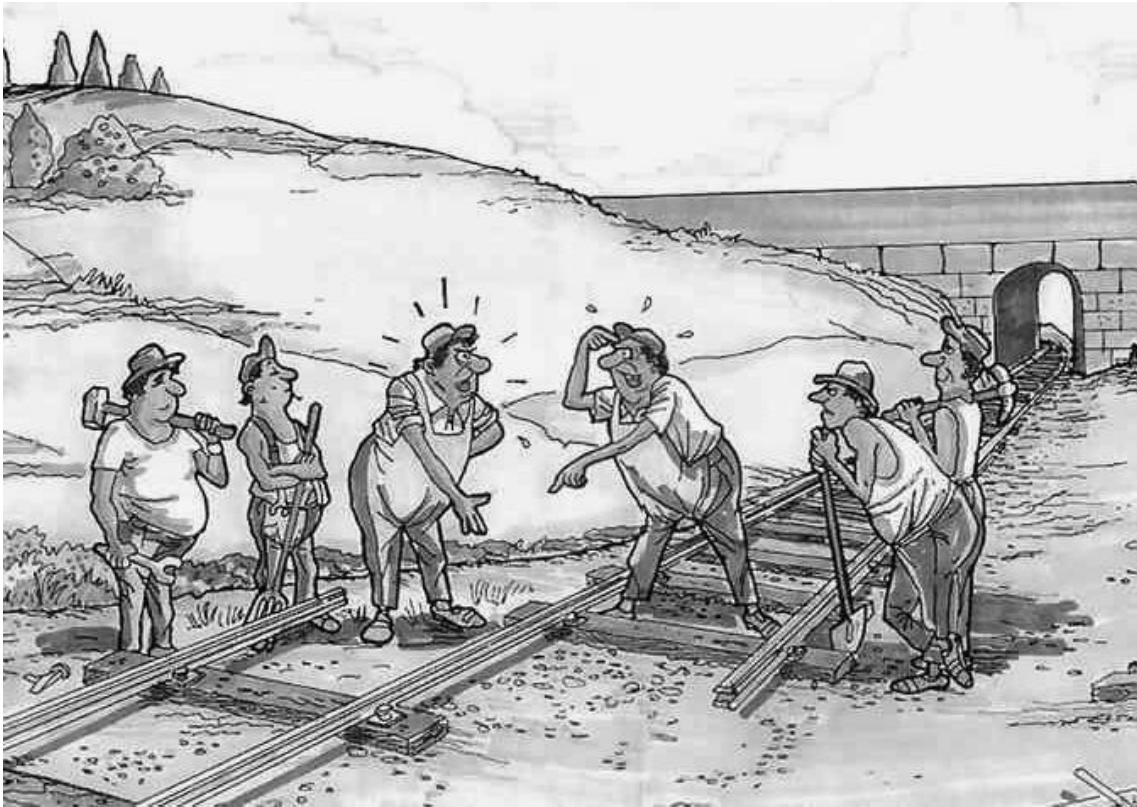


## UNIDAD 5: DISEÑO Y REALIZACIÓN DE PRUEBAS



En este capítulo aprenderemos a utilizar diferentes técnicas para elaborar casos de prueba. Usaremos una herramienta de depuración definiendo puntos de ruptura y examinando variables durante la ejecución de un programa. Aprenderemos a utilizar la herramienta JUNIT para elaborar pruebas unitarias para clases Java.

