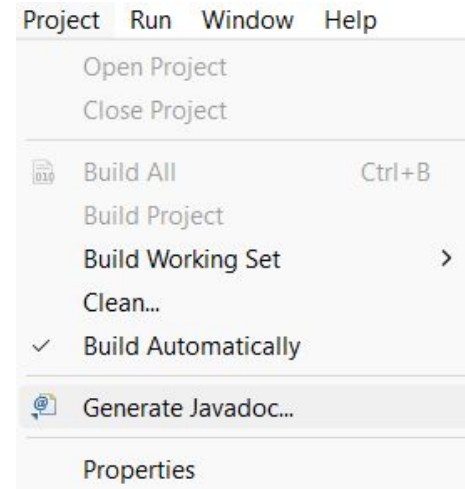


# ED. Documentación y pruebas

# Documentación - Javadoc

- Generando código con Papyrus obtenemos los bloques que utiliza Javadoc para una correcta documentación. Todos empiezan con `/**` y acaban con `*/`
- Podemos incluirlos también manualmente.
- Existen variables que utiliza para una correcta elaboración de la documentación:
  - general: `@author`, `@version`
  - métodos: `@param`, `@return`
- **Javadoc** generará una carpeta doc donde estarán los ficheros en formato html y js. El fichero index.html nos da paso a toda la documentación.



# Pruebas estructurales o de caja blanca

- Vimos las pruebas de camino básico con su complejidad ciclomática,
- Una vez elegidos los valores, las herramientas de depuración nos ayudan a seguir el camino.
- Nos sirven también para revisar el comportamiento de los bucles. Mientras estamos codificando.

→ **Perspectiva Debug** Encontrar defectos en la estructura durante la ejecución.

→ **Crear puntos de ruptura (toggle breakpoint)**

→ **Debug As. Inspeccionar variables según se avanza con**



→ **Opciones de Step para ir instrucción por instrucción (F5) o lo mismo pero saltándose los métodos (F6).**


# Herramientas de depuración

❑ **Depuración:** encontrar defectos durante las pruebas y corregirlos.

❑ **Ayudas de un depurador:**

- Ejecutar código paso a paso.
- Puntos de ruptura.
- Suspender la ejecución.
- Examinar valores de variables a lo largo de la ejecución.

❑ **Para ejecutar en modo depuración en Eclipse:**

- Opción de menú Run → Debug.
- En menú contextual de la clase, opción Debug As → Java Application.
- Icono  y opción de menú Debug As → Java Application.

# Herramientas de depuración

Opción de menú Window → Perspective → Open Perspective → Debug



# Pruebas funcionales o de caja negra

Se deben crear casos de prueba para revisar todos los requisitos funcionales.

## Particiones o clases de equivalencia

1. Determinar las condiciones de entrada del programa.
2. Identificar clases de equivalencia para cada condición de entrada y asignar un número a cada clase. Reglas:
  - Valor específico → 1 clase válida (ese valor) y 2 no válidas.
  - Rango de valores → 1 clase válida (dentro del rango) y 2 no válidas.
  - Conjunto de valores → 1 clase válida por cada valor y 1 no válida.
  - Condición lógica → 1 clase válida y 1 no válida.
  - Si algunos elementos de la clase se tratan de forma distinta, se divide la clase.
3. Crear el número mínimo de casos de prueba con todas las clases válidas.
4. Crear un caso de prueba por cada clase no válida.

# Ejemplo DNI (extendido)

## Particiones o Clases de equivalencia

Condición de entrada	Clases válidas	Clases no válidas
Edad	$18 \leq \text{edad} \leq 65$ (1)	edad < 18 (2) edad > 65 (3) No es un número (4)
NIF	Una cadena de 9 caracteres compuesta por 8 números y una letra (5)	< 9 caracteres (6) > 9 caracteres (7) Alguno de los 8 primeros caracteres no es un número (8) El último carácter no es una letra (9)
Nacionalidad	Española (10)	No española (11)

**Tabla 3.2.** Caso de prueba con clases de equivalencia válidas

Edad	NIF	Nacionalidad	Clases incluidas
35	32323267G	Española	(1), (5), (10)

**Tabla 3.3.** Casos de prueba con clases de equivalencia no válidas

Edad	NIF	Nacionalidad	Clases incluidas
16	78787654Z	Española	(2), (5), (10)
73	88788888U	Española	(3), (5), (10)
AB	56837483Y	Española	(4), (5), (10)
45	879847F	Española	(1), (6), (10)
23	6767676762 <sup>a</sup>	Española	(1), (7), (10)
64	TT789009R	Española	(1), (8), (10)
19	569832349	Española	(1), (9), (10)
23	98828282C	No española	(1), (5), (11)

# Pruebas funcionales o de caja negra

## Análisis de valores límite

- Es una técnica complementaria a la de clases de equivalencia.
- Reglas en relación con las condiciones de entrada:
  1. Rango de valores → 1 caso para cada extremo y 1 justo por debajo del rango y otro por arriba.
  2. Conjunto de valores → 1 caso para cada extremo, otro para 1 menos que el mínimo y otro para 1 más que el máximo.
  3. Emplear la regla 1 para cada condición de salida con un rango de valores.
  4. Emplear la regla 2 para cada condición de salida con un conjunto de valores.
  5. Si la entrada o salida es un conjunto ordenado → centrarse en el primer y último elemento.



