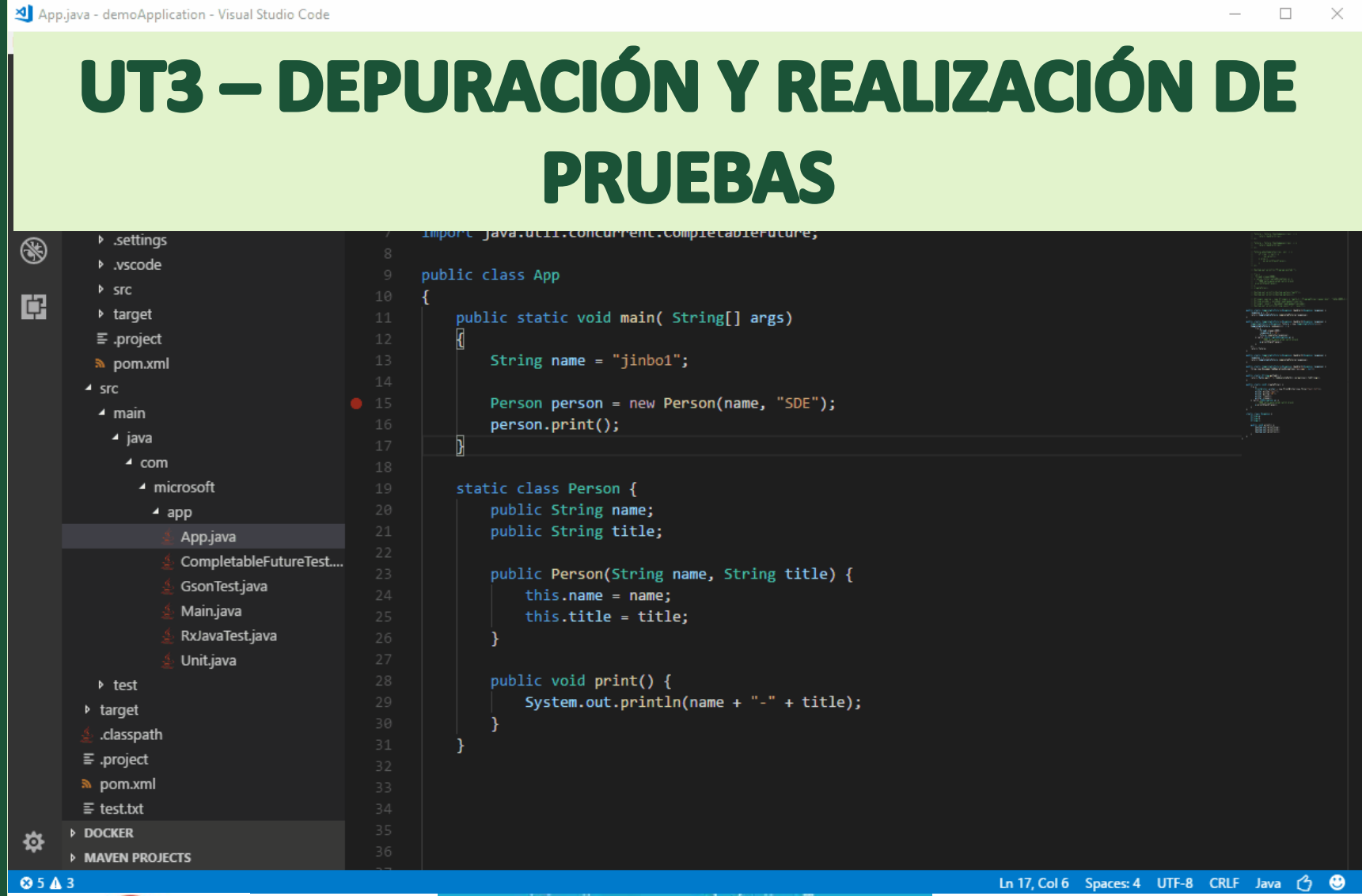
EFA MORATALAZ

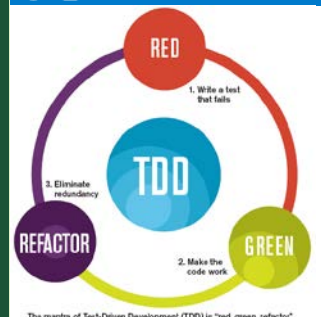
1º CFGS Desarrollo de Aplicaciones Multiplataforma

ENTORNOS DE DESARROLLO

YOLANDA MORENO Gª-MAROTO

UT3 – DEPURACIÓN Y REALIZACIÓN DE PRUEBAS





JUnit

EFA MORATALAZ

1º CFGS Desarrollo de Aplicaciones Multiplataforma

ENTORNOS DE DESARROLLO

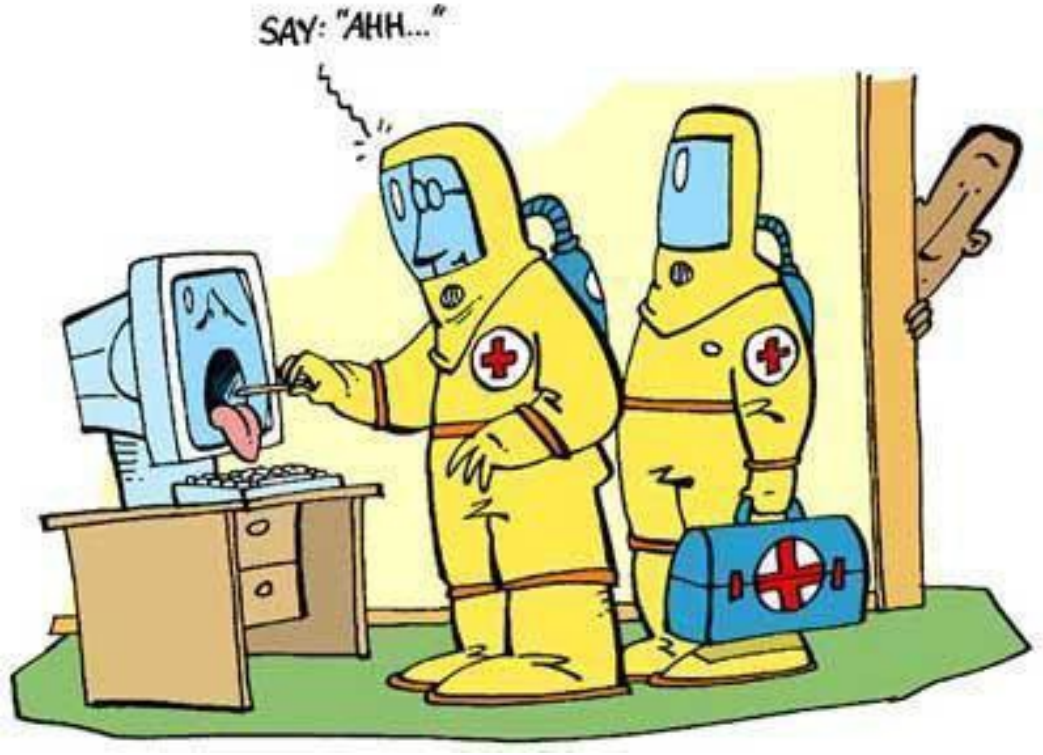
INDICE

UT3 – DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

1. PRUEBAS
2. PRUEBAS UNITARIAS
3. JUNIT

PRUEBAS

- ✓ Las **pruebas software** son un **conjunto finito** de casos de prueba
- ✓ Se establecen con el objetivo de determinar si la aplicación funciona correctamente según lo esperado para un **conjunto infinito** de ejecuciones de dominio