## Contenido

5.1. INTRODUCCIÓN .....	3
5.2. TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA .....	3
5.2.1. Pruebas de caja blanca .....	4
5.2.2. Pruebas de caja negra .....	5
5.3. ESTRATEGIAS DE PRUEBAS DEL SOFTWARE .....	5
5.3.1. Prueba de unidad .....	6
5.3.2. Prueba de integración .....	7
5.3.3. Prueba de validación .....	8
5.3.4. Prueba del sistema .....	9
5.4. DOCUMENTACIÓN PARA LAS PRUEBAS .....	9
5.5. PRUEBAS DE CÓDIGO .....	10
5.5.1. Prueba del camino básico .....	10
NOTACIÓN DE GRAFO DE FLUJO .....	11
COMPLEJIDAD CICLOMÁTICA .....	13
OBTENCIÓN DE LOS CASOS DE PRUEBA .....	14
5.5.2. Partición o clases de equivalencia .....	14
5.5.4. Herramientas para análisis estático de código .....	17
5.6. HERRAMIENTAS DE DEPURACIÓN .....	17
5.6.1. Puntos de ruptura y seguimiento .....	23
5.6.2. Examinar y modificar variables .....	26
5.7. PRUEBAS UNITARIAS CON JUNIT .....	28
Pruebas parametrizadas .....	40
Suites .....	42

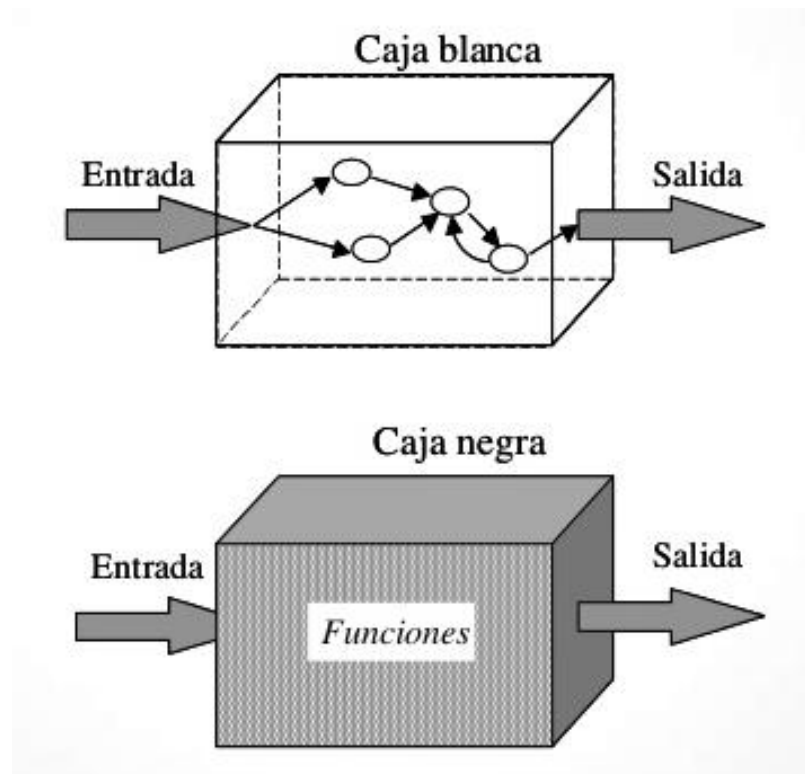
## 5.1. INTRODUCCIÓN

Las pruebas de software consisten en verificar y validar un producto software antes de su puesta en marcha. Constituyen una de las etapas del desarrollo de software, y básicamente consiste en probar la aplicación construida. Se integran dentro de las diferentes fases del ciclo de vida del software dentro de la ingeniería de software.

La ejecución de pruebas de un sistema involucra una serie de etapas: planificación de pruebas, diseño y construcción de los casos de prueba, definición de los procedimientos de prueba, ejecución de las pruebas, registro de resultados obtenidos, registro de errores encontrados, depuración de los errores e informe de los resultados obtenidos.

## 5.2. TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA

Un caso de prueba es un conjunto de entradas, condiciones de ejecución y resultados esperados, desarrollado para conseguir un objetivo particular o condición de prueba. Para llevar a cabo un caso de prueba, es necesario definir las precondiciones y postcondiciones, identificar unos valores de entrada y conocer el comportamiento que debería tener el sistema ante dichos valores. Tras realizar ese análisis e introducir dichos datos en el sistema, se observa si su comportamiento es el previsto o no y por qué. De esta forma se determina si el sistema ha pasado o no la prueba. Para llevar a cabo el diseño de casos de prueba se utilizan dos técnicas o enfoques: prueba de caja blanca y prueba de caja negra (véase figura). Las primeras se centran en validar la estructura interna del programa (necesitan conocer los detalles procedimentales del código) y las segundas se centran en validar los requisitos funcionales sin fijarse en el funcionamiento interno del programa (necesitan saber la funcionalidad que el código ha de proporcionar). Estas pruebas no son excluyentes y se pueden combinar para descubrir diferentes tipos de errores.



