## Tema 3, apéndice 1

# Pruebas unitarias con JUnit

**Desarrollo de aplicaciones multiplataforma** **Carlos Alberto Cortijo Bon**

# Índice

# 1. JUnit y TDD o desarrollo orientado a pruebas

Junit es un *framework* para Java para pruebas unitarias. Permite definir, organizar, gestionar y ejecutar automáticamente conjuntos de pruebas unitarias. Un conjunto de pruebas unitarias para una clase o para un programa se refleja en un programa de pruebas unitarias. Cuando se ejecuta este programa, se verifica si, para cada condición de prueba, el resultado obtenido es igual que el esperado.

JUnit sigue la filosofía del ***Test Driven Development*** o desarrollo orientado a pruebas. Las pruebas o el *testing* se realizan conjuntamente con el desarrollo. Para cada clase se crea una clase de prueba. Y las pruebas se realizan de manera automática. Junto con la funcionalidad del programa, se desarrollan las pruebas.

De hecho, las pruebas podrían desarrollarse antes incluso que la propia funcionalidad del programa. Si después de un cambio que se planea hacer en una clase o un programa, se espera un determinado resultado en una determinada condición de prueba, se puede añadir esta condición de prueba al programa de pruebas unitarias. Al principio, esta prueba fallará. Pero una vez realizados los cambios, no debería fallar, ni tampoco el resto de condiciones de prueba.

Con un buen desarrollo orientado a pruebas, **un conjunto de especificaciones se refleja en un conjunto de condiciones de prueba**. Cada especificación se puede reflejar en una condición de prueba o un conjunto de ellas. Esto propicia especificaciones *testables* o ejecutables en lugar de especificaciones abstractas que no está claro cómo se pueden verificar en la práctica. Si una versión del software pasa las pruebas, entonces es conforme a las especificaciones. El conjunto de pruebas se va ampliando (y posiblemente reorganizando y rediseñando) conforme se desarrollan nuevas versiones del software. La automatización de las pruebas hace posible verificar que cada nueva versión del software sigue pasando todas las pruebas previamente existentes, además de las nuevas que se añadan para la nueva funcionalidad.

Este planteamiento es especialmente útil para las **pruebas de regresión**. Después de cada cambio en el software, por pequeño que sea, se puede ejecutar un conjunto de pruebas de la funcionalidad fundamental del programa. De esta forma, se verifica que los cambios realizados no introducen, además de nueva funcionalidad, nuevos fallos, es decir, que no rompen nada. Conforme se va desarrollando nueva funcionalidad, se puede ir ampliando o modificando el conjunto de pruebas de regresión. La clave para que este planteamiento sea efectivo es la automatización de las pruebas.

Nota: en realidad, deberían llamarse pruebas de no regresión, y a veces se les llama así. Una regresión es un retroceso, o una degradación, un fallo que se introduce en un proceso que debería ser de continuo avance y mejora.

# 2. Creación de clase de test para JUnit en NetBeans

Netbeans, como los demás entornos de desarrollo en general, incluye asistentes para la creación de pruebas unitarias basadas en JUnit. Estos ayudan en la creación de conjuntos de pruebas unitarias para una clase.

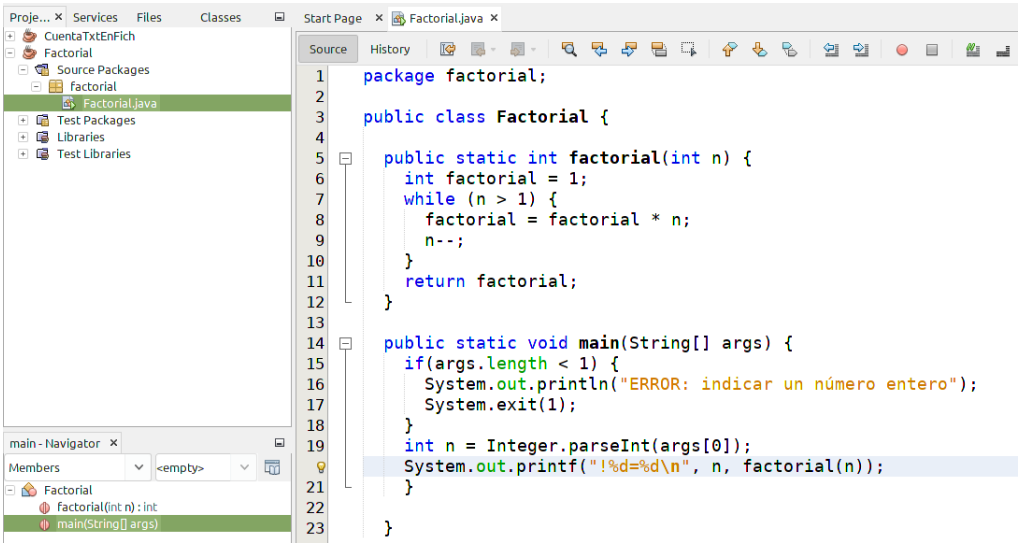
Las pruebas unitarias para una clase se crean en una clase especial, incluida en el mismo proyecto que la clase.

En los siguientes apartados se explica cómo crear un proyecto con clases de prueba para JUnit para para cada una de las clases del proyecto.

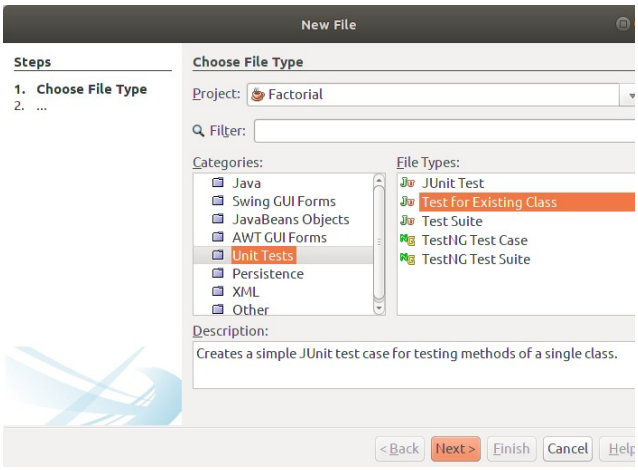
## 2.1. Creación de una clase de prueba para Junit (NetBeans)

En este apartado se explica cómo añadir una clase de pruebas basada en NetBeans para una clase existente en un proyecto cualquiera.

