Diseño e implementación de pruebas

Álvaro González Sotillo

6 de mayo de 2026

Índice

1. Pruebas del *software* **1**

2. Pruebas de caja blanca **3**

3. Herramientas para la cobertura de código **6**

4. Pruebas de caja negra **7**

5. TDD **8**

6. JUnit **9**

7. Referencias **13**

1. Pruebas del *software*

- El desarrollo solo termina tras comprobar que el código funciona
  - Un método
  - Varias clases que colaboran
  - Una aplicación
  - Un conjunto de aplicaciones y servidores
- Se comprueba que funciona mediante **pruebas**

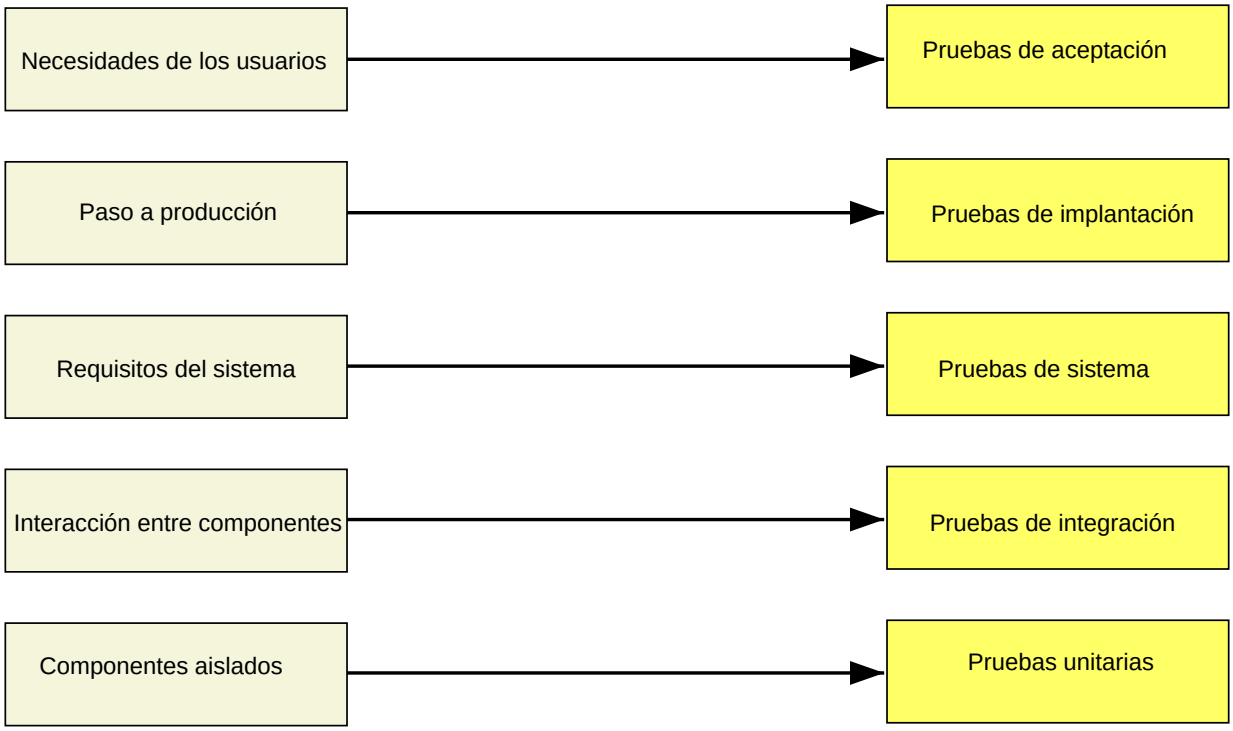
1.1. Ejemplos

Objeto de prueba	Caso de prueba
`Math.abs()` se corresponde con el valor absoluto	Se espera que `Math.abs(-10)` sea `10`
Si Mario cae sobre un *goompa*, Mario rebota	Se espera que Mario rebote al saltar sobre el pimer *goompa* en el mundo
No se puede comprar a crédito	Se espera que el usuario `US9474`, con saldo `35`, no puede comprar el art

1.2. Qué es un caso de prueba

- Objeto de prueba:
  - qué requisito se está comprobando
- Condiciones de la prueba:
  - estado del sistema previo a la prueba: variables, base de datos, conexiones, ventanas abiertas, ficheros...
  - acciones sobre el sistema: invocar métodos, pulsar botones, lanzar comandos...
- Resultado esperado
  - estado final del sistema: variables, base de datos, ventanas mostradas...