#### 5.2.1. Pruebas de caja blanca

También se las conoce como pruebas estructurales o de caja de cristal. Se basan en el minucioso examen de los detalles procedimentales del código de la aplicación. Mediante esta técnica se pueden obtener casos de prueba que:

- Garanticen que se ejecutan al menos una vez todos los caminos independientes de cada módulo.
- Ejecuten todas las sentencias al menos una vez (no hay código muerto).
- Ejecuten todas las decisiones lógicas en su parte verdadera y en su parte falsa.
- Ejecuten todos los bucles en sus límites.
- Utilicen todas las estructuras de datos internas para asegurar su validez.

Una de las técnicas utilizadas para desarrollar los casos de prueba de caja blanca es la prueba del camino básico que se estudiará más adelante.

### 5.2.2. Pruebas de caja negra

Estas pruebas se llevan a cabo sobre la interfaz del software, no hace falta conocer la estructura interna del programa ni su funcionamiento. Se pretende obtener casos de prueba que demuestren que las funciones del software son operativas, es decir, que las salidas que devuelve la aplicación son las esperadas en función de las entradas que se proporcionen.

A este tipo de pruebas también se les llama prueba de comportamiento. El sistema se considera como una caja negra cuyo comportamiento solo se puede determinar estudiando las entradas y las salidas que devuelve en función de las entradas suministradas.

Con este tipo de pruebas se intenta encontrar errores de las siguientes categorías:

- Funcionalidades incorrectas o ausentes.
- Errores de interfaz.
- Errores en estructuras de datos o en accesos a bases de datos externas.
- Errores de rendimiento.
- Errores de inicialización y finalización.

Existen diferentes técnicas para confeccionar los casos de prueba de caja negra, algunos son:

clases de equivalencia, análisis de valores límite, métodos basados en grafos, pruebas de comparación, etc. En este capítulo se estudiarán algunas de estas técnicas.

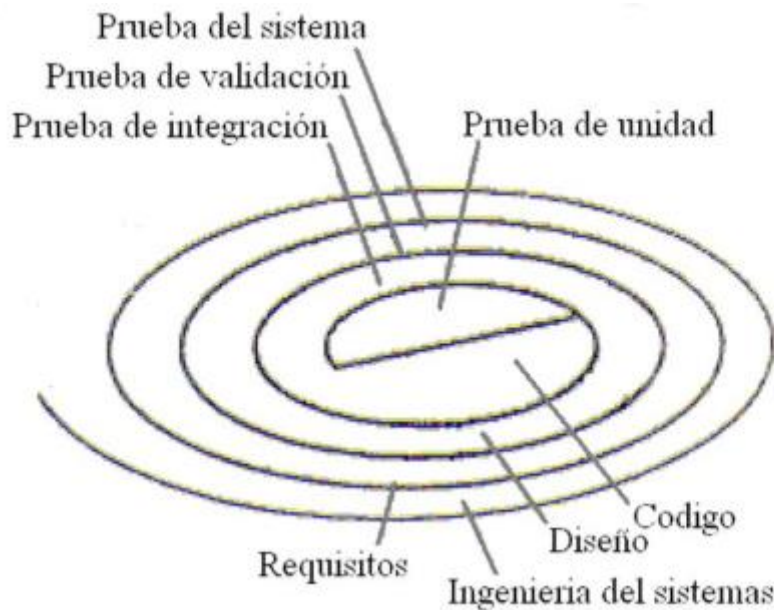
## 5.3. ESTRATEGIAS DE PRUEBAS DEL SOFTWARE

La estrategia de prueba del software se puede ver en el contexto de una espiral.