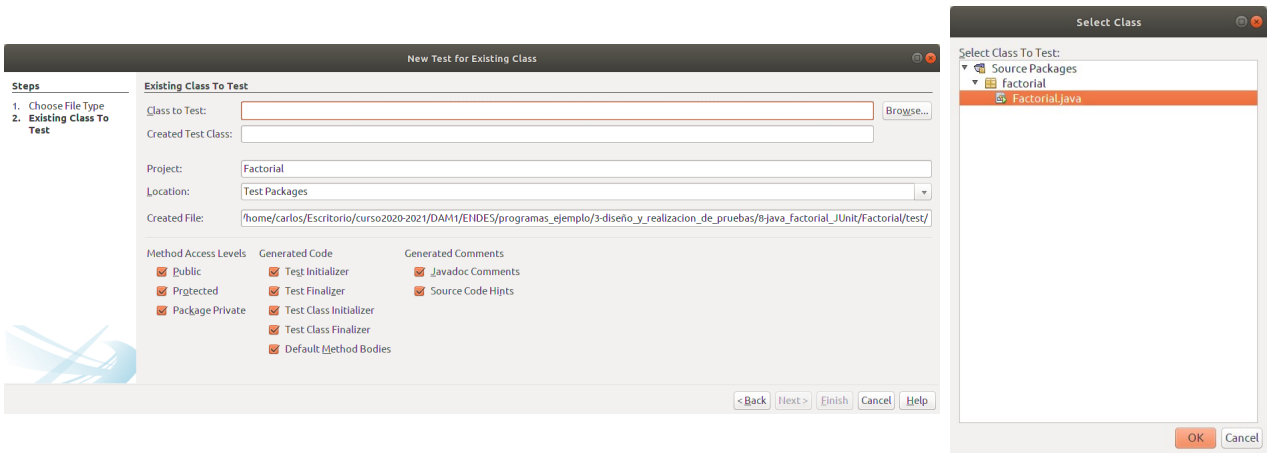
Como ejemplo, se hará esto con una clase `Factorial` que tiene solo un método que calcula el factorial de un número. Aparte de eso, en el método `main` se calcula el factorial para un número que se pasa como argumento de línea de comandos. Si no se ha pasado ninguno, se termina la ejecución con un código de error 1 (`System.exit(1)`).




Para crear una clase de pruebas, se pulsa con el botón derecho del ratón sobre el proyecto, y se elige la opción New…, Other…, y después se selecciona la categoría "Unit Tests", y el tipo de fichero "Test for Existing Class".



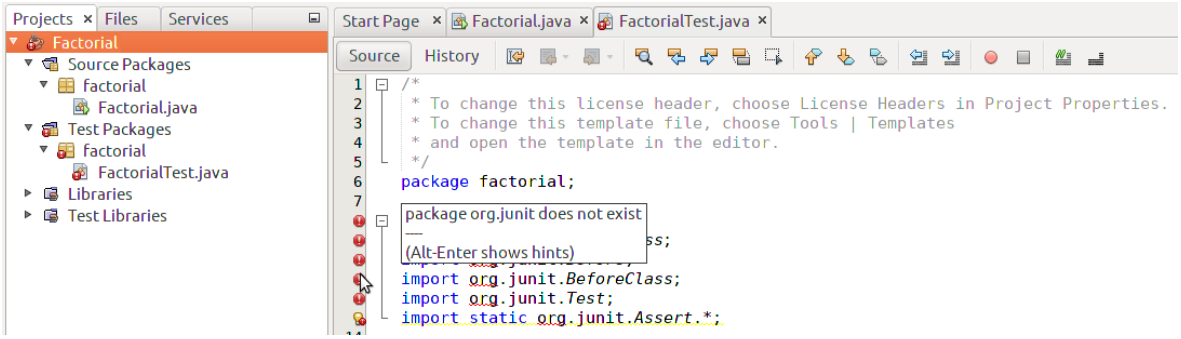

En el siguiente diálogo, en principio se pueden dejar las opciones por defecto. Hay que seleccionar una clase, y para ello hay que pulsar el botón "Browse...". En el siguiente diálogo se selecciona la clase. En este proyecto solo hay una.




Con ello se crea una clase de prueba en el proyecto, dentro de "Test Packages". Se puede ver que las clases de pruebas están en una jerarquía de paquetes y clases que es un reflejo de la jerarquía de paquetes y clases del proyecto.

## 2.2. Añadir clases de JUnit al proyecto

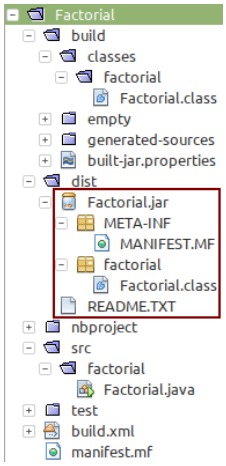
Faltan las clases de Junit, como se puede ver en el error que se muestra con las directivas `import org.junit...`




Las clases de JUnit están en un fichero de tipo `jar` que hay que descargar y añadir al proyecto.

**Recuerda: ficheros de tipo `jar`**

Un fichero de tipo jar (***j**ava **ar**chive*) contiene una colección de clases compiladas en *bytecode* e información adicional acerca de ellas. Ahora veremos un nuevo uso para ellos.