¿Qué se consigue mediante una fase de pruebas completa? **Verificar…**

- ✓ El correcto funcionamiento de los componentes del sistema
- ✓ El ensamblaje adecuado entre los distintos componentes
- ✓ El buen funcionamiento de las interfaces entre componentes y de las interfaces de comunicación con otros sistemas
- ✓ La integración completa del software en el hardware del entorno de operación
- ✓ El sistema desde el punto de vista de su funcionalidad y rendimiento
- ✓ Que los cambios en un componente no introducen cambios no previstos en otros componentes

- ✓ **Pruebas Unitarias**
- ✓ Pruebas de Integración
- ✓ Pruebas del Sistema
- ✓ Pruebas de Implantación
- ✓ Pruebas de Regresión

- ✓ Tienen como objetivo verificar la funcionalidad y estructura de cada componente individualmente una vez que ha sido codificado
- ✓ Constituyen las primeras pruebas a realizar en un sistema
- ✓ Los demás tipos de pruebas deben apoyarse sobre ellas
- ✓ Existen dos enfoques principales para el diseño de casos de prueba:
  - ❑ Pruebas unitarias de Caja Blanca
  - ❑ Pruebas unitarias de Caja Negra

- ✓ Verifican el correcto ensamblaje entre los distintos componentes una vez que han sido probados unitariamente
- ✓ S e busca comprobar que interactúan correctamente a través de sus interfaces, cubren la funcionalidad establecida y se ajustan a los requisitos
- ✓ Si aparecen errores en la aplicación, es muy probable que sea en la interfaz entre el nuevo componente y el resto de sus vecinos
- ✓ Estrategias de integración:
  - ❑ De arriba abajo (top-down)
  - ❑ De abajo arriba (bottom-up)
  - ❑ No incremental (Big-bang)

- ✓ Después de probar cada componente de forma individual y comprobar que se integran correctamente, se necesita probar el sistema de forma global
- ✓ En esta etapa se realizan diversas pruebas dependiendo de la tipología del producto software:
- ✓ **Pruebas funcionales**
  - Aseguran que el sistema de información realiza correctamente todas las funciones que se han detallado en las especificaciones dadas por el usuario del sistema
- ✓ **Pruebas de comunicaciones/interfaces**
  - Determinan que las interfaces entre los componentes del sistema funcionan adecuadamente. Asimismo, se han de probar las interfaces hombre/máquina
- ✓ **Pruebas de rendimiento**
  - Consisten en determinar que los tiempos de respuesta están dentro de los intervalos establecidos en las especificaciones del sistema

✓ Pruebas de volumen

- Examinan el funcionamiento del sistema cuando está trabajando con grandes volúmenes de datos, simulando las cargas de trabajo esperadas

✓ Pruebas de sobrecarga

- Se somete al sistema al umbral limite de los recursos introduciendo a cargas masivas. El objetivo es establecer los puntos extremos en los cuales el sistema empieza a operar por debajo de los requisitos establecidos

✓ Pruebas de disponibilidad de datos

- Se comprueba que el sistema es capaz de recuperarse ante fallos sin comprometer la integridad de los datos

✓ Pruebas de facilidad de uso

- Comprueban la adaptabilidad del sistema a las necesidades de los usuarios, asegurando que se acomoda a su modo habitual de trabajo y determinando las facilidades que se aportan para introducir datos y obtener los resultados

✓ Pruebas de operación

- Comprueban la correcta implementación de los procedimientos de operación, incluyendo la planificación y control de trabajos, arranque y rearranque del sistema, etc.

✓ Pruebas de entorno

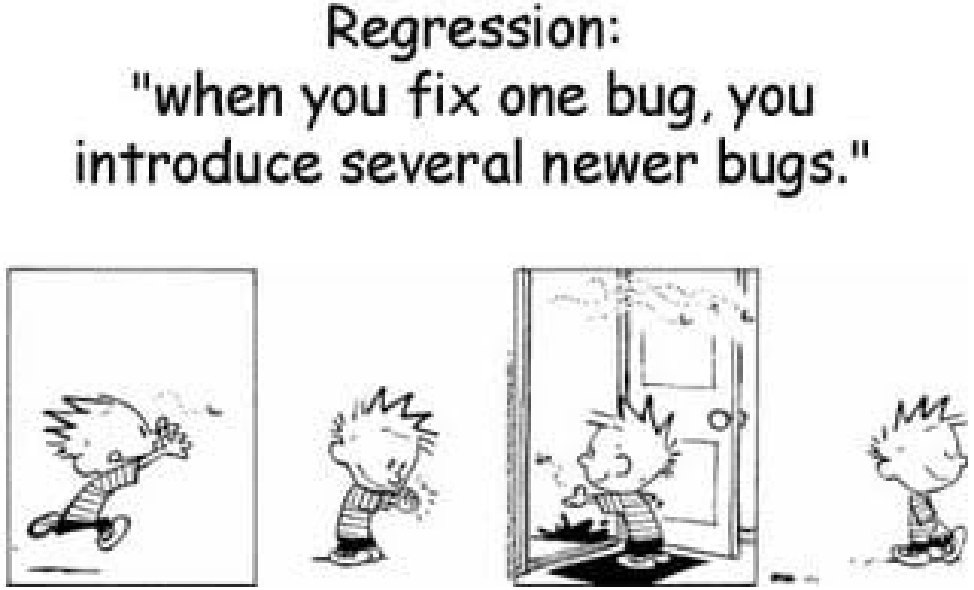
- Verifican las interacciones del sistema con otros sistemas del mismo entorno

✓ Pruebas de seguridad

- Verifican los mecanismos de control de acceso para evitar alteraciones indebidas de los datos

- ✓ Estas pruebas se realizan en el entorno de operación
- ✓ Permiten al usuario final comprobar el funcionamiento correctodel sistema integrado de hardware y software en un entorno similar al de producción
- ✓ El usuario final debe comprobar que el sistema responde satisfactoriamente a los requisitos de rendimiento, seguridad, usabilidad, operación y coexistencia con el resto de los sistemas de la instalación

- ✓ S e comprueba que los cambios sobre un componente de un sistema de información, no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados
- ✓ Este tipo de pruebas implica la repetición de las pruebas que ya se han realizado previamente