- En el vértice de la espiral comienza la **prueba de unidad**. Se centra en la unidad más pequeña de software, el módulo tal como está implementado en código fuente.
- La prueba avanza para llegar a la **prueba de integración**. Se toman los módulos probados mediante la prueba de unidad y se construye una estructura de programa que esté de acuerdo con lo que dicta el diseño. El foco de atención es el diseño.
- La espiral avanza llegando a la **prueba de validación** (o de aceptación). Prueba del software en el entorno real de trabajo con intervención del

usuario final. Se validan los requisitos establecidos como parte del análisis de requisitos del software, comparándolos con el sistema que ha sido construido.

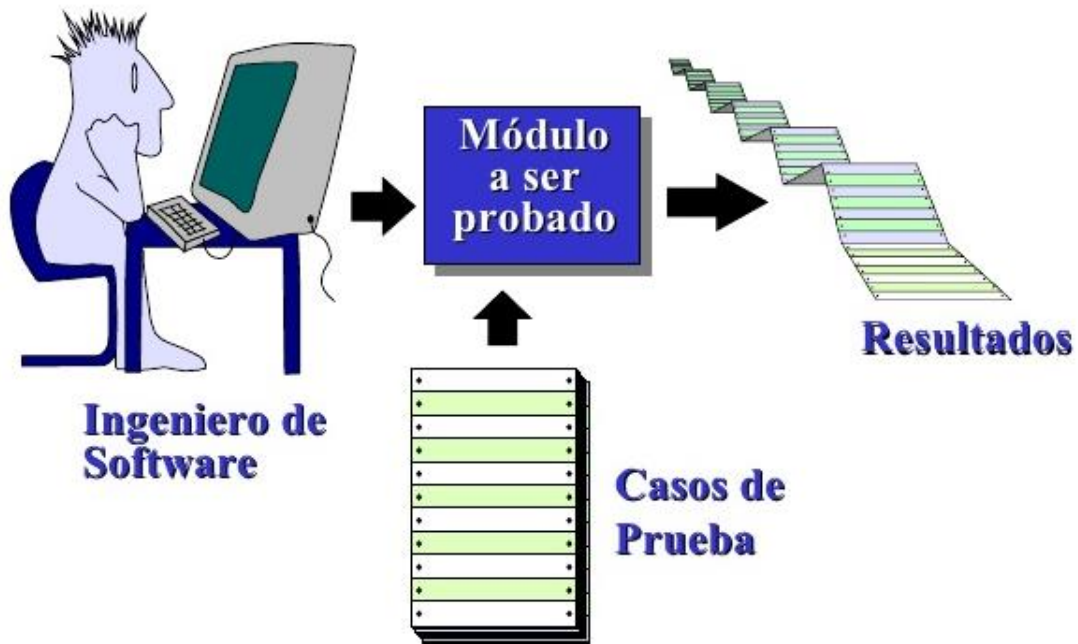
- Finalmente se llega a la **prueba del sistema**. Verifica que cada elemento encaja de forma adecuada y se alcanza la funcionalidad y rendimiento total. Se prueba como un todo el software y otros elementos del sistema.



#### 5.3.1. Prueba de unidad

En este nivel se prueba cada unidad o módulo con el objetivo de eliminar errores en la interfaz y en la lógica interna. Esta actividad utiliza técnicas de caja negra y caja blanca, según convenga para lo que se desea probar. Se realizan pruebas sobre:

- La interfaz del módulo, para asegurar que la información fluye adecuadamente.
- Las estructuras de datos locales, para asegurar que mantienen su integridad durante todos los pasos de ejecución del programa.
- Las condiciones límite, para asegurar que funciona correctamente en los límites establecidos durante el proceso.
- Todos los caminos independientes de la estructura de control, con el fin de asegurar que todas las sentencias se ejecutan al menos una vez.
- Todos los caminos de manejo de errores.