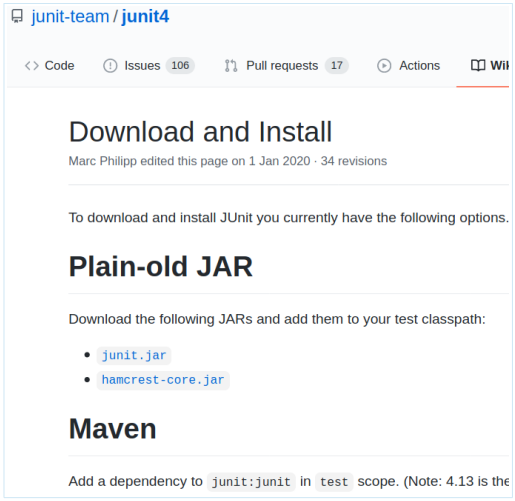



Ya se vio en el tema anterior que cuando en un entorno de desarrollo se construye un proyecto (opción `Build` o `Clean and Build`) el resultado es un fichero de tipo `jar`, que se puede ver en la vista de ficheros dentro de un directorio `dist`, y que también se puede ejecutar desde línea de comandos con la máquina virtual de Java:

```
$ java -jar Factorial.jar
Introduzca n: 6
factorial: 720
$
```

La funcionalidad de JUnit la proporciona una biblioteca de clases que viene empaquetada en ficheros `jar`, que hay que descargar y añadir al proyecto.

## Descarga de JUnit




La página web de JUnit es http://junit.org.

Aquí se utilizará la versión 4, cuya página web es https://junit.org/junit4.

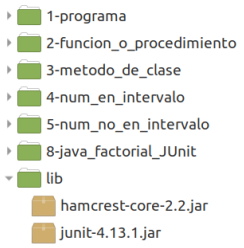
Desde ahí hay un enlace "Download and Install" que conduce a la página de descarga https://github.com/junit-team/junit4/wiki/Download-and-Install.

Se puede ver que la funcionalidad de JUnit está en dos ficheros `jar`. Hay que descargar estos ficheros.

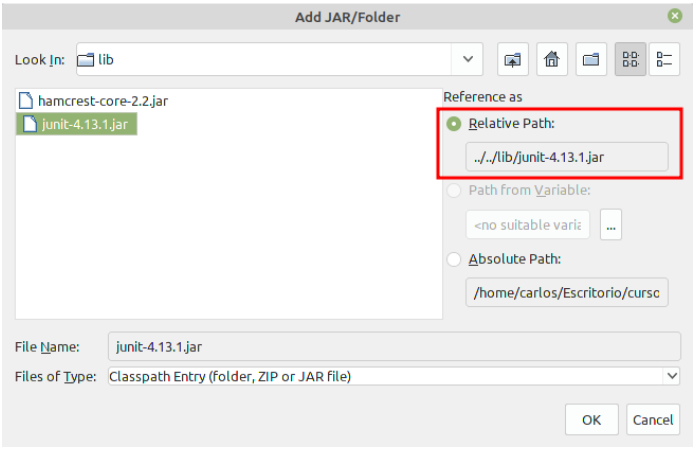
- `junit.jar`
- `hamcrest-core.jar`

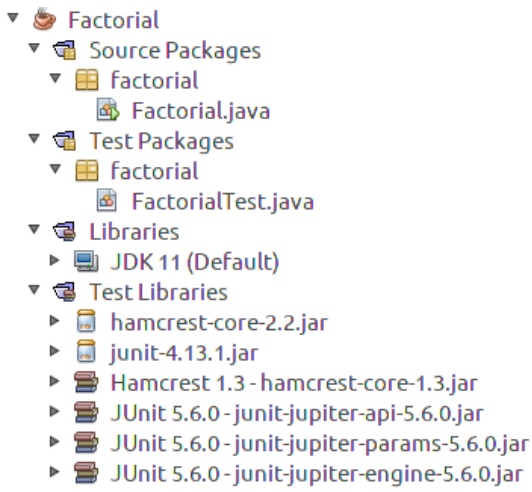
## Atención: Ubicación de los ficheros jar de JUnit

Para simplificar su uso en distintos proyectos, se creará un directorio `lib` en el mismo directorio en que se vayan a crear los proyectos para NetBeans, y en él se copiarán los ficheros jar de JUnit.

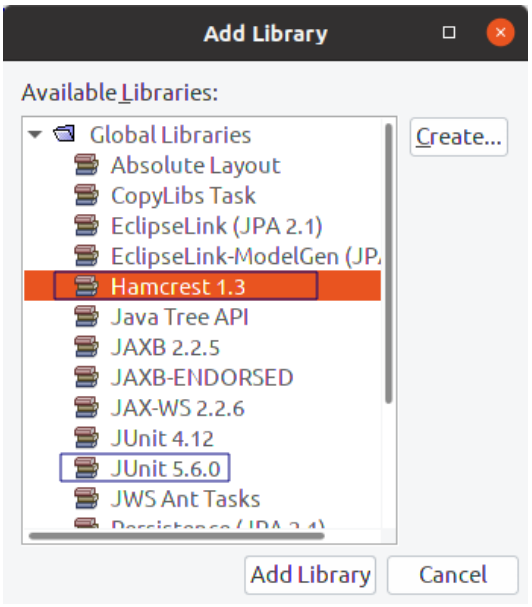



Ahora hay que añadir estos ficheros jar al proyecto. Para ello se pulsa con el botón derecho del ratón sobre el nodo "Test Libraries" en la vista de proyectos, se selecciona la opción "Add JAR/Folder...", y se seleccionan los ficheros jar a añadir, situados en el directorio `lib` creado previamente, y donde se han copiado. Para que el proyecto pueda funcionar sin problemas en otros lugares, se selecciona la opción "Relative Path". Con ello, el proyecto queda como se muestra a continuación.



**IMPORTANTE**: Según la versión utilizada de NetBeans, puede añadirse un conjunto diferente de bibliotecas estándares de clases. Es importante añadir las que se muestran en la imagen anterior de la derecha. Si falta alguna, se puede añadir pulsando con el botón derecho sobre el nodo "Test Libraries", y seleccionándolas de la lista que se muestra




A las bibliotecas que ya se habían añadido al crear la clase de prueba, se han añadido en el paso anterior los ficheros jar de JUnit.