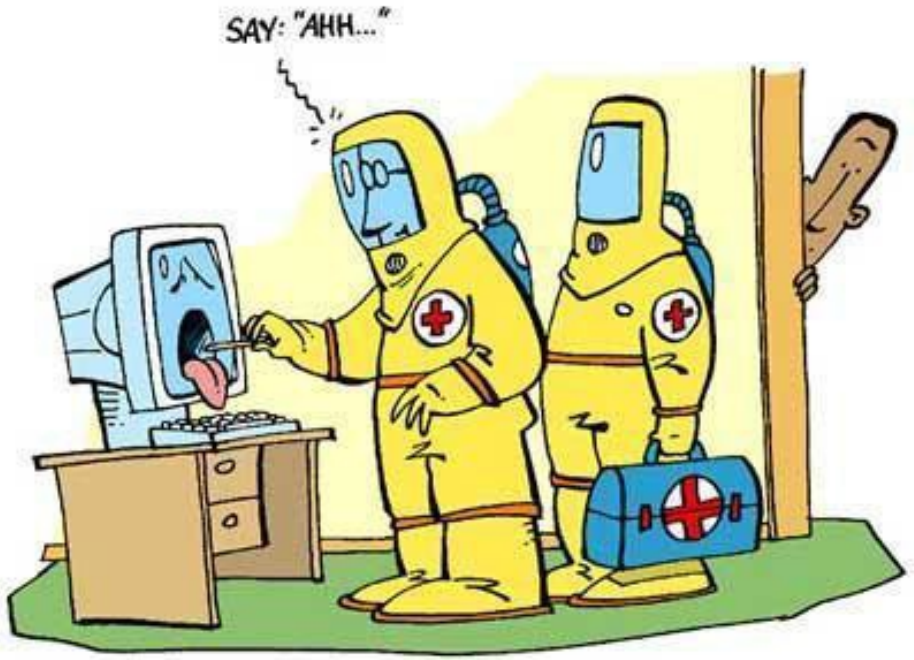


PRUEBAS UNITARIAS

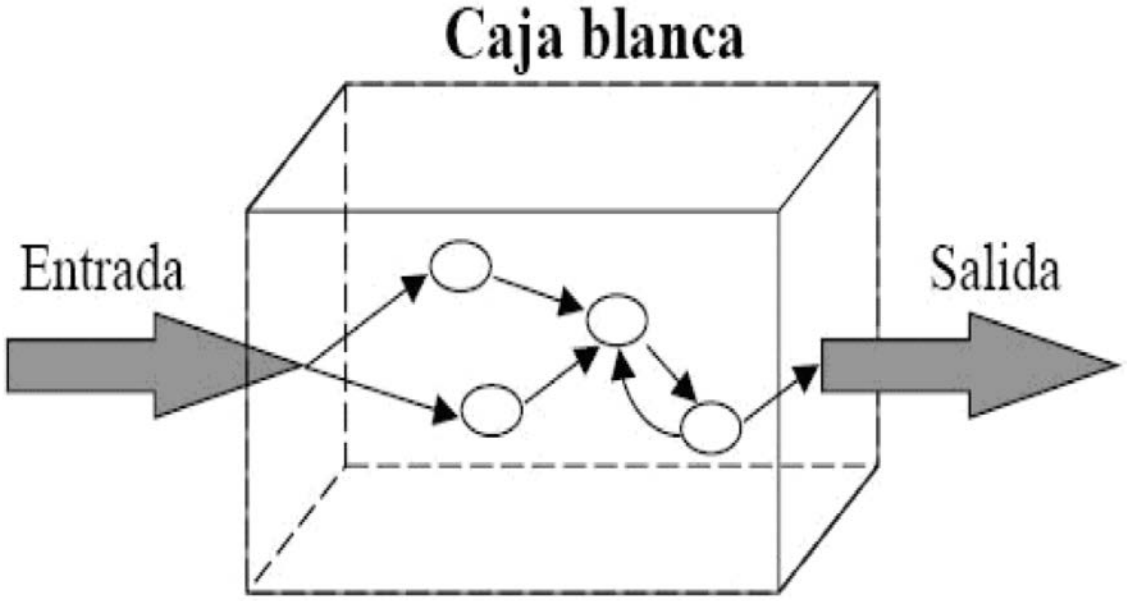
- ✓ S on una serie de condiciones establecidas con el objetivo de determinar si la aplicación funciona correctamente según lo definido en la **especificación del análisis de requisitos**
- ✓ Para cada tarea, pueden surgir diversos casos de prueba, teniendo en cuenta todos los factores posibles para no dejar ningún cabo suelto sin probar y evitar así que ocurran errores no conocidos

ID	¿qué?	descripción	requisitos	Res esperado	Res obtenido
001	App móvil	Carga fichero	Que exista	OK / KO	OK / KO

- ✓ Las pruebas unitarias pueden ser de varios tipos:
  - ✓ Caja blanca
  - ✓ Caja negra
  - ✓ Rendimiento
  - ✓ Coherencia



- ✓ Las pruebas de caja blanca se centran en el **funcionamiento interno** del programa
- ✓ S e observa y comprueba cómo se realiza una operación
- ✓ Existen:
  - Prueba del camino básico
  - Prueba de condiciones
  - Prueba de bucles



- ✓ Se basa en el principio de que cualquier diseño procedimental se puede representar mediante un grafo de flujo
- ✓ Consiste en definir los diferentes caminos para llegar desde el inicio hasta la solución
- ✓ De esta forma sabremos los flujos de código que se tienen que probar
  - ✓ Ejemplo: Un programa comprueba si dos números son pares y, de ser así comprueba la suma de ambos. Si los dos son impares, comprueba el producto de ambos. Y si no es ninguna de las anteriores (dos pares o dos impares), comprueba la resta de ambos.