### 5.3.2. Prueba de integración

En este tipo de prueba se observa cómo interaccionan los distintos módulos. Se podría pensar que esta prueba no es necesaria, ya que, si todos los módulos funcionan por separado, también deberían funcionar juntos. Realmente el problema está aquí, en comprobar si funcionan juntos.

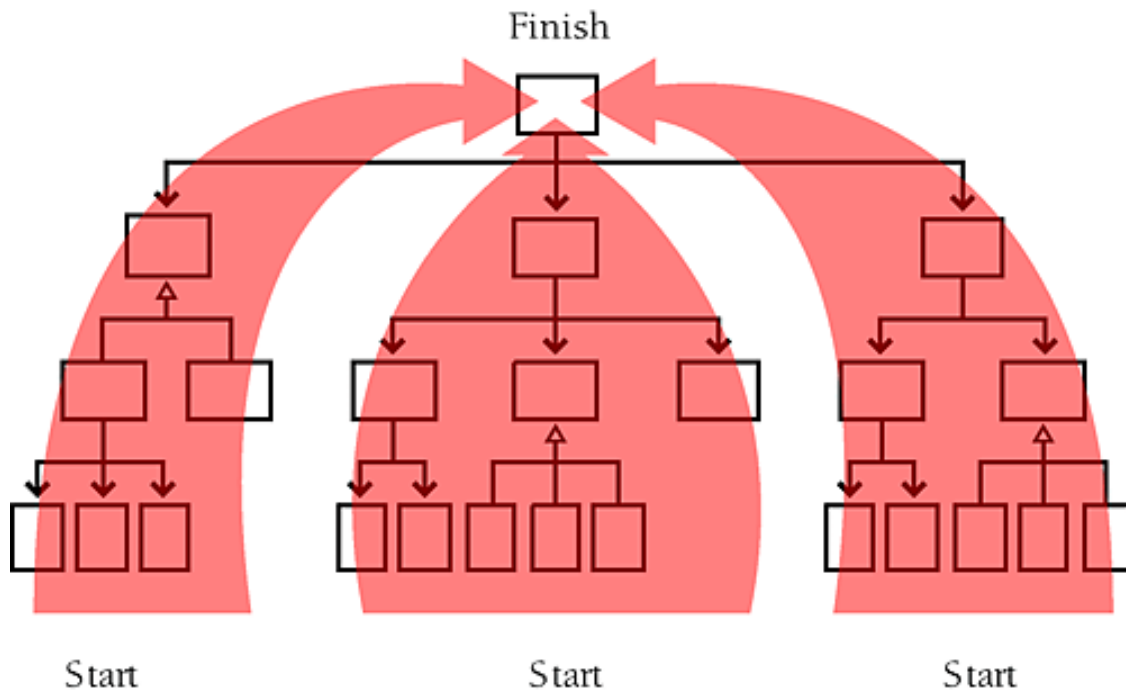
Existen dos enfoques fundamentales para llevar a cabo las pruebas:

- Integración no incremental. Se prueba cada módulo por separado y luego se combinan todos y se prueba todo el programa completo. En este enfoque se encuentran gran cantidad de errores y la corrección se hace difícil.
- Integración incremental. El programa completo se va construyendo y probando en pequeños segmentos, en este caso los errores son más fáciles de localizar. *Se dan dos estrategias: ascendente y descendente.* En la integración ascendente la construcción y prueba del programa empieza desde los módulos de los niveles más bajos de la estructura del programa. En la descendente la integración comienza en el módulo principal (programa principal) moviéndose hacia abajo por la jerarquía de control.

La siguiente figura representa varios módulos y la interconexión entre ellos. El módulo principal es el que está en la raíz, y se muestra una estrategia de



integración ascendente. Se empieza probando los módulos de más bajo nivel en la jerarquía modular del sistema y se procede a probar la integración de abajo hacia arriba hasta llegar al programa principal.