Se puede ver que todo lo relacionado con las pruebas o *tests* está en una estructura paralela. En paralelo con los paquetes o *Packages* que contienen las clases del proyecto, están los *Test Packages* con las clases de prueba, de hecho una clase de prueba para cada clase. Y en paralelo con las bibliotecas o Libraries, están las Test Libraries con la funcionalidad adicional necesaria para las pruebas. Esto refleja la filosofía del *Test Driven Development* o desarrollo orientado a pruebas: el código de programa para la nueva funcionalidad se desarrolla en paralelo con el código para sus pruebas.

## 2.3. Creación de casos de prueba para métodos estáticos

Ahora falta añadir los casos de prueba a la clase de prueba. Para ello se pueden utilizar las plantillas de método de prueba incluidas en la clase de prueba. En esta se incluye una plantilla para cada método de la clase. Esta clase solo tiene métodos estáticos. Los métodos no estáticos se verán en un apartado posterior, pero el planteamiento básico es el mismo.

Todos los métodos de prueba están precedidos por la anotación `@Test`.

Los métodos de prueba tienen un patrón común, definido por las tres A: *Arrange*, *Act*, *Assert*.

- *Arrange* (organizar). Se asigna un valor para cada parámetro del método (en este caso, `int n`), y se asigna un valor para el resultado esperado (en este ejemplo, en la variable `expResult`).
- *Act* (actuar). Se llama al método con los valores asignados a los parámetros en el paso anterior y se obtiene el resultado en una variable `result` (en este caso, de tipo `int`).
- *Assert* (afirmar). Se compara el valor obtenido al llamar al método (`result`) con el esperado `expResult`, para verificar que son iguales, con el método `assertEquals`.

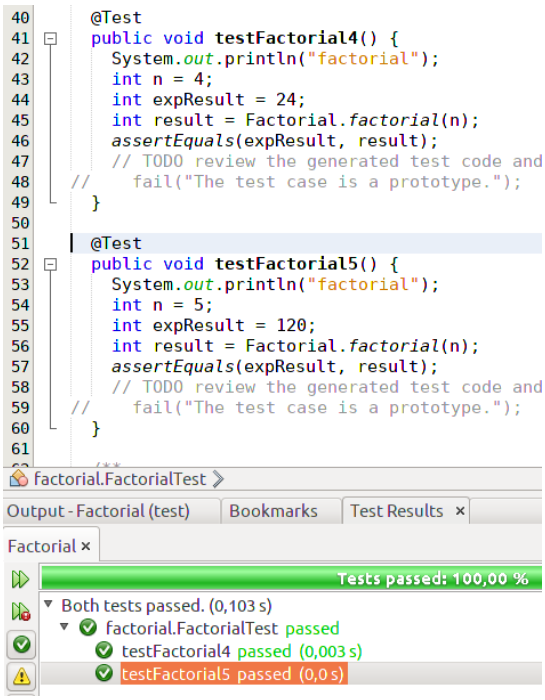
A continuación se muestran los cambios realizados en la plantilla para el método `factorial` para ejecutar el caso de prueba `n=4`. Habría que crear un nuevo método de prueba para cada caso de prueba. En cada uno, habría que asignar valores apropiados para la entrada `n` y para la salida esperada `expResult`.

```java
@Test
public void testFactorial() {
  System.out.println("factorial");
  int n = 0;
  int expResult = 0;
  int result = Factorial.factorial(n);
  assertEquals(expResult, result);

  // TODO review the generated test code
and remove the default call to fail.
  fail("The test case is a prototype.");

}
```

```java
@Test
public void testFactorial4() {
  System.out.println("factorial");
  int n = 4;
  int expResult = 24;
  int result = Factorial.factorial(n);
  assertEquals(expResult, result);

}
```

# 3. Ejecución de los casos de prueba

Para ejecutar los casos de prueba, se pulsa con el botón derecho del ratón sobre el proyecto, o sobre la clase `Factorial`, y se selecciona la opción Test. Cada caso de prueba que se ha ejecutado con éxito (es decir, para el que se cumple la condición `assertEquals(expResult, result)`, lo que significa que el resultado obtenido es igual al resultado esperado), aparece con una marca en verde.




Nota: es posible que dé errores al ejecutar las pruebas, según la versión específica del entorno de desarrollo, por utilizar distintas versiones de JUnit.
Puede ser necesario realizar algunos cambios en los `import` del principio, para que utilice las clases apropiadas de los ficheros jar de JUnit descargados y añadidos manualmente al proyecto, en lugar de las de las bibliotecas añadidas por el asistente de JUnit.

a) Cambiar:

```java
import org.junit.jupiter.api.Test;
```

por

```java
import org.junit.Test;
```

b) Cambiar

```java
import static org.junit.jupiter.Assertions.*;
```

por

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

***Actividad 1***

Revisa el conjunto de condiciones de prueba obtenido mediante el método (de caja blanca) de los caminos básicos para el programa que calcula el factorial de un número. ¿Es el anterior conjunto de condiciones de prueba lo suficientemente completo según el planteamiento del método de los caminos básicos? Si no, ¿qué le faltaría?

***Actividad 2***

Para el programa que, dado un número $x$ y un intervalo $[a, b]$, determina si el número está en el intervalo, crear una clase que lo implemente en un método. Crear una clase de prueba con un método de prueba para cada uno de los ca-

sos de prueba obtenidos mediante el método (de caja blanca) de los caminos básicos. ¿Crees conveniente añadir algún caso de prueba más? ¿Cuál o cuáles? Hazlo, y vuelve a ejecutar el plan de pruebas.