DIAGRAMA DE FLUJO

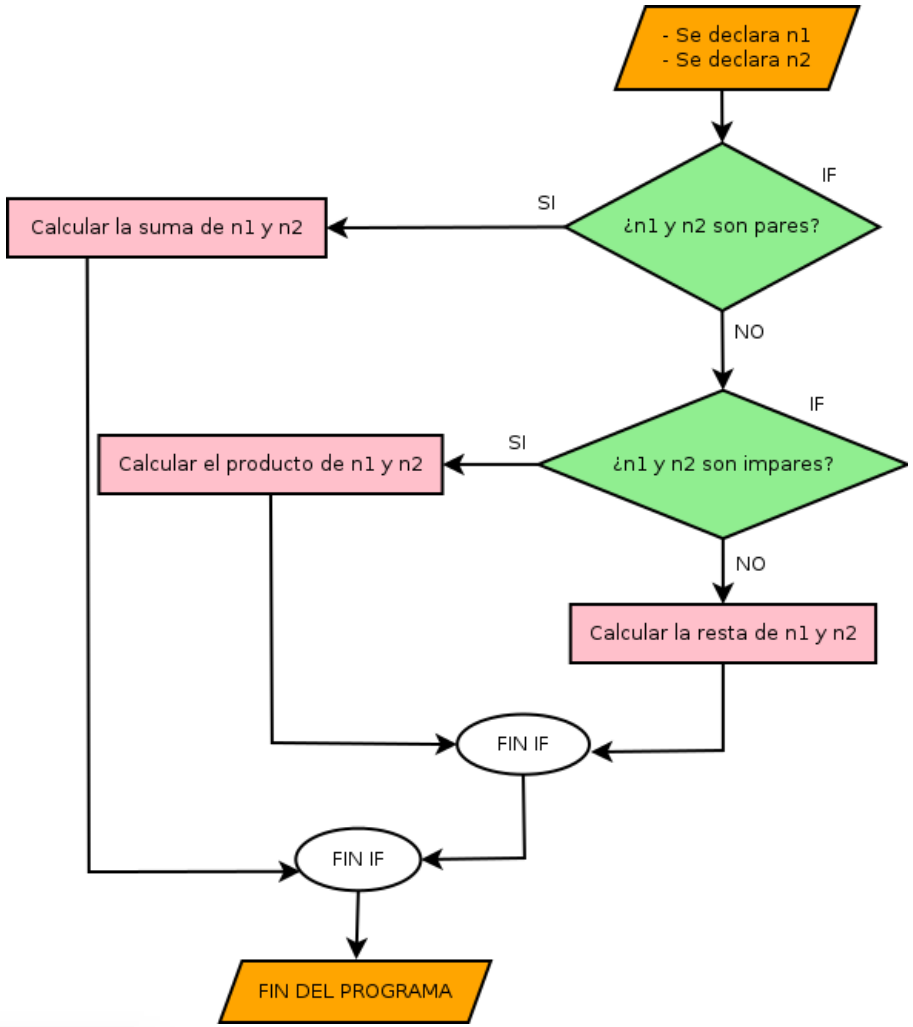
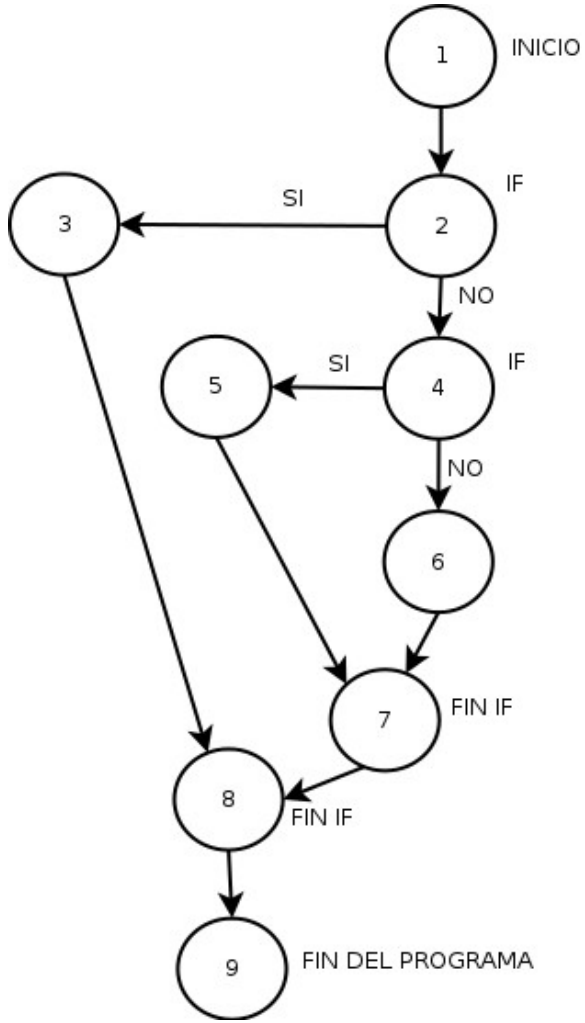


DIAGRAMA DE NODOS



Ejemplo diagrama de flujo:

```
int isFreaky(int videogames,
    int manga, int technology){
    if(videogames>0){
        if(manga>0){
            if(technology>0){
            return 1;
            }
            else{
            return 0;
            }
        else{
        return 0;
        }
    else{
    return 0;
    }
}
```

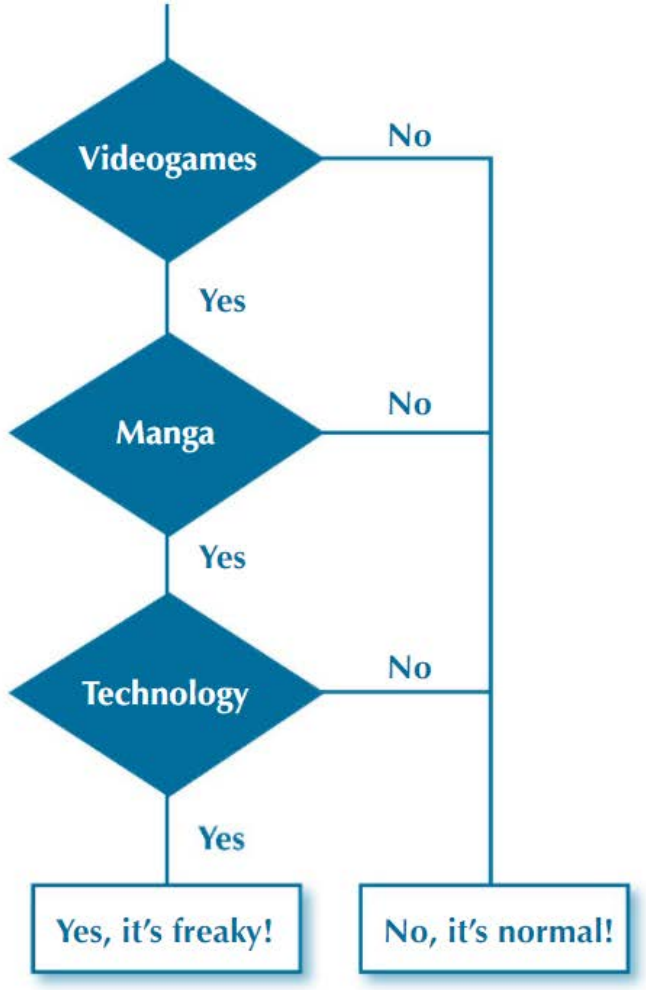


Figura 3.2
Diagrama de flujo de una función.

COMPLEJIDAD CICLOMÁTICA

- La complejidad ciclomática mide el número de caminos independientes dentro de nuestro código que es sometido a prueba, una forma de calcularla es:
  - V(G) = a – n + 2