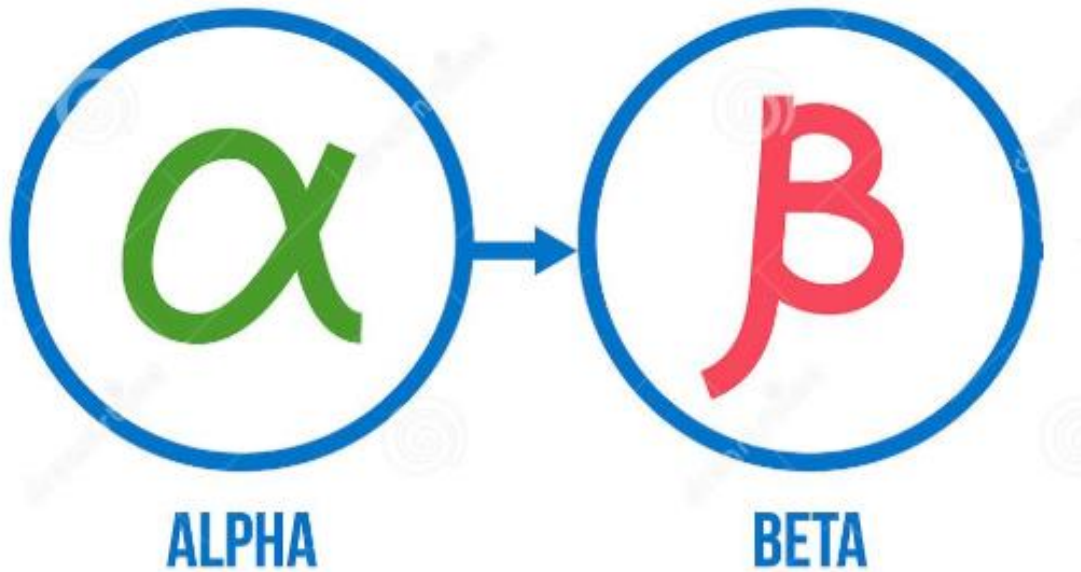


### 5.3.3. Prueba de validación

La validación se consigue cuando el software funciona de acuerdo con las expectativas razonables del cliente definidas en el documento de especificación de requisitos del software o ERS. Se llevan a cabo una serie de pruebas de caja negra que demuestran la conformidad con los requisitos. Las técnicas a utilizar son:

- **Prueba Alfa.** Se lleva a cabo por el cliente o usuario en el lugar de desarrollo. El cliente utiliza el software de forma natural bajo la observación del desarrollador que irá registrando los errores y problemas de uso.
- **Prueba Beta.** Se lleva a cabo por los usuarios finales del software en su lugar de trabajo. El desarrollador no está presente. El usuario registra todos los problemas que encuentra, reales y/o imaginarios, e informa al desarrollador en los intervalos definidos en el plan de prueba. Como resultado de los problemas informados el desarrollador lleva a cabo las modificaciones y prepara una nueva versión del producto.





#### 5.3.4. Prueba del sistema

La prueba del sistema está formada por un conjunto de pruebas cuya misión es ejercitar profundamente el software. Son las siguientes:

- Prueba de recuperación. En este tipo de prueba se fuerza el fallo del software y se verifica que la recuperación se lleva a cabo apropiadamente.
- Prueba de seguridad. Esta prueba intenta verificar que el sistema está protegido contra accesos ilegales.
- Prueba de resistencia (Stress). Trata de enfrentar el sistema con situaciones que demandan gran cantidad de recursos, por ejemplo, diseñando casos de prueba que requieran el máximo de memoria, incrementando la frecuencia de datos de entrada, que den problemas en un sistema operativo virtual, etc.

#### 5.4. DOCUMENTACIÓN PARA LAS PRUEBAS

El estándar IEEE 829-1998 describe el conjunto de documentos que pueden producirse durante el proceso de prueba. Son los siguientes:

- Plan de Pruebas. Describe el alcance, el enfoque, los recursos y el calendario de las actividades de prueba. Identifica los elementos a probar, las características que se van a probar, las tareas que se van a realizar, el personal responsable de cada tarea y los riesgos asociados al plan.