1.3. Nivel de las pruebas



1.3.1. Pruebas unitarias

- Prueban unidades de código
- En java, un método o una clase

1.3.2. Pruebas de integración

- Comprueban varias clases trabajando de forma conjunta

1.3.3. Pruebas de sistema

- Se verifica el sistema completo (aplicación o conjunto de aplicaciones)
- Se suelen realizar en sistemas no finales
  - Entorno de preproducción / entorno de pruebas

1.3.4. Pruebas de implantación

- Se verifica el sistema completo (como las pruebas de sistema)
- Pero el entorno es el definitivo
  - Entorno de producción

1.3.5. Pruebas de aceptación

- Se verifica el sistema completo (como las pruebas de implantación)
- Se realizan por el usuario final
- Suelen tener valor contractual: si las pruebas se aceptan, el proyecto está legalmente entregado y es funcional

1.4. Cuánto probar

- ¿Cuántas pruebas se necesitan para asegurar que `Math.abs()` es correcto?
- Que funcione para `4`, `24`, `-2341`, `3113` no garantiza al 100 % que funcione para `-54`.
- Pero tampoco es práctico probar para todos los números `int`, `log`, `float` y `double`
- Es necesario definir un conjunto de pruebas
  - Factible: no demasiadas pruebas
  - Completo: lo probado garantiza que los casos no probados también son correctos

1.5. Cómo decidir qué probar

- Caja blanca: decido qué probar mirando el código
  - Cobertura de decisiones
- Caja negra: decido qué probar en base a la funcionalidad
  - Particiones equivalentes
  - Análisis de valores límite
  - Conjetura de errores

2. Pruebas de caja blanca

- Se basan en la cobertura del código
  - Cobertura de decisiones: cada salto (`if`, `while`, `for`) se ha ejecutado como cierto y falso
  - Cobertura de condiciones: cada parte de una condición compuesta (`and`, `or`) se ha ejecutado como cierta y falso
- Complejidad ciclomática
  - Medida de cuántos caminos de ejecución son posibles
  - Son un límite mínimo para conseguir la cobertura

2.1. Complejidad ciclomática

- Lo aplicaremos solo a funciones (métodos)
- Un método se representa como un grafo
  - Solo un nodo de entrada
  - Un nodo final por cada `return`
  - Las operaciones son los nodos
  - Un nodo puede tener varias salidas si es de decisión (`if`, `switch`, `for`)
- La complejidad final es el número de áreas:

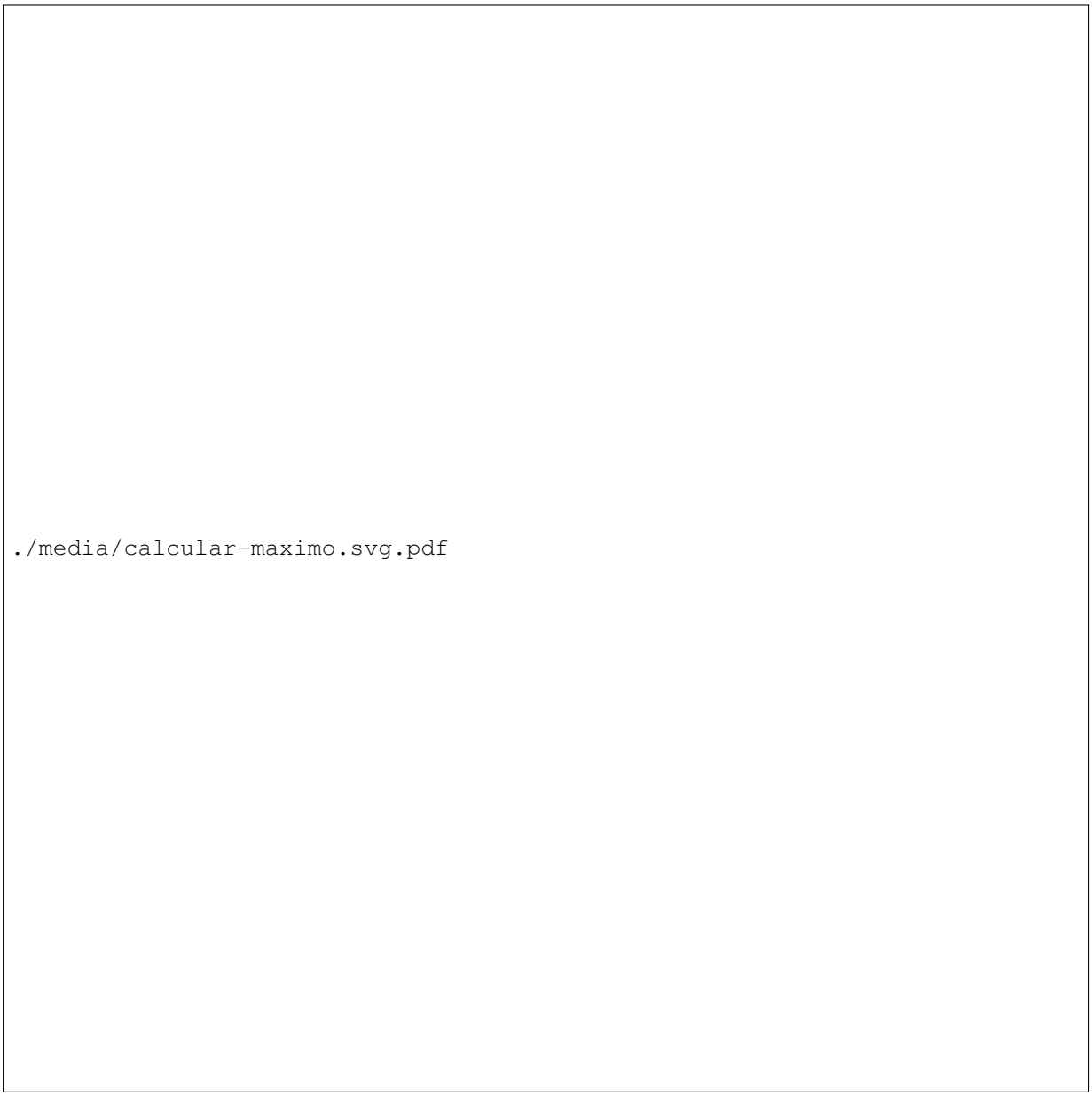
```
Complejidad = Aristas - Nodos + 2*NodosFinales
```

- Más complejo implica más difícil de probar (y de entender)

2.2. Ejemplo de complejidad ciclomática

- Miraremos la cobertura de decisiones

```java
public int calcularMaximo(int a, int b, int c) {
    if (a > b && a > c) { // Camino 1
        return a;
    } else if (b > c) { // Camino 2
        return b;
    } else { // Camino 3
        return c;
    }
}
```



2.3. Ejemplo de complejidad cilomática (II)

- Ejemplo de cobertura de condiciones
- Nota: Sonarqube no calcula la complejidad ciclomática, sino la complejidad cognitiva

```
private static void insertionSort(byte[] a, int low, int high) {
    1 for (int i, k = low; ++k < high; ) {
        byte ai = a[i = k];

        2 if (ai < a[i - 1]) {
            3 while (--i >= low  4 && ai < a[i]) {
                a[i + 1] = a[i];
            }
            a[i + 1] = ai;
        }
    }
}
```