# Ejemplo DNI (extendido)

## Análisis de valores límite

**Tabla 3.4.** Clases de equivalencia válidas y no válidas para la condición de entrada edad empleando la técnica de análisis de valores límite

Condición de entrada	Clases válidas	Clases no válidas
Edad	edad = 18 (12) edad = 65 (13)	edad = 17 (14) edad = 66 (15)

**Tabla 3.5.** Casos de prueba con clases de equivalencia válidas que se generan empleando la técnica de análisis de valores límite

Edad	NIF	Nacionalidad	Clases incluidas
18	32323267G	Española	(12), (5), (10)
65	32323267G	Española	(13), (5), (10)

# Pruebas funcionales o de caja negra

## Conjetura de errores

- ❑ Generar una lista típica de errores no relacionados con aspectos funcionales y de las situaciones propensas a estos errores.
- ❑ Ejemplos:
  - ✓ Valor 0 en la entrada y en la salida.
  - ✓ Número variable de valores → no introducir ningún valor, uno solo o todos los valores iguales.
  - ✓ La persona usuaria introduce datos erróneos (tipos de datos).
  - ✓ Puede haber malinterpretaciones de la especificación.

# Estrategia

Prueba de unidad de cada clase → análisis de valores límite, clases de equivalencia y conjetura de errores.

Si no se ha alcanzado la cobertura deseada → pruebas de caja blanca.

Pruebas de integración con técnicas de caja blanca.  
Incrementar el alcance progresivamente.

Prueba del sistema → pruebas de caja negra.

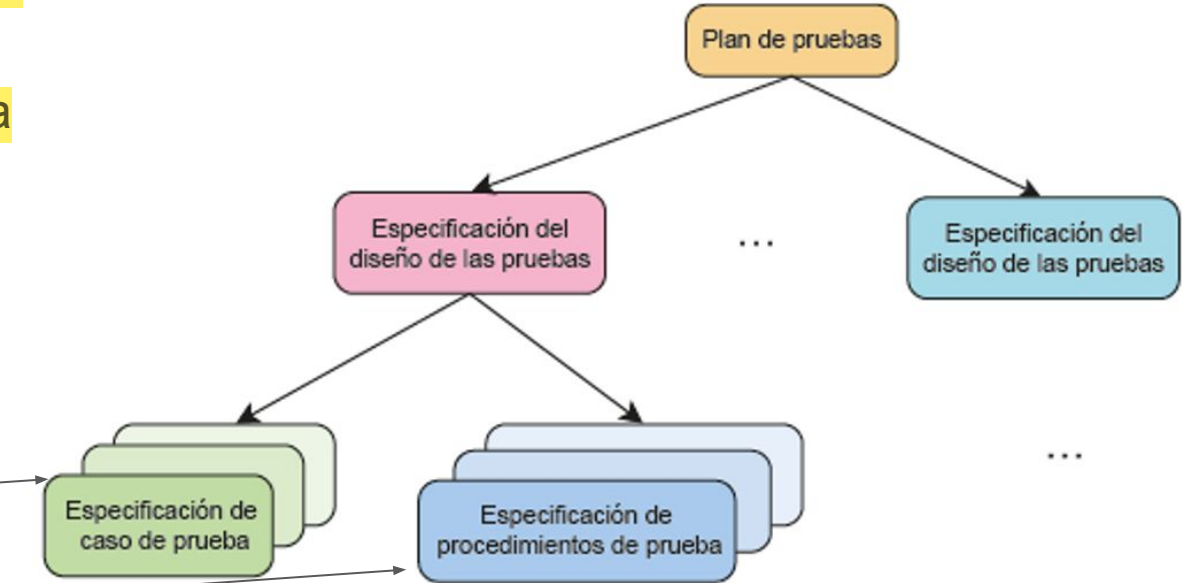
Prueba de validación → pruebas de caja negra del sistema.

# Documentación

¿Cuándo se elabora el plan de pruebas?

¿En qué fase del ciclo de vida?