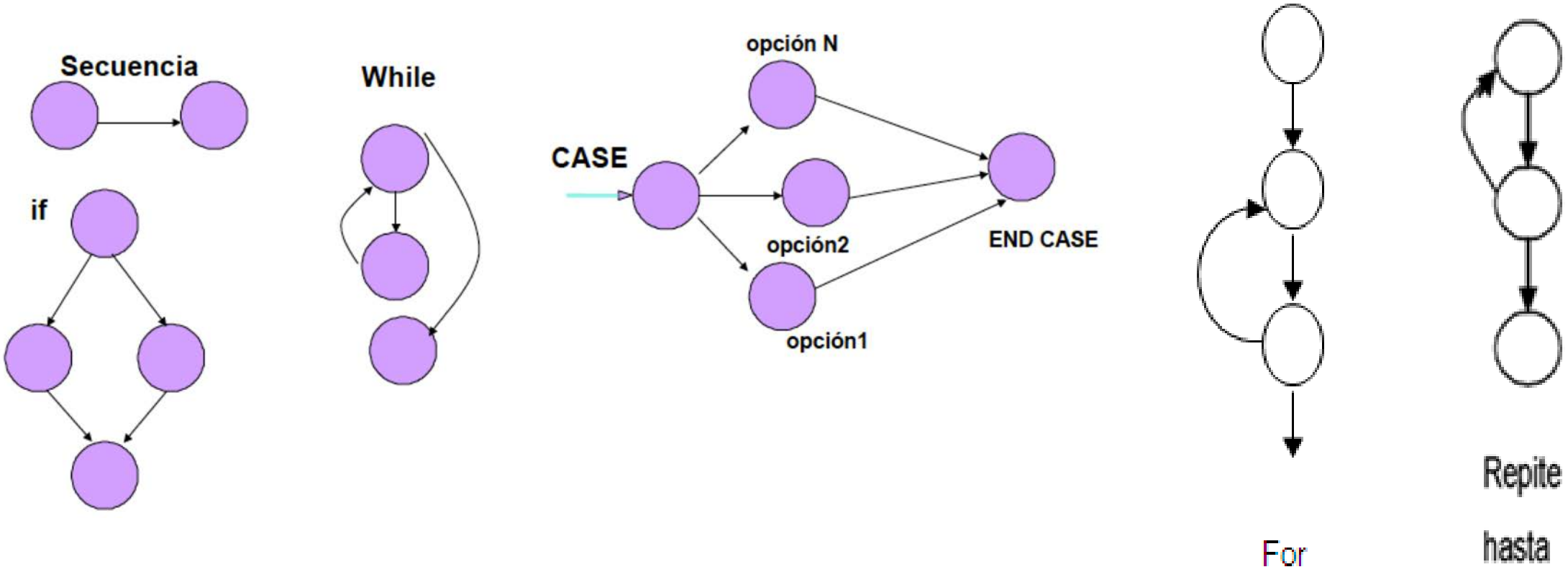
a: número de aristas del grafo

a: número de nodos

1. Se crea el Diagrama de Flujo
2. Se crea el Diagrama de Nodos
3. Se determinan los caminos posibles
   - ✓ De esta forma se conocen las opciones que habrá que probar mediante los diferentes mecanismos de pruebas

- ✓ Según el enunciado se tendrían entonces tres caminos distintos (complejidad ciclomática = 3)
  - ✓ 1, 2, 3, 8, 9
  - ✓ 1, 2, 4, 5, 7, 8, 9
  - ✓ 1, 2, 4, 6, 7, 8 , 9

✓ Diferentes estructuras aplicables para el diagrama de nodos



- ✓ Se evalúa la construcción de los condicionales
- ✓ Es necesario evitar las condiciones redundantes *(condiciones cortocircuitadas)*
  - ✓ ***Ejemplo 1:*** *if(E1 == true || E2 == true)*
  - ✓ *Solo se debe evaluar el valor de E2 cuando el valor de E1 sea falso*

A	B	Salida
0	0	0
0	1	1
1	0	1
1	1	1

- ✓ ***Ejemplo 2:*** *if(alumno != null && alumno.getNombre().equals("Manolo"))*
- ✓ *Evitar NullPointerExcepFon*

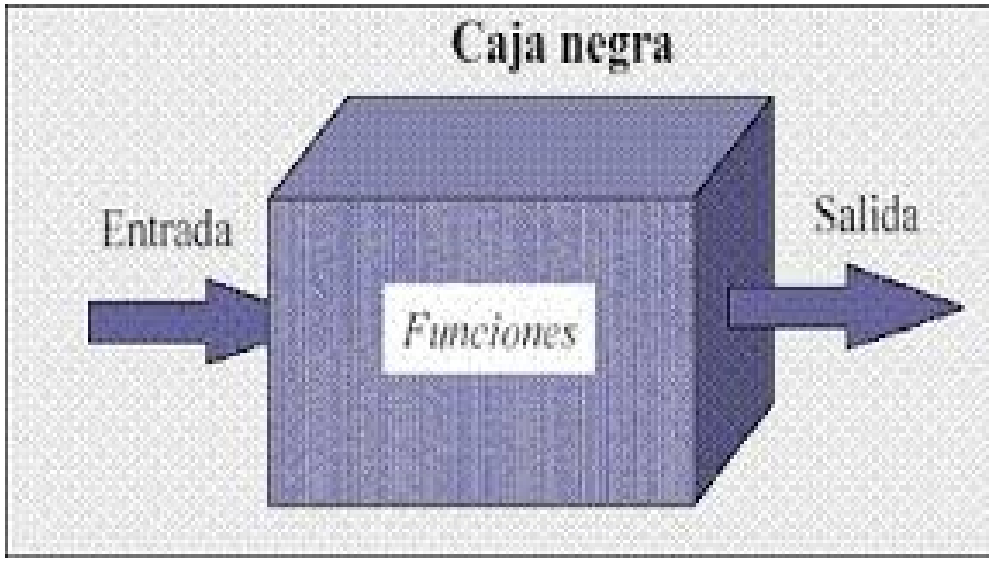
- ***Ejemplo 3:***

```
if (videogames=1 && manga=1 && technology=1){ freaky = 1}
```

- Se deberán comprobar todas y cada una de las combinaciones de las tres variables.

- ✓ Consiste en evaluar todas las opciones posibles para un bucle de ‘n’ iteraciones.
  - ✓ El flujo del programa no entra ninguna vez al bucle
  - ✓ Pasa una única vez por el bucle
  - ✓ Pasa dos veces por el bucle
  - ✓ Pasa m veces por el bucle, donde m<n
  - ✓ Hacer n-1 y n iteraciones en el bucle

- ✓ Estas pruebas se enfocan en la información de entrada y salida de la aplicación
- ✓ Se validan y controlan los datos de entrada para evitar errores
- ✓ Se verifican los datos de salida y se contrastan con los resultados esperados
- ✓ Existen:
  - ✓ Clases de equivalencia
  - ✓ Análisis de valores límite
  - ✓ Pruebas de interfaces



✓ Criterios para identificar las clases de equivalencia

Condición entrada	Ejemplo	Clases de equivalencia válidas	Clases de equivalencia no válidas
Un valor específico	"..Introducir **cinco** valores.."	1 clase que contemple dicho valor	2 clases que representen un valor por encima y otro por debajo
Un rango de valores	...Valores **entre 0 y 10**…	1 Clase que contemple los valores del rango	2 clases fuera del rango, una por encima y otra por debajo
Un valor enumerado	Introducir un día de la semana	1 Clase que contemple un valor del enumerado	1 clase que contemple un valor fuera del enumerado
	Introducir un número **entero**	1 clase que contemple los elemento de ese tipo	N clases que representen tipos distintos(Nulo, alfanuméricos, fraccionarios)
Condición lógica	Introducir un número **par**	**1 clase que cumpla la condición**	**1 clase que no cumpla**
Elementos de un conjunto tratados diferente por el programa	**Las personas menores de 25 años tendrán una bonificación del 10%**	**1 clase que cumpla la condición**	

Ejemplo 1:

- ✓ *Se tiene un programa que valida si es correcto un número de teléfono móvil. La salida del programa será una cadena de texto que indique:*
  - ✓ ***MÓVIL VÁLIDO:*** *En caso de ser un número de móvil válido*
  - ✓ ***MÓVIL INVÁLIDO:*** *En caso de ser un número de móvil no válido*
  - ✓ ***MÓVIL INEXISTENTE:*** *En caso de que el programa no reciba datos de entrada*

- ✓ Este método divide y separa los campos de entrada según el tipo de dato y las restricciones que conllevan

Condición	Clases válidas	Clases inválidas
Comprobar número de móvil válido	1. Numero de 9 dígitos que comience en 6 2.Numero de 9 dígitos que comience en 7	3. Número menor de 9 dígitos 4. Número mayor de 9 dígitos 5. Número negativo 6. No es número 7. Cadena Nula

- ✓ Estos serían los casos de prueba resultantes

Caso de prueba	Clases válidas	Clases inválidas	Salida
625789478	1.	---	Móvil válido
721457789	2.	---	Móvil válido
1234	---	3.	Móvil inválido
145879634789	---	4.	Móvil inválido
-459789	---	3, 5	Móvil inválido
	---	7	Móvil sin especificar
DAM	---	6	Móvil inválido

- ✓ Ejemplo 2: el usuario podrá introducir un nombre de usuario y contraseña, el usuario deberá tener mayúsculas, minúsculas y al menos 6 letras. No podrá tener otro tipo de caracteres. La contraseña tendrá entre 8 y 10 caracteres (letras y números).