./media/sort.svg.pdf

2.4. Reducir la complejidad ciclomática

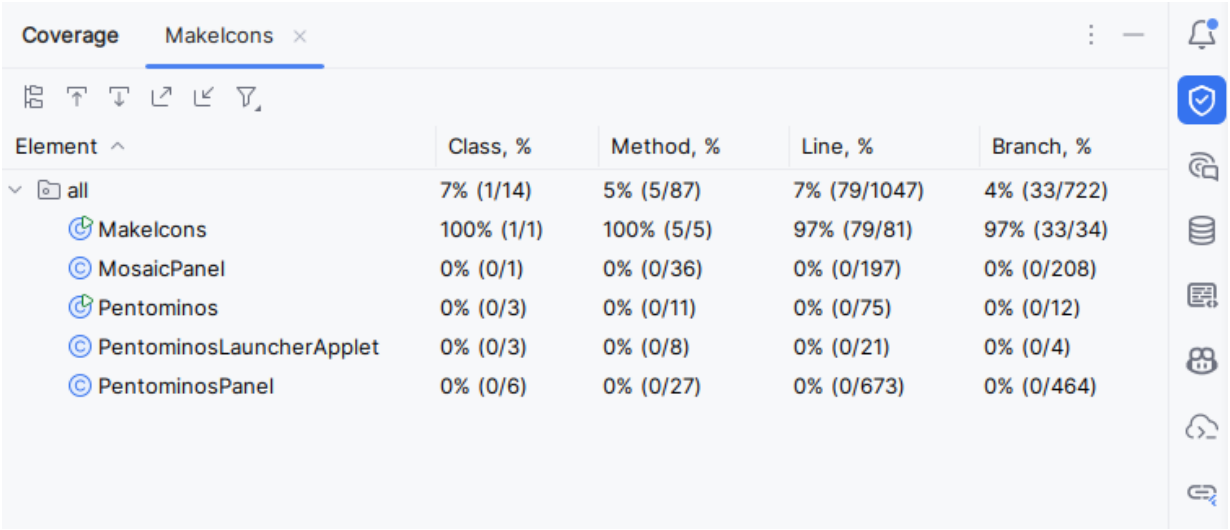
- Dividir funciones grandes: Extraer métodos pequeños para reducir la complejidad.
- Evitar estructuras de control anidadas: *fail fast* (`return` en cuanto se pueda)

3. Herramientas para la cobertura de código

- Idea: el código no puede estar probado si no se pasa por todas las líneas de código
- Pruebas de cobertura: durante las pruebas, todas las líneas de código se deben ejecutar al menos una vez

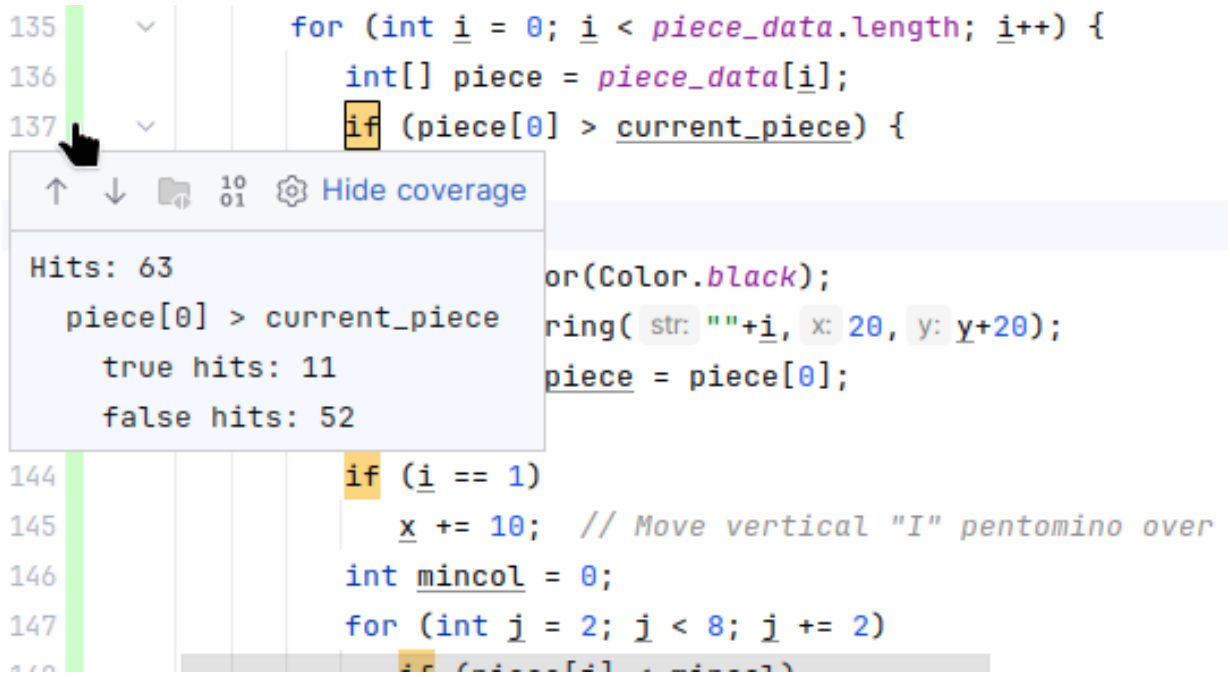
3.1. Cobertura en Intellij

- `SHIFT SHIFT run with coverage`



3.2. Condiciones y decisiones

- Cobertura de decisiones: cada salto (`if`, `while`, `for`) se ha ejecutado como cierto y falso
- Cobertura de condiciones: cada parte de una condición compuesta (`and`, `or`) se ha ejecutado como cierta y falso
  - Esta es más complicada de conseguir



3.3. Cobertura en proyectos grandes

- Por ejemplo, Jacoco
- Se puede integrar en SonarQube

4. Pruebas de caja negra

- Clases de equivalencia
- Análisis de valores límite
- Conjetura de errores

4.1. Clases de equivalencia

- Los valores de entrada suelen se de varias *clases*
- Hacemos una prueba por cada *clase*
- Ejemplo:
  - `Math.abs()` puede recibir positivos, negativos, cero, Inf y NaN.
- Ejercicios:
  - `int[] ordenaTresEnterosDeMayorAMenor(int,int,int)` (sin usar sort)
  - Función que valida un número de teléfono que puede ser local a España o internacional

4.2. AVL

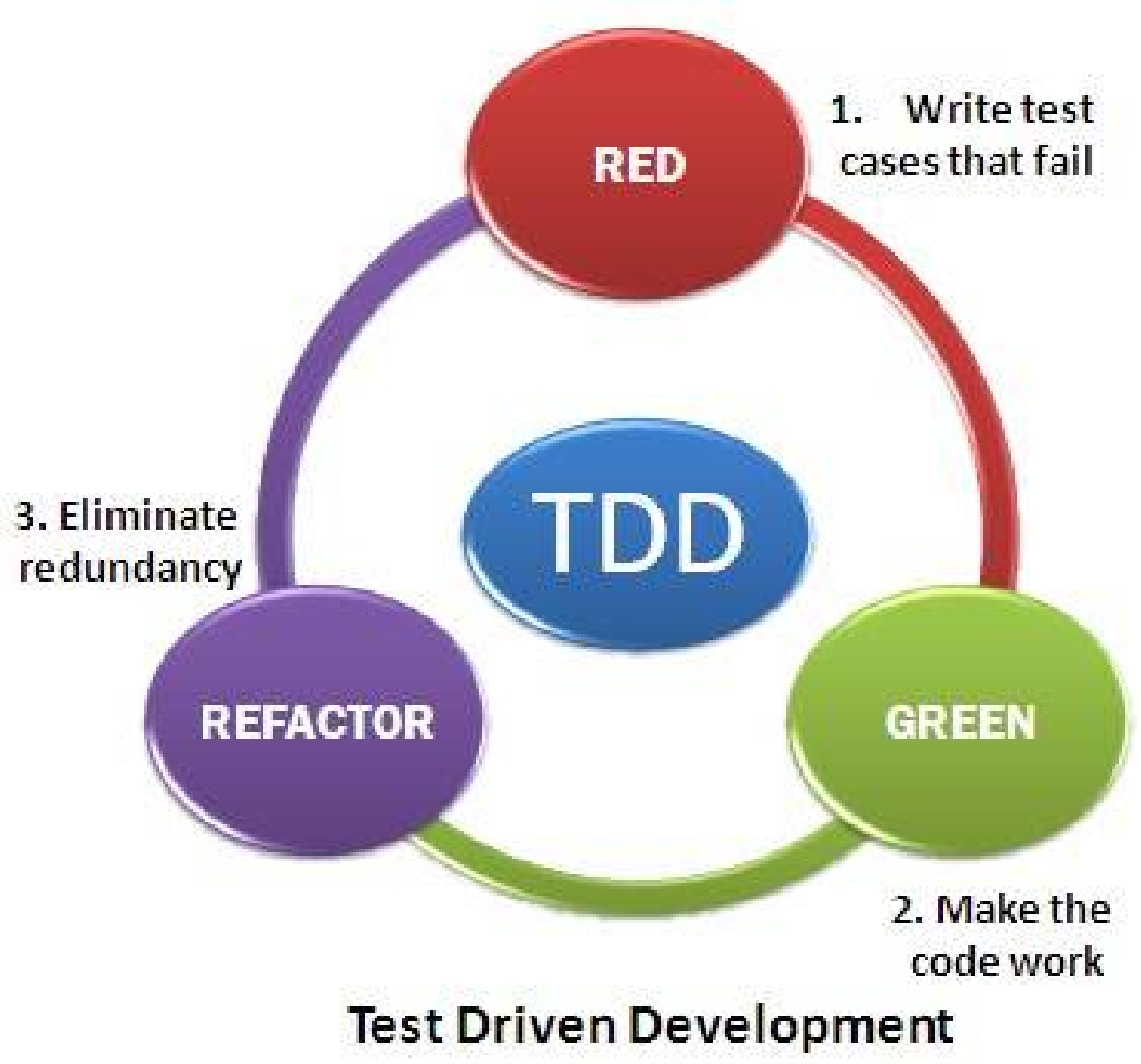
- Los errores se acumulan en los límites de los algoritmos
- Idea: probar los valores límite entre las clases de equivalencia
- Ejercicios
  - `boolean esEdadLaboral(int edad)`
  - `boolean esEdadLaboral(int edad, int añoactual)`

4.3. Conjetura de errores

- La experiencia da una idea de qué puede fallar
  - ¿ `0013462353` es un número de teléfono válido?
  - ¿ `+33 341234522,2,2` es un número de teléfono válido?
  - El número 1, ¿es primo?
  - ¿Podemos tener ciudadanos rusos en la base de datos?
  - ¿puedo leer un fichero con un emoji?

5. TDD

- *Test Driven Development*
  - Se deciden las pruebas que el *software* debe pasar
  - Se implementan dichas pruebas (rojo, ninguna funciona)
  - Se desarrolla el *software* hasta que se pasan las pruebas
  - Se revisa el código para mejorar el diseño
- Problemas
  - Es cortoplacista (se centra en solucionar pruebas)
  - No favorece el diseño con vista a largo plazo



6. JUnit

- JUnit permite automatizar pruebas unitarias y de integración
- Origen de facto de la familia XUnit: CCPUnit, NUnit, PyUnit...
- Muy popular: integración con prácticamente todas las herramientas e IDEs

6.1. Añadir JUnit

- Las clases de JUnit están en su fichero jar. También necesita el jar de Hamcrest.
  - Esto es para la versión 4.x
  - La versión 6 es bastante más complicada (para este curso, no compensa)
- Intellij: añadir los ficheros jar como librerías del proyecto/módulo
- En proyectos reales: aconsejable utilizar maven o gradle

6.2. Definir métodos de prueba

- Una prueba es un método anotado con `@Test`

- El método no recibe parámetros
- Se realizan varias pruebas con `Assert.assertXXXX`

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

import example.util.Calculator;

class CalculatorTest {

    private final Calculator calculator = new Calculator();

    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }
}
```

6.3. Aserciones

- Todas en https://junit.org/junit4/javadoc/4.8/org/junit/Assert.html

```
assertArrayEquals(expected, actual)
assertEquals(expected, actual)
assertFalse(boolean condition)
assertNotNull(Object object)
assertNotSame(Object unexpected, Object actual)
assertNull(Object object)
assertSame(Object expected, Object actual)
assertTrue(boolean condition)
fail(String message)
```

6.4. Excepciones

```
    @Test(expected = EmailFormatException.class)
    public void testEmaiVacio() throws EmailFormatException {
        EmailSeparator.separaEmail("");
    }
```

6.5. Más anotaciones

Anotacion	Descripcion
@Test	Indica que el método es un test
@DisplayName	Indica un nombre para el test class o el test method
@Tag	Define etiquetas para filtrar por tests
@Before	Se aplica a un método para indicar que se ejecute antes de cada método de prueba.
@After	Se aplica a un método para indicar que se ejecuta después de cada método de prueba.
@BeforeClass	Se aplica a un método static para indicar que se ejecuta antes que todos lo métodos de prueba de la clase.
@AtterClass	Se aplica a un método static para indicar que se ejecuta antes que todos lo métodos de prueba de la clase.
@Ignore	Ese aplica a un método de prueba para evitar esa prueba

6.6. Ejemplo: email

```
    /**
     * Separa un correo electrónico en dos partes: nombre de usuario y dominio.
     *
     * @param email La dirección de correo electrónico a separar.
     * @return Un array de cadenas donde el primer elemento
     *         es el nombre de usuario y el segundo es el dominio.
     *         Devuelve null si el formato del correo no es correcto.
     */
    public static String[] separaEmail(String email) {
        ...
    }
```

```java
public class EmailSeparator {
    public static String[] separaEmail(String email) {

        // Detectar la posición del símbolo '@'
        int atIndex = -1;
        for (int i = 0; i < email.length(); i++) {
            if (email.charAt(i) == '@') {
                atIndex = i;
            }
        }

        // Si no se encontró '@' o está al principio o al final
        if (atIndex <= 0 || atIndex >= email.length() - 1) {
            return null;
        }

        // Separar el nombre de usuario y el dominio
        String username = email.substring(0, atIndex);
        String domain = email.substring(atIndex + 1);

        // Verificar que el dominio no esté vacío
        if (domain.isEmpty()) {
            return null;
        }

        // Devolver el resultado en un array
        return new String[]{username, domain};
    }
}
```

6.7. Ejercicio: emails

- Comprender qué es una dirección de email
- Pensar en casos de prueba: AVL, clases de equivalencia...
- Implementar los casos de prueba
- Y solucionar los problemas del método `separaEmail()`

6.8. Ejercicio: polígonos

- El método `String tipoDePolígono(int[] longitudLados)`
- recibe la lista ordenada de la longitud de los lados de un polígono (el último se conecta con el primero)
- devuelve (en orden de prioridad)
  - `çuadrado"` si los lados pueden formar un cuadrado
  - `rectángulo"` si los lados pueden formar un rectángulo
  - `regular"` si los lados pueden forman un polígono regular
  - `ïregular"` si los lados pueden forman un polígono
  - `ïmposible"` si hay pocos lados para hacer un polígono, o algún otro problema
- Por ejemplo
  - `(2,2,2,2)` es un polígono regular y un cuadrado, pero es más prioritario `çuadrado"`
  - `(2,1,2,1)` es un polígono irregular y un rectángulo, pero es más prioritario `rectangulo"`
- Qué hacer:
  - Definir casos de prueba: AVL, clases de equivalencia, conjetura de errores...
  - Implementar los casos de prueba
  - Implementar el método

6.9. Ejercicio: puntos

```
public class Punto{
    public int x;
    public int y;

    public static Punto[][] masCercanos( Punto[] puntos ){
    }
}
```

- Devuelve los pares de puntos más cercanos (distancia euclídea)
- Por ejemplo `(1,2) (1,1) (2,11) (3,3) (4,3) (5,8) (100,0) (100,1)` devolvería un array de longitud 3. Cada posición es un array de dos puntos
  - `(1,2) (1,1)`
  - `(3,3) (4,3)`
  - `(100,0) (100,1)`
- Por ejemplo `(1,20) (18,1) (2,110) (3,3) (51,8) (100,0) (100,1) (4,5)` devolvería un array de longitud 1. Cada posición es un array de dos puntos
  - `(3,3) (4,5)`
- Pista: dibuja los puntos de los ejemplos para entenderlo
- Qué hacer:
  - Definir casos de prueba: AVL, clases de equivalencia, conjetura de errores...
  - Implementar los casos de prueba
  - Implementar el método

6.10. Ejercicio: matrículas

```
/**
 * Decide si una matrícula de coche (tipo 9999ZZZ) es anterior a otra.
 * No importan mayúsculas y minúsculas.
 * Tampoco importa si hay separación por espacios entre los números y las letras
 * @param m1 Matrícula española actual
 * @param m2 Matrícula española actual
 * @return "iguales" si son iguales
 *         "menor" si m1 es anterior a m2
 *         "mayor" si m1 es posterior a m2
 *         "error" si m1 o m2 no son matrículas válidas
 */
public static String comparaMatrícula(String m1, String m2){
    ....
}
```

- Qué hacer:
  - Definir casos de prueba: AVL, clases de equivalencia, conjetura de errores...
  - Implementar los casos de prueba
  - Implementar el método

6.11. Buenas prácticas

- Cada caso de prueba debe:
  - Probar una sola cosa
  - Ser lo más pequeño posible
  - Ser independiente: no debe depender de otros casos de prueba
  - Poder ser repetido las veces necesarias (idempotente)

7. Referencias

- Formatos:
  - Transparencias
  - PDF
  - Página web
  - EPUB
- Alojado en Github
- https://entornos.abrilcode.com/doku.php?id=apuntes:pruebas