Qué se va a hacer (casos) y cómo se van a hacer (procedimiento)



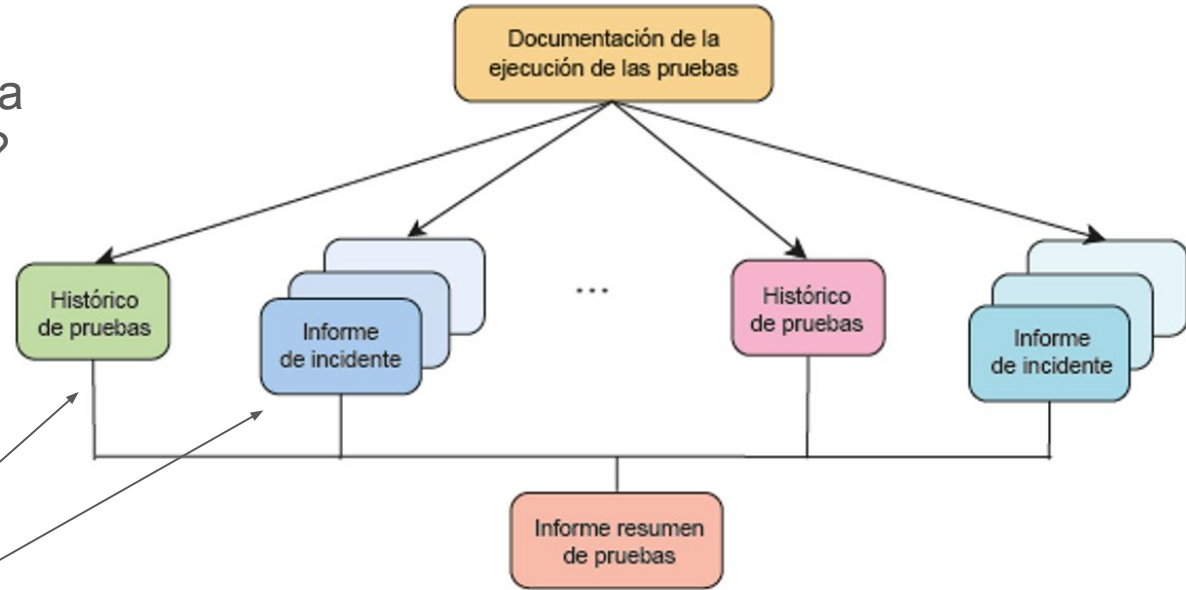
**Figura 3.4.** El plan de pruebas es un documento único para una aplicación con indicaciones generales acerca de las pruebas. Este se desglosa, por cada elemento que se va a probar, en varias especificaciones del diseño de las pruebas. Cada uno de estos documentos contiene además varias especificaciones de casos de prueba y varias especificaciones de procedimientos de pruebas.

# Documentación

¿Cuándo se elabora el plan de pruebas?

¿En qué fase del ciclo de vida?

Qué resultados hemos tenido. Tanto positivos como negativos.