1. *Usuario:*
   - *Clases válidas:* "Pelegrino" y "Rocinante".
   - *Clases inválidas:* "marrullero44", "nene", "Portaavionesgigante", "Z&aratustra" y "Ventajoso12".
2. *Contraseña:*
   - *Clases válidas:* "5Entrevias" y "s8brino".
   - *Clases inválidas:* "corta","muyperoquemuylarguisima","oletugarbo"y"999999999".

- Es una técnica complementaria a las clases de equivalencia, pero centrada en los valores límite
- Indica que si especificamos un rango delimitado de valores o un número de valores específicos, también se deberá probar por el valor inmediatamente superior e inmediatamente inferior de dichas cotas
- El objeto de esta prueba se halla en comprobar que están bien puestos los límites en las fronteras (<, <=, >, >=)
  - Si tenemos que los valores básicos estarán entre a y b, habrá que probar los valores a-1, a+1, b-1 y b+1
  - Ejemplo: Edad de población activa, entre 18 y 65 años
    - Los valores a introducir serían 17, 19, 64 y 66

✓ Obtención de los casos de prueba de Valores límite:

Condición entrada	Ejemplo	Caso de prueba
Un valor específico	"..Introducir **tres** valores.."	1 caso que ejercite el valor numérico (15,3,4) 1 caso que ejercite el valor justo por encima (15,3,4,4) 1 caso que ejercite el valor justo por debajo(15,3)
Un rango de valores	...Valores **entre 0 y 10**…	1 caso que ejercite el valor mínimo (0) 1 caso que ejercite por encima del mínimo (1) 1 caso que ejercite por debajo del mínimo (-1) 1 caso que ejercite el valor máximo (10) 1 caso que ejercite un valor por encima del máximo (11) 1 caso que ejercite un valor por debajo del máximo (9)
Elementos de un conjunto tratados diferente por el programa	**Las personas menores de 25 años tendrán una bonificación del 10%**	1 caso que cumpla la condición (25) 1 caso que ejercite el valor justo por encima (26) 1 caso que ejercite el valor justo por debajo(24)

A) Cómo testear una interfaz

Una primera prueba puede consistir en seguir el manual de usuario. El *tester* deberá introducir datos (mejor datos reales que inventados) como si se tratase del propio usuario y comprobar que las salidas proporcionadas son las esperadas.

Si el software pasa esta prueba, entonces, podrá pasar a sufrir un testeo más serio utilizando casos de prueba.

B) Testear la usabilidad

Tiene por objeto evaluar si el producto generado va a resultar lo esperado por el usuario. Hay que ver y trabajar con la interfaz desde el punto de vista del usuario. Además, en su testeo, deberían utilizarse datos reales.

Solamente, al observar cómo el usuario interactúa con el software y escuchando su *feedback*, pueden detectarse aquellas características de este que lo hacen difícil y tedioso de utilizar. Una vez detectadas esas disfunciones, se realizarán los cambios pertinentes, de tal manera que el software sea fácil de usar y eficiente. Es importante escuchar la opinión del cliente porque, a la postre, es la persona que va a trabajar de forma sistemática con el software.

C) Testear la accesibilidad

Mucha gente no sabe que la accesibilidad no solamente es que el software esté diseñado para usuarios con discapacidad, sino que también sea accesible por *frameworks* de test automatizados.

Un software es accesible cuando el programa o aplicación se adecua a los usuarios con discapacidad, pueden hacer su trabajo de forma efectiva y la satisfacción con él es buena. Además, hay que tener en cuenta que, en ocasiones, hay estándares y requerimientos preestablecidos de accesibilidad que el software ha de cumplir.

- ✓ Las pruebas de rendimiento miden el tiempo que le ha tomado a la aplicación realizar una acción específica
- ✓ Depende en gran medida de la máquina en donde se esté ejecutando la aplicación, por lo que todas las medidas habría que hacerlas con la misma máquina
- ✓ Se realizan acciones equivalentes codificadas de diferente forma y se ve cuál conlleva un mejor rendimiento, es decir tarda menos tiempo

- Ejemplo: Tengo que mostrar por pantalla los datos personales de la población activa de Manzanares.
  - Podemos codificar en Java un método que acceda a una base de datos MySQL
  - Podemos codificar en Java un método que acceda a un fichero XML que contenga toda esa información

```
public static void main(String[] args) {
  // guardar timestamp inicio
  long start = System.currentTimeMillis();

  // hacer algo
  for (int a=1; a<=1000000; a++)
  {
    System.out.println(a);
  }

  // calcular tiempo transcurrido
  long end = System.currentTimeMillis();
  long res = end - start;
  System.out.println("Segundos: "+res/1000);
}
```

- ✓ Las pruebas de coherencia son subjetivas, es decir, no están centradas en los datos, sino en el flujo de trabajo de la aplicación de un modo coherente
- ✓ **Se comprueba si la funcionalidad de la aplicación es correcta y no si la aplicación funciona correctamente**
- ✓ Es decir, que una aplicación no tenga ningún error en el código o la interfaz no quiere decir que funcione correctamente
- ✓ *Ejemplo: Una web de pedidos tiene un campo observaciones, donde el cliente puede añadir información adicional sobre el pedido. ¿Tiene sentido que una vez realizado el pedido el cliente pueda seguir rellenando este campo?*

JUNIT

- ✓ Es un framework para Java enfocado en la realización de pruebas unitarias (unit testing). Consiste en unas librerías JAR que debemos añadir a un proyecto en Java.
- ✓ Existen varios plugins para poder utilizar con diferentes Entornos de Desarrollo Integrado (**IDE**).
- ✓ Con JUnit, ejecutar tests es tan fácil como compilar tu código. El compilador **"testea"** la sintaxis del código y los tests **"validan"** la integridad del código.
- ✓ Los tests realizados se podrán presentar junto con el código, para validar el trabajo realizado.

Qué debemos probar

- ✓ Los tests unitarios se usan para evitar el miedo a que algo se rompa. Cualquier método que no tengamos claro si funciona de forma correcta o no, es un método susceptible de ser probado de forma unitaria.
- ✓ Los métodos getters y setters normalmente son tan sencillos que no suelen albergar dudas, pero si hay alguna razón para que no funcionen, entonces también se deben probar.
- ✓ En resumen, se debe probar todas las cosas hasta que todas nos inspiren confianza, basándonos en nuestro criterio.
- ✓ En caso de querer comprobar todos los requisitos de un programa se deben al menos plantear dos casos de prueba para cada requisito o funcionalidad: una prueba afirmativa y una prueba negativa.