***Actividad 3***

Para el programa que calcula la suma de todos los números positivos de una secuencia de números enteros, dada en un array, crear una clase que lo implemente en un método. Crea una clase de prueba con un método de prueba para cada uno de los casos de prueba obtenidos mediante el método (de caja blanca) de los caminos básicos. Añade varios casos de prueba más, para probar con más números y con distintas combinaciones de números positivos, negativos, y cero.

# 4. Creación de casos de prueba para métodos no estáticos

Se pone a continuación un ejemplo de una clase `Personaje` que está en una posición dentro de un tablero de 10 filas x 10 columnas. La clase tiene un método para moverse en una dirección, y también para realizar una secuencia de movimientos dados en un *array*. Se define un tipo enumerado `Direccion` que tiene un valor para cada una de las cuatro direcciones arriba, abajo, izquierda y derecha. En el método `main()` se incluye un pequeño programa de prueba.

```
package personaje;

enum Direccion {
    IZQUIERDA, DERECHA, ARRIBA, ABAJO
}

public class Personaje {

    private static final int MAX_X = 10;
    private static final int MAX_Y = 10;

    // miembros de datos
    private final String nomPers;
    private int x;
    private int y;

    // constructor
    public Personaje(String nombre, int x, int y) {
        this.nomPers = nombre;
        this.x = x;
        this.y = y;
    }

    // métodos
    public String getNomPers() {
        return nomPers;
    }

    public int getX() {
        return x;
    }
```

```
    public int getY() {
        return y;
    }

    public void avanza(Direccion mov) {
        switch (mov) {
            case IZQUIERDA:
                if (x > 0) {
                    x--;
                }
                break;
            case DERECHA:
                if (x < MAX_X) {
                    x++;
                }
                break;
            case ARRIBA:
                if (y > 0) {
                    y--;
                }
                break;
            case ABAJO:
                if (y < MAX_Y) {
                    y++;
                }
                break;
        }

    }

    public static void main(String[] args) {

        Personaje p = new Personaje("Bicho", 2, 3);

        System.out.printf("Posicion inicial: (%d, %d)\n", p.getX(), p.getY());

        p.avanza(Direccion.ARRIBA);
        p.avanza(Direccion.IZQUIERDA);
        p.avanza(Direccion.IZQUIERDA);
        p.avanza(Direccion.ARRIBA);
        p.avanza(Direccion.IZQUIERDA);
        p.avanza(Direccion.DERECHA);
        p.avanza(Direccion.ABAJO);

        System.out.printf("Posicion final: (%d, %d)\n", p.getX(), p.getY());
    }
}
```

La generación de la clase de prueba se hace igual que como ya se ha explicado antes. En este caso, se generan dos clases de prueba, para la clase `Personaje` y también para la clase `Direccion`. No nos interesa esta última, que se puede borrar sin más.

En cuanto a la clase `Direccion`, se crea una plantilla de método de prueba para cada uno de los métodos de la clase. Solo interesa probar el método `avanza`, del que se muestra a continuación la plantilla de su método de prueba.

```
@Test
public void testAvanza() {
    System.out.println("avanza");
    Direccion mov = null;
    Personaje instance = null;
    instance.avanza(mov);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
```

Lo que se quiere probar es cada uno de los movimientos. Y se quieren probar los casos límites en que el personaje se encuentra en un límite del tablero y se intenta realizar un movimiento que no puede realizarse porque, de realizarse, provocaría que el personaje salga del tablero. Se pone como ejemplo dos métodos de prueba para los movimientos a la izquierda, uno en que el movimiento se puede realizar y otro en que no se puede realizar, porque el personaje está en el borde izquierdo.

Para empezar, se crea un objeto de tipo `Persona` en una posición dada. Después se ejecuta el método `avanza` con un movimiento determinado, y por último se utiliza el método `assert` para verificar que la posición del objeto es correcta después de ejecutarse el método. Las coordenadas de la posición se obtienen con los métodos `getX()` y `getY()`.

```
@Test
public void testAvanzaIzq() {
    System.out.println("avanza izq.");
    Direccion mov = Direccion.IZQUIERDA;
    int x = 3;
    int y = 2;
    Personaje instance = new Personaje("p", x, y);
    instance.avanza(mov);
    assert (instance.getX() == x - 1 && instance.getY() == y);
}

@Test
public void testAvanzaIzqLim() {
    System.out.println("avanza izq.");
    Direccion mov = Direccion.IZQUIERDA;
    int x = 0;
    int y = 2;
    Personaje instance = new Personaje("p", x, y);
    instance.avanza(mov);
    assert (instance.getX() == x && instance.getY() == y);
}
```

***Actividad 4***

Completa la clase de prueba para la clase `Personaje` con métodos análogos para probar los avances en el resto de direcciones. Deben ser 8 métodos de prueba: 2 por cada una de las cuatro direcciones. Para cada dirección, debe ha-

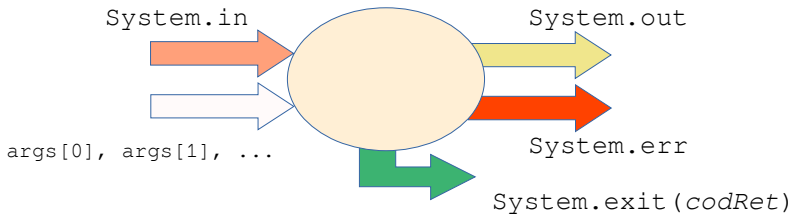
ber un método de prueba para cuando el personaje está en un borde en el que no se puede realizar el movimiento, y otro cuando no es el caso y se puede realizar el movimiento.

# 5. Interacción de un programa con su entorno

Hasta ahora se ha utilizado JUnit para pruebas de métodos estáticos de clases. Este uso es muy apropiado para pruebas de algoritmos sencillos con planes de prueba obtenidos mediante métodos de caja blanca para un algoritmo. El algoritmo se implementa en un método estático de clase, y cada condición de prueba se traduce en un método de prueba de una clase de prueba, en la que se asignan valores a las variables de entrada, y se comprueba el valor devuelto.