- Especificaciones de prueba. Están cubiertas por tres tipos de documentos: la especificación del diseño de la prueba (se identifican los requisitos, casos de prueba y procedimientos de prueba necesarios para llevar a cabo las pruebas y se especifica la función de los criterios de pasa no-pasa), la especificación de los casos de prueba (documenta los valores reales utilizados para la entrada, junto con los resultados previstos), y la especificación de los procedimientos de prueba (donde se identifican los pasos necesarios para hacer funcionar el sistema y ejecutar los casos de prueba especificados).
- Informes de pruebas. Se definen cuatro tipos de documentos: un informe que identifica los elementos que están siendo probados, un registro de las pruebas (donde se registra lo que ocurre durante la ejecución de la prueba), un informe de incidentes de prueba (describe cualquier evento que se produce durante la ejecución de la prueba que requiere mayor investigación) y un informe resumen de las actividades de prueba.

## 5.5. PRUEBAS DE CÓDIGO

La prueba del código consiste en la ejecución del programa (o parte de él) con el objetivo de encontrar errores. Se parte para su ejecución de un conjunto de entradas y una serie de condiciones de ejecución; se observan y registran los resultados y se comparan con los resultados esperados. Se observará si el comportamiento del programa es el previsto o no y por qué.

Para las pruebas de código se van a mostrar diferentes técnicas que dependerán del tipo de enfoque utilizado: de caja blanca, se centran en la estructura interna del programa; o de caja negra, más centrado en las funciones, entradas y salidas del programa.

### 5.5.1. Prueba del camino básico

La prueba del camino básico es una técnica de prueba de caja blanca que permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución.

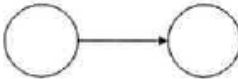
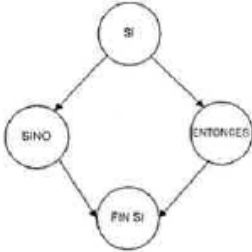
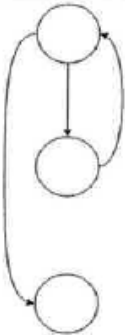
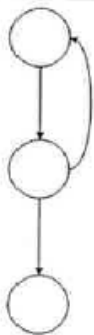
Los casos de prueba obtenidos del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa.

Para la obtención de la medida de la complejidad lógica (o complejidad ciclomática) emplearemos una representación del flujo de control denominada grafo de flujo o grafo del programa.

#### NOTACIÓN DE GRAFO DE FLUJO

El grafo de flujo de las estructuras de control se representa de la siguiente forma.

Se ha de tener en cuenta que cada círculo representa una o más sentencias, sin bifurcaciones, en pseudocódigo o código fuente.

ESTRUCTURA	GRAFO DE FLUJO
<b>SECUENCIAL</b> Instrucción 1 Instrucción 2 ..... Instrucción n	
<b>CONDICIONAL</b> <b>Si</b> <condición> <b>Entonces</b> <Instrucciones> <b>Si no</b> <Instrucciones> <b>Fin si</b>	
<b>HACER MIENTRAS</b> <b>Mientras</b> <condición> <b>Hacer</b> <instrucciones> <b>Fin mientras</b>	
<b>REPETIR HASTA</b> <b>Repetir</b> <instrucciones> <b>Hasta que</b> <condición>	

### CONDICIONAL MÚLTIPLE

**Según sea** <variable> **Hacer**

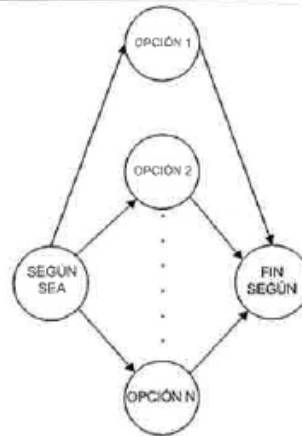
**Caso** opción 1:  
<Instrucciones>

**Caso** opción 2:  
<Instrucciones>

**Caso** opción 3:  
<Instrucciones>