Métodos de prueba vs Casos de prueba

- ✓ Un método de prueba es simplemente un fragmento de código que realiza una comprobación sobre una funcionalidad de un programa. Para que un método sea un método de prueba debe estar precedido por la anotación: @Test
- ✓ Un caso de prueba (test case), como hemos dicho es una situación o caso (un estado concreto del programa) bajo el que se prueba una funcionalidad del programa.

JUnit en eclipse

- ✓ Se pueden crear JUnit Test Case y JUnit Test Suites.
- ✓ JUnit Test Case (caso de prueba) son clases o módulos con métodos para probar de métodos de un módulo o una clase concreta.
- ✓ JUnit Test Suites (suite de prueba) nos sirven para organizar los casos de prueba, de forma que cada suite agrupa los casos de prueba de módulos que están funcionalmente relacionados.
- ✓ JUnit5 ofrece herramientas para poder ejecutar tests de las versiones anteriores (JUnit4 y JUnit3), por lo que JUnit5 puede ejecutar tests escritos en las versiones anteriores.

JUnit en eclipse (ejemplo de clase de pruebas)

```java
public class ClaseTest {

	@BeforeAll
	public static void setUpBeforeClass() throws Exception {
		//Codigo que se ejecuta antes de cualquier prueba
	}

	@BeforeEach
	public void setUp() throws Exception {
		//Codigo que se ejecuta antes de cada prueba
	}

	@Test
	public void test1() {
		//Caso de prueba
	}

	@Test
	public void test2() {
		//Caso de prueba
	}

	@Test
	public void test3() {
		//Caso de prueba
	}

	@AfterEach
	public void tearDown() throws Exception {
		//Metodo que se ejecuta después de cada prueba
	}


	@AfterAll
	public static void tearDownAfterClass() throws Exception {
		//Codigo que se ejecuta después de todas las pruebas
	}
}
```

JUnit en eclipse (ejemplo de clase de pruebas)

Si tenemos una clase `Matematicas` con distintos métodos:

```
public class Matematicas{
   public static int suma(int num1, int num2){
      return num1 + num2;
   }
}
```

Y creamos una clase `MatematicasTest` con métodos de prueba para probar los métodos.

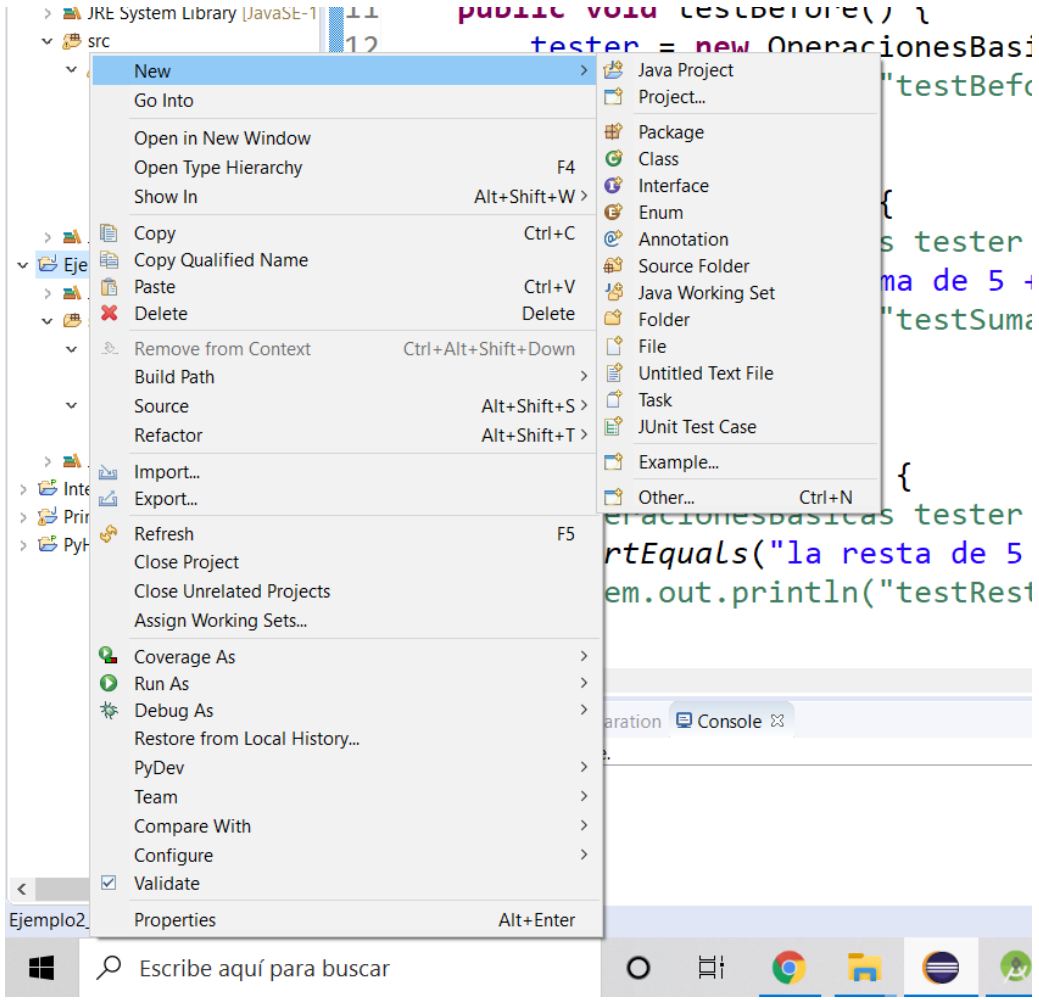
```
class MatematicasTest {
   @Test
   void testSuma() {
      int actual= Metodos.suma(7,0);
      int esperado = 7;
      assertEquals(esperado, actual);
   }
}
```

JUnit en eclipse

- Ejemplo de uso:
  - Una vez creado el proyecto y nuestra clase o métodos, crearemos nuestra clase de prueba (JUnit Test Case).
  - Para ello sobre el proyecto o paquete donde queramos crearla:
    - Botón derecho→new→JUnit Test Case

JUnit en eclipse

- ✓ En los siguientes pasos nos pedirá que escojamos la versión de JUnit (elegimos JUnit 4), el nombre de la clase de pruebas y la clase sobre la que vamos a realizar las pruebas.
- ✓ También podemos decirle que nos cree algunos métodos por defecto pero de momento no marcaremos ninguno.



Anotaciones JUnit 4

- ✓ JUnit 4 se basa en anotaciones para determinar los métodos de a testear así como para ejecutar código previo a los tests a realizar. En la tabla a continuación se muestran las anotaciones disponibles:

Anotación	Descripción
@Test public void method()	La anotación @Test identifica el método como método de test.
@Test (expected = Exception.class)	Falla si el método no lanza la excepción esperada.
@Test(timeout=100)	Falla si el método tarda más de 100 milisegundos.
@Before public void method()	Este método es ejecutado antes de cada test. Se usa para preparar el entorno de test (p.ej., leer datos de entrada, inicializar la clase).
@After public void method()	Este método es ejecutado después de cada test. Se usa para limpiar el entorno de test (p.ej., borrar datos temporales, restaurar valores por defecto). Se puede usar también para ahorrar memoria limpiando estructuras de memoria pesadas.