Este planteamiento es demasiado limitado para realizar pruebas de caja negra, en las que no necesariamente se prueban métodos de clase, sino que se prueban programas, incluso sin un conocimiento previo de su código fuente. Para realizar pruebas de este tipo, hay que tener en cuenta la manera en que los programas interactúan con el mundo exterior.

En general, se puede representar la interacción de un programa con el mundo exterior, de manera general, con el siguiente esquema.




| | |
|---|---|
| **Entrada** | **Argumentos de línea de comandos**. En Java se obtienen en el parámetro del método `main`, `String[] args`, donde `args[0]`, `args[1]`, `args[2]`, etc, son los sucesivos parámetros de línea de comandos. |
| | **Entrada estándar**. En Java la entrada estándar es `System.in`. Una forma en que se pueden obtener datos de diversos tipos desde la entrada estándar es con un objeto de clase `Scanner` creado para `System.in`, por ejemplo con `Scanner s = new Scanner(System.in)` |
| **Salida** | **Salida estándar**. En Java, la salida estándar es `System.out` en Java. Se puede escribir en ella con los métodos `print`, `println`, o `printf`. |
| | **Salida de error**. En Java, la salida de error es `System.err` en Java. Se utiliza para escribir información relativa a errores.<br>Para escribir en la salida de error se utilizan los mismos métodos que para escribir a la salida estándar |
| **Código de retorno** | Un programa devuelve siempre un código de retorno una vez que ha terminado su ejecución. Por defecto es 0, lo que indica que el programa se ejecutó con éxito. Pero puede devolver un valor distinto, lo que significa que finalizó con un error, y el código de error indica el error específico. Se puede finalizar la ejecución de un programa devolviendo un código de retorno *codRetorno* con `System.exit(`*codRetorno*`)`. |

Supongamos que se tiene el siguiente programa, al que se le pasa por línea de comandos una medida en pies y pulgadas, y que convierte la medida en cm. El cálculo en sí es muy sencillo: un pie equivale a doce pulgadas, y una pulgada equivale a 2,54 cm. Por lo tanto, la medida en cm es 2,54*(12*pies+pulgadas).

El programa obtiene los valores para las dos variables de entrada `pies` y `pulgadas` de los parámetros de línea de comandos, y realiza una serie de validaciones sobre ambos. Si son correctos, muestra la medida equivalente en cm en la salida estándar (`System.out`). Si no, muestra un mensaje de error en la salida de error (`System.err`). Las validaciones que realiza el programa son las siguientes:

- `pies` es un número entero no negativo.
- `pulgadas` es un número positivo, que puede tener decimales, y menor que 12. Con 12 pulgadas se tendría un pie. El programa exige, por tanto, una representación normalizada de la medida, de manera que la parte de la medida que se expresa en pulgadas sea menor que una pie.

```
package piesypulgadas;

public class PiesYPulgadas {

  public static double convierteACm(int pies, double pulgadas) {
    return 2.54 * (pulgadas + 12 * pies);
  }

  public static void main(String[] args) {
    int pies = 0;
    try {
      pies = Integer.parseInt(args[0]);
    } catch (NumberFormatException ex) {
      System.err.println("ERROR: pies no entero.");
      return;
//      System.exit(1);
    }
    if(pies < 0) {
      System.err.println("ERROR: pies negativo.");
      return;
//      System.exit(2);
    }
    double pulgadas = 0;
    try {
      pulgadas = Double.parseDouble(args[1]);
    } catch (NumberFormatException ex) {
      System.err.println("ERROR: pulgadas no numérico.");
      return;
//      System.exit(3);
    }
    if(pulgadas < 0) {
      System.err.println("ERROR: pulgadas negativo.");
      return;
//      System.exit(4);
    } else if(pulgadas >= 12) {
      System.err.println("ERROR: pulgadas mayor o igual que 12.");
      return;
//      System.exit(5);
    }
    System.out.printf("%f\n", convierteACm(pies, pulgadas));
  }
}
```

# 6. Extensión de Junit para clase `System` (`system-rules`)

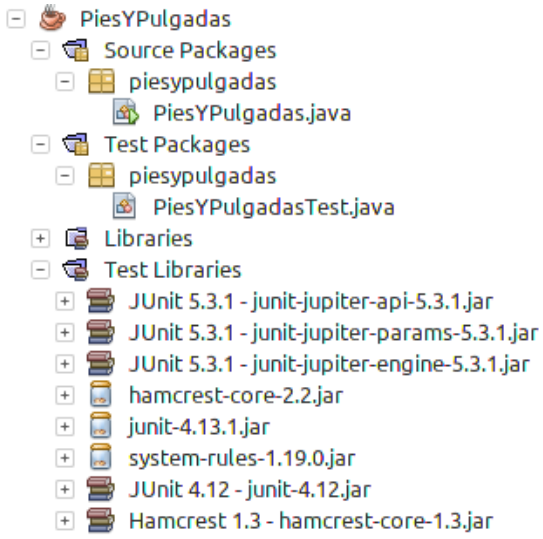
Se puede ver, por tanto, que la interacción de un programa en Java con el mundo exterior se puede gestionar a través de los parámetros del método `main` (`args[0]`, `args[1]`, `args[2]`, etc) y a través de la clase `System`.

El control de estos mecanismos permite realizar una verdadera prueba de caja negra, sin suponer absolutamente nada acerca de la implementación del programa.

Pues bien, existe una biblioteca de clases para JUnit que permite interactuar con todos estos mecanismos, lo que permite automatizar las pruebas de caja negra de programas desarrollados en Java, es decir, del programa en conjunto, con respecto a sus especificaciones, y sin basarse en ningún conocimiento previo de su implementación, es decir, de las clases que utiliza ni de cómo las utiliza.

Esta biblioteca de clases es `system-rules`, disponible en un fichero jar de nombre `system-rules-`*`num_versión`*`.jar`. Este debe añadirse al proyecto junto con los ficheros jar de Junit.

