));
            suite2.addTest(new ComplejoTest2("testDivisionPorCeroComplejos"));
           suite.addTest(suite1);
           suite.addTest(suite2);
           return suite;
   }

    public static void main(String args[]) {
         junit.swingui.TestRunner.run(ComplejoTest.class);
    }


}
```

El método estático suite construye la suite. Este método es el que invoca implícitamente la clase TestRunner cuando invocamos a run.

En la interfaz del ejecutor aparecerá una vista de árbol con la estructura de la suite:

La ficha "Test Hierarchy" del ejecutor de interfaz gráfica nos permite seleccionar una suite concreta, permitiendo hacer ejecución selectiva de pruebas.

Cuadro Resumen: Pasos para escribir un caso de prueba

1. Definir una subclase de TestCase.
2. Sobreescribir el método setUp() para inicializar el/los objetos implicados en la prueba (si es necesario).
3. Sobreescribir el método tearDown() para liberar el/los objetos implicados en la prueba (si es necesario).
4. Definir uno o más métodos con signatura testXXX() que prueban las diferentes funcionalidades y casos de la clase.
5. Opcionalmente, definir un método de clase suite() que crea una TestSuite a la que se añaden todos los métodos testXXX() del TestCase.
6. Definir un método main() que ejecuta el TestCase (o invocar un ejecutor de tests y especificar el caso de prueba o la suite).

3.6. Algunos consejos sobre el uso de JUnit

Se aconseja poner las clases que implementan los casos de prueba junto a las propias clases, es decir, en su mismo paquete Java. Así, se facilita la localización de las mismas (y además los casos de prueba tienen acceso a los atributos y métodos con visibilidad de paquete, que en ocasiones puede interesar probar también).

4. Referencias

(Morell, 1992) Morell, L.J., Deimel, L.E., Unit Analysis and Testing, SEI Curriculum Module SEI-CM-9.2.0, June 1992. Software Engineering Institute, Carnegie Mellon University, http://www.sei.cmu.edu/publications/documents/cms/cm.009.html

(Beck 1999) Beck, K., Extreme Programming Explained: Embrace Change, The XP Series, Addison Wesley, 1999.