Anotaciones JUnit 4

@BeforeClass public static void method()	Este método es ejecutado una vez antes de ejecutar todos los test. Se usa para ejecutar actividades intensivas como conectar a una base de datos. Los métodos marcados con esta anotación necesitan ser definidos como static para trabajar con JUnit.
@AfterClass public static void method()	Este método es ejecutado una vez después que todos los tests hayan terminado.Se usa para actividades de limpieza, como por ejemplo, desconectar de la base de datos. Los métodos marcados con esta anotación necesitan ser definidos como static para trabajar con JUnit.
@Ignore	Ignora el método de test. Es útil cuando el código a probar ha cambiado y el caso de uso no ha sido todavía adaptado. O si el tiempo de ejecución del método de test es demasiado largo para ser incluído.

Métodos assert Junit (Clase Assertions JUnit5 / Assert JUnit4)

- ✓ **Las aserciones permiten verificar el comportamiento de nuestro método**. En caso de incumplir dichas aserciones el método de prueba fallará identificando el error producido.
- ✓ La clase Assertions contiene los métodos principales de las librerías de JUnit, y los usamos para realizar las comprobaciones en los método de prueba. Son métodos estáticos que no devuelven nada, pero hacen que JUnit nos diga si una prueba falla o no.
- ✓ Se usan para afirmar una condición como resultado del funcionamiento esperado de una parte del código. Si no se cumple lo que se afirma, la prueba falla y JUnit nos lo indica.

JUnit4 vs Junit 5

FEATURE	JUNIT 4	JUNIT 5
Declare a test method	`@Test`	`@Test`
Execute before all test methods in the current class	`@BeforeClass`	`@BeforeAll`
Execute after all test methods in the current class	`@AfterClass`	`@AfterAll`
Execute before each test method	`@Before`	`@BeforeEach`
Execute after each test method	`@After`	`@AfterEach`
Disable a test method / class	`@Ignore`	`@Disabled`
Test factory for dynamic tests	NA	`@TestFactory`
Nested tests	NA	`@Nested`
Tagging and filtering	`@Category`	`@Tag`
Register custom extensions	NA	`@ExtendWith`

JUnit4 vs JUnit 5

Las siguientes anotaciones de JUnit 5 se emplean en la cabecera de cada método de test:

Anotación	Descripción
@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
@BeforeEach	Se aplica a un método para indicar que se ejecute antes de cada método de prueba. (JUnit4 @Before)
@AfterEach	Se aplica a un método para indicar que se ejecuta después de cada método de prueba. (JUnit4 @After)
@BeforeAll	Se aplica a un método `static` para indicar que se ejecuta antes que todos lo métodos de prueba de la clase. (JUnit4 @BeforeClass)
@AfterAll	Se aplica a un método `static` para indicar que se ejecuta antes que todos lo métodos de prueba de la clase. (JUnit4 @AfterClass)
@Disable	Ese aplica a un método de prueba para evitar esa prueba (JUnit4 @Ignore)

JUnit 5: Aserciones

La clase `Assertions` contiene los métodos principales de las librerías de JUnit, y los usamos para realizar las comprobaciones en los método de prueba. Son métodos estáticos que no devuelven nada, pero hacen que JUnit nos diga si una prueba falla o no.

Metodo	Descripción
assertTrue(boolean valor)	Falla si valor no es `true`
assertFalse(boolean valor)	Falla si valor no es `false`
assertFalse(boolean valor, String mensaje)	Falla si valor no es `false` y muestra el mensaje
assertEquals(int esperado, int actual)	Falla si `esperado` es distinto de `actual`
assertEquals(int esperado, int actual, String mensaje)	Falla si `esperado` es distinto de `actual`
assertEquals(double esperado, double actual, double delta)	Falla si la diferencia entre `esperado` y `actual` es mayor a `delta`
assertNull(Object obj)	Falla si `obj` es distinto de `null`
assertNotNull(Object obj)	Falla si `obj` es `null`
assertEquals(Object esperado, Object actual)	Falla si los objetos son distintos, evaluando su método `equals()`
assertNotEquals(Object esperado, Object actual)	Falla si los objetos no son distintos, evaluando su método `equals()`
assertSame(Object esperado, Object actual)	Falla si las referencias de los objetos sin distintas

@BeforeEach @AfterEach ejemplo:

Actúan sobre un método para indicar que se ejecutará antes o después de ejecutar cada caso de prueba. Se usa para establecer el contexto del test.

```
public class ClaseTest{

//Una vez antes de cada test
@BeforeEach
void setUp(){
   //Me aseguro que antes de cada test solo haya un vehiculo en la lista
   listaVehiculos.clear();
   listaVehiculos.add(new Vehiculo("ASD-123");
}

//Una vez después de cada test
@AfterEach
void tearDown(){
   listaVehiculos.clear();
}

...
//Resto de métodos de prueba
```

@BeforeEach @AfterEach ejemplo: `@Before` y `@After` en JUnit4.

Actúan sobre un método para indicar que se ejecutará antes o después de ejecutar cada caso de prueba. Se usa para establecer el contexto del test.

```
public class ClaseTest{

//Una vez antes de cada test
@BeforeEach
void setUp(){
   //Me aseguro que antes de cada test solo haya un vehiculo en la lista
   listaVehiculos.clear();
   listaVehiculos.add(new Vehiculo("ASD-123");
}

//Una vez después de cada test
@AfterEach
void tearDown(){
   listaVehiculos.clear();
}

...
//Resto de métodos de prueba
```

@BeforeAll @AfterAll ejemplo:

`@AfterClass` y `@BeforeClass` en Junit4.

Se realiza una sola vez para toda la clase: al iniciar la clase o al terminarla. Se suele usar para abrir recursos: creación de instancias, conexiones, apertura de ficheros, etc.

La diferencia con las anotaciones anteriores, es que @BeforeAll y @AfterAll actúan sobre métodos `static`.

```
public class ClaseTest

   static FileWriter escritor;

   static GestorVehiculos gestor;

   //Se ejecuta al inicio de la clase
   @BeforeAll
   static void setUpBeforeClass() throws Exception {
      gestor = new GestorVehiculo()
   }

   //Se ejecuta al final de la clase
   @AfterAll
   static void tearDownAfterClass() throws IOException{
      escritor.close();
   }

   ...
   //Resto de métodos de prueba
```

JUnit4 vs Junit 5

- ✓ **Nombrado de los test**

```
Shell
1  @DisplayName("Aserciones soportadas")
2  class AssertionsTest {
3
4      @Test
5      @DisplayName("Verdadero o falso ?")
6      void true_or_false() {
7          ...
8      }
9
10     @Test
11     @DisplayName("Existencia de un objeto")
12     void object_existence() {
13         ...
14     }
15 }
```

JUnit4 vs Junit 5

- ✓ **Nombrado de los test**

```
@DisplayName
```

para indicar el nombre del test mediante un texto libre (como cosa curiosa vemos que podemos usar emoticonos dentro del

```
@DisplayName
```

).

Esto nos sirve para que las herramientas presenten este texto en lugar del nombre del método.

JUnit4 vs Junit 5

- ✓ **Nombrado de los test**

Una cosa que puede pasar desapercibida del código anterior es que ni la clase y ni los métodos de test necesitan ser públicos (

```
public
```

).