En la siguiente captura de pantalla se muestra la estructura de un proyecto con una clase de prueba `PiesYPulgadas`, con su correspondiente clase de prueba para Junit `PiesYPulgadasTest`, y que utiliza `system-rules`.




La página web oficial de system-rules es https://stefanbirkner.github.io/system-rules. En ella se puede obtener el fichero jar y hay documentación y ejemplos de uso.

Para tener acceso a la salida estándar (`System.out`), a la salida de error (`System.err`), y al valor del código de retorno del progama (que es por defecto 0, pero en Java se puede terminar la ejecución del programa devolviendo uno distinto con `System.exit(`*`codRetorno`*`)`), deben añadirse las siguientes definiciones de variables de instancia a la clase de prueba.

| | |
|---|---|
| Salida estándar<br>`System.out` | `@Rule`<br>`public final SystemOutRule systemOutRule = new SystemOutRule().enableLog();` |
| Salida de error<br>`System.err` | `@Rule`<br>`public final SystemErrRule systemErrRule = new SystemErrRule().enableLog();` |
| Código de retorno<br>`System.exit()` | `@Rule`<br>`public final ExpectedSystemExit exitRule = ExpectedSystemExit.none();` |

Las pruebas del programa se hacen llamando al método `main()`. Cada condición de prueba se traduce en un método de prueba en el que se llama al método `main()`. En esta llamada se le pueden pasar valores para las variables de entrada. Es importante tener claro que para este proyecto de ejemplo, al contrario que para proyectos anteriores de ejemplo, se está probando el método `main`, y se está centrando la atención en él. En proyectos anteriores interesaba probar un método de una clase, y el método `main` podía incluso no hacer nada o, en cualquier caso, lo que hiciera era irrelevante.

Se puede verificar la salida realizada a `System.out` y la salida de error realizada a `System.err` como se muestra en la siguiente clase de ejemplo, junto con su correspondiente clase de prueba. Cuando los valores introducidos son incorrectos, se detecta en validaciones previas, y se envía un mensaje de error a la salida de error `System.err`. Si son correctos, se envía un texto con el resultado de la conversión a la salida estándar `System.out`. Se incluyen comentadas llamadas a `System.exit` que devuelven un código de error distinto para cada condición de error. Este es un planteamiento alternativo al que se dedicará la atención más adelante.

La clase de prueba para la anterior clase es la siguiente. Incluye un método de prueba para cada posible condición de error, y también uno para una condición no de error. Para la condición no de error, es importante formatear el valor exactamente igual que el generado por el programa, para lo que se utiliza el método `String.format` con la misma cadena de formato (`"%f\n"`) utilizada por el programa para escribir el valor en cm resultante.

Por otra parte, es necesario importar al principio las clases de `system-rules` que se van a utilizar.

```
package piesypulgadas;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.Rule;
//import org.junit.contrib.java.lang.system.ExpectedSystemExit;
import org.junit.contrib.java.lang.system.SystemErrRule;
import org.junit.contrib.java.lang.system.SystemOutRule;

public class PiesYPulgadasTest {

  public PiesYPulgadasTest() {
  }

  @BeforeAll
  public static void setUpClass() {
  }

  @AfterAll
  public static void tearDownClass() {
  }

  @BeforeEach
  public void setUp() {
  }

  @AfterEach
  public void tearDown() {
  }

  @Rule
  public final SystemErrRule systemErrRule = new SystemErrRule().enableLog();

  @Rule
  public final SystemOutRule systemOutRule = new SystemOutRule().enableLog();

//  @Rule
//  public final ExpectedSystemExit exitRule = ExpectedSystemExit.none();

  /**
   * Test of main method, of class PiesYPulgadas.
   */
  @Test
```

```
  public void testMainPies_1_PiesNoEntero() {
    String[] args = {"a", "7"};
//    exitRule.expectSystemExitWithStatus(1);
    PiesYPulgadas.main(args);
    assertEquals("ERROR: pies no entero.\n", systemErrRule.getLog());
  }

  @Test
  public void testMainPies_1_PiesNegativo() {
    String[] args = {"-5", "2"};
//    exitRule.expectSystemExitWithStatus(2);
    PiesYPulgadas.main(args);
    assertEquals("ERROR: pies negativo.\n", systemErrRule.getLog());
  }

  @Test
  public void testMainPies_1_PulgadasNoNumerico() {
    String[] args = {"3", "c"};
//    exitRule.expectSystemExitWithStatus(3);
    PiesYPulgadas.main(args);
    assertEquals("ERROR: pulgadas no numérico.\n", systemErrRule.getLog());
  }

  @Test
  public void testMainPies_1_PulgadasNegativo() {
    String[] args = {"2", "-5.3"};
//    exitRule.expectSystemExitWithStatus(4);
    PiesYPulgadas.main(args);
    assertEquals("ERROR: pulgadas negativo.\n", systemErrRule.getLog());
  }

  @Test
  public void testMainPies_1_PulgadasNoMenorQue12() {
    String[] args = {"2", "15.3"};
//    exitRule.expectSystemExitWithStatus(5);
    PiesYPulgadas.main(args);
    assertEquals("ERROR: pulgadas mayor o igual que 12.\n",
      systemErrRule.getLog());
  }

  @Test
  public void testMainClasesCorrectas() {
    String[] args = {"3", "2.3"};
    PiesYPulgadas.main(args);
    assertEquals(String.format("%f\n",(3*12+2.3)*2.54),
      systemOutRule.getLog());
  }
}
```

En el ejemplo anterior, los mensajes de error se escriben en la salida de error `System.err`. Pero algunos programas pueden escribirlos en la salida estándar `System.out`. Los métodos de prueba deberán utilizar los mecanismos apropiados para el programa que se pruebe.

Si además de dar un mensaje de error, se termina la ejecución con `System.exit(`*codRet*`)` en lugar de `return`, la clase anterior quedaría como sigue.

```java
package piesypulgadas;

public class PiesYPulgadas {

  public static double convierteACm(int pies, double pulgadas) {
    return 2.54 * (pulgadas + 12 * pies);
  }

  public static void main(String[] args) {
    int pies = 0;
    try {
      pies = Integer.parseInt(args[0]);
    } catch (NumberFormatException ex) {
      System.err.println("ERROR: pies no entero.");
//      return;
      System.exit(1);
    }
    if(pies < 0) {
      System.err.println("ERROR: pies negativo.");
//      return;
      System.exit(2);
    }
    double pulgadas = 0;
    try {
      pulgadas = Double.parseDouble(args[1]);
    } catch (NumberFormatException ex) {
      System.err.println("ERROR: pulgadas no numérico.");
//      return;
      System.exit(3);
    }
    if(pulgadas < 0) {
      System.err.println("ERROR: pulgadas negativo.");
//      return;
      System.exit(4);
    } else if(pulgadas >= 12) {
      System.err.println("ERROR: pulgadas mayor o igual que 12.");
//      return;
      System.exit(5);
    }
    System.out.printf("%f\n", convierteACm(pies, pulgadas));
  }
}
```

Y entonces, se verifica el código de retorno con `exitRule.expectSystemExitWithStatus(codRet)` antes de la llamada al método `main` que provoca la terminación con el código de reotrno dado, distinto de cero. La verificación de la salida hacia la salida estándar `System.out` o hacia la salida de error `System.err` es imposible, o al menos no es posible de manera sencilla y directa.

```java
package piesypulgadas;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
```

```java
import org.junit.jupiter.api.BeforeAll;
import org.junit.Test;
import org.junit.Rule;
import org.junit.contrib.java.lang.system.ExpectedSystemExit;
//import org.junit.contrib.java.lang.system.SystemErrRule;
import org.junit.contrib.java.lang.system.SystemOutRule;

public class PiesYPulgadasTest {

  public PiesYPulgadasTest() {
  }

  @BeforeAll
  public static void setUpClass() {
  }

  @AfterAll
  public static void tearDownClass() {
  }

  @BeforeEach
  public void setUp() {
  }

  @AfterEach
  public void tearDown() {
  }

  @Rule
  public final ExpectedSystemExit exitRule = ExpectedSystemExit.none();

  /**
   * Test of main method, of class PiesYPulgadas.
   */
  @Test
  public void testMainPies_1_PiesNoEntero() {
    String[] args = {"a", "7"};
    exitRule.expectSystemExitWithStatus(1);
    PiesYPulgadas.main(args);
  }

  @Test
  public void testMainPies_1_PiesNegativo() {
    String[] args = {"-5", "2"};
    exitRule.expectSystemExitWithStatus(2);
    PiesYPulgadas.main(args);
  }

  @Test
  public void testMainPies_1_PulgadasNoNumerico() {
    String[] args = {"3", "c"};
    exitRule.expectSystemExitWithStatus(3);
    PiesYPulgadas.main(args);
```

```
  }

  @Test
  public void testMainPies_1_PulgadasNegativo() {
    String[] args = {"2", "-5.3"};
    exitRule.expectSystemExitWithStatus(4);
    PiesYPulgadas.main(args);
  }

  @Test
  public void testMainPies_1_PulgadasNoMenorQue12() {
    String[] args = {"2", "15.3"};
    exitRule.expectSystemExitWithStatus(5);
    PiesYPulgadas.main(args);
  }

  @Test
  public void testMainClasesCorrectas() {
    String[] args = {"3", "2.3"};
    PiesYPulgadas.main(args);
    assertEquals(String.format("%f\n",(3*12+2.3)*2.54),
      systemOutRule.getLog());
  }
}
```

Se puede argumentar que es más correcto que un programa devuelva un código de error, además de escribir los mensajes de error. Es decir, que finalice su ejecución con `System.exit(codRet)` y no únicamente con `return`. Esto permite obtener el código de retorno cuando el programa se invoca desde otro (por ejemplo, desde un *shell script*), y actuar de una forma diferente según el código de error.

En particular, y como se ha visto, es mucho más sencillo verificar un código de error que un mensaje de error con JUnit. Además, será mucho más difícil que en el futuro se cambien los códigos de error que los mensajes de error. los mensajes de error podrían querer cambiarse en un futuro. Por ejemplo, se podría querer traducirlos a otros idiomas. Estas consideraciones ilustran de nuevo las ventajas del TDD o *Test Driven Development*, y es el desarrollo de programas que se presten de manera natural a verificación o pruebas automáticas, y el desarrollo conjunto de las clases y de sus correspondientes clases de prueba.

***Actividad 5***

En el tema dedicado a desarrollo de conjuntos de condiciones de prueba basados en métodos de caja negra se ha puesto un ejemplo para un programa que tiene dos entradas de tipo entero horas y minutos, y como salida una cadena de caracteres *hh:mm*, donde *hh* representa la hora con dos dígitos, y *mm* representa los minutos con dos dígitos. Se han desarrollado condiciones de prueba con el método de las clases de equivalencia y se han obtenido más condiciones de prueba para los valores límites.

Crea un programa en Java con esta funcionalidad, que obtenga las horas y los minutos de los argumentos de línea de comandos. El programa tiene que hacer todas las validaciones y dar los mensajes de error apropiados para las condiciones de error, según el conjunto de condiciones de prueba desarrollado. Los mensajes de error deben escribirse en la salida de error. El programa debe finalizar, en caso de error, sin devolver un código de retorno distinto de cero, es decir, no debe terminar con `System.exit(codRet)`. Si las entradas son correctas, el programa debe escribir el resultado en la salida estándar.

***Actividad 6***

Crear una copia del programa anterior y hacer los cambios necesarios para que, en caso de error, además de escribir

un mensaje de error a la salida de error, el programa finalice con un código de retorno distinto para cada posible condición de error.

Cambiar los métodos de prueba para condiciones de error para que verifiquen el código de retorno.