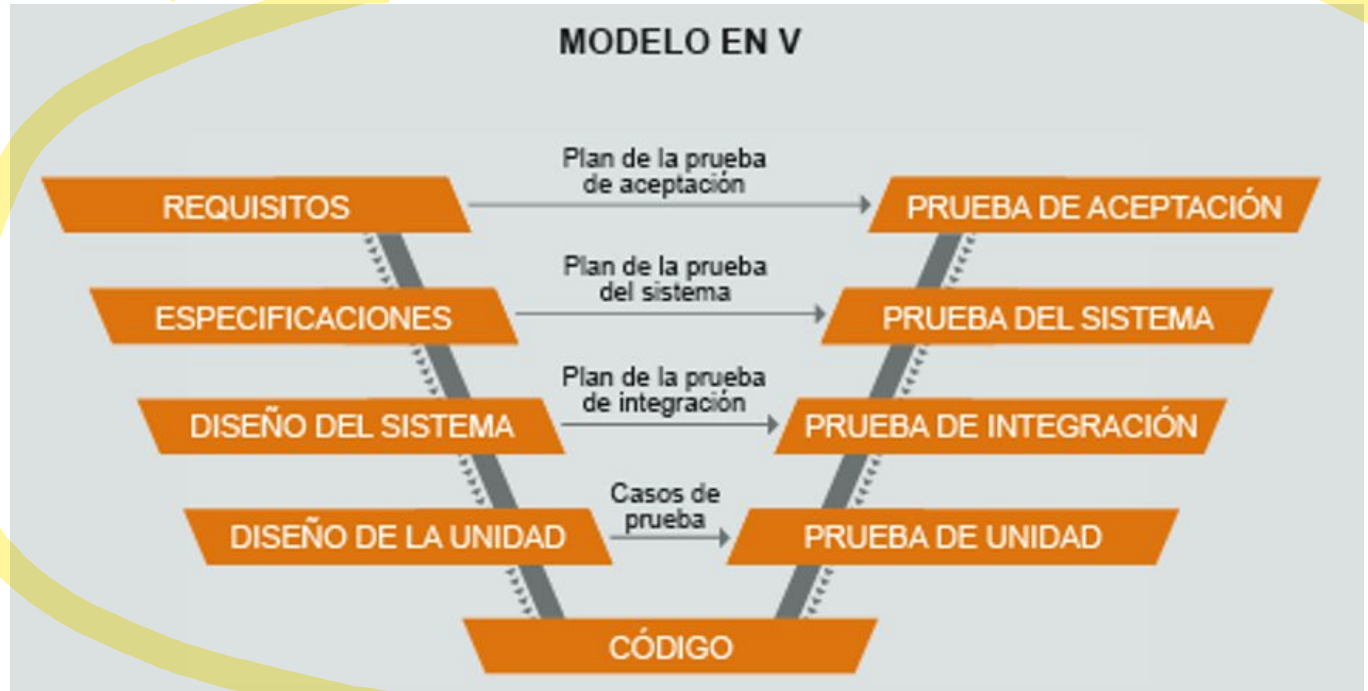


**Figura 3.5.** Se parte de una especificación del diseño de las pruebas y, por cada ejecución de estas, a partir de una serie de casos de pruebas y sus procedimientos, se genera un histórico de pruebas con todos los hechos relevantes ocurridos durante esas pruebas y un informe por cada incidente ocurrido. Todo ello se recoge en el informe resumen de pruebas.

# Documentación

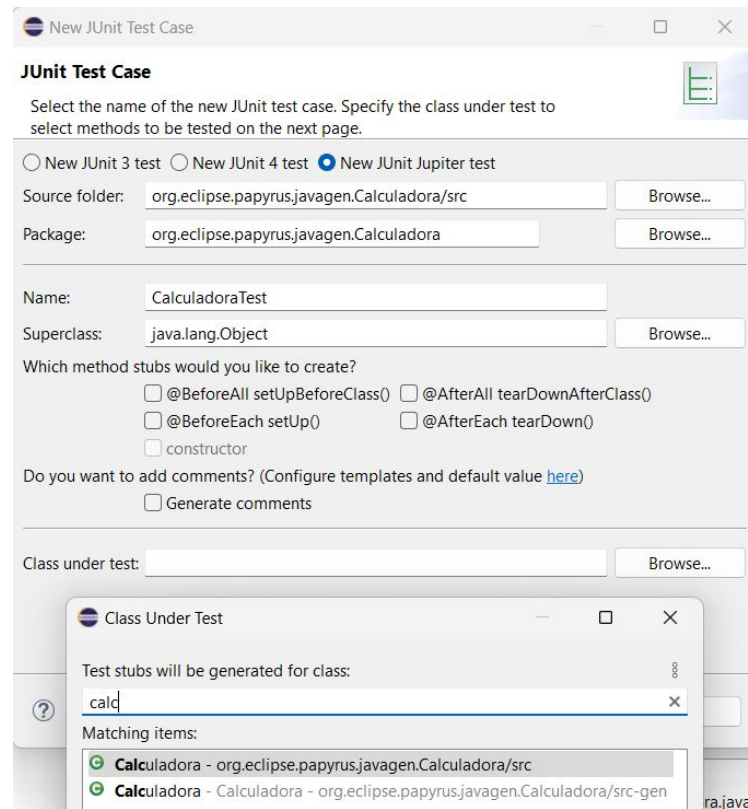
¿Cuándo se ejecutan las pruebas?

¿En qué fase del ciclo de vida?



# Pruebas automáticas - JUnit

- Junit es una herramienta para realizar pruebas unitarias automatizadas.
- Integrada en Eclipse, no es necesario instalar paquetes.
- Pulsando botón derecho en el proyecto, o en el fichero o en la clase; → New → JUnit Test Case
- Pulsamos Next para elegir los métodos: Sumar, Restar, Multiplicar y Dividir.



# Pruebas automáticas - JUnit

- Genera un código para hacer pruebas al que tendremos que añadir los *assertions*
- Revisar la documentación ([Assertions \(JUnit 5.0.1 API\)](#)) para ver el polimorfismo, nosotros vamos a usar

## **assertEquals**

```
public static void assertEquals(int expected,  
                                int actual,
```

- `String message)`

*Asserts* that `expected` and `actual` are equal.



# Pruebas automáticas - JUnit

- Diferencia entre:
  - **Error** con un aspa roja. El error es que no puede hacer la prueba.
  - El **fallo** es que la prueba del código da un resultado no esperado:
    - En los métodos como el mensaje “Error”,
    - Usando fail.

```
void testSumar() {  
    //fail("Not yet implemented");  
    int valoresperado=8;  
    Calculadora c = new Calculadora();  
    int resultado=c.Sumar(5,3);  
  
    assertEquals(valoresperado,resultado,"Error");  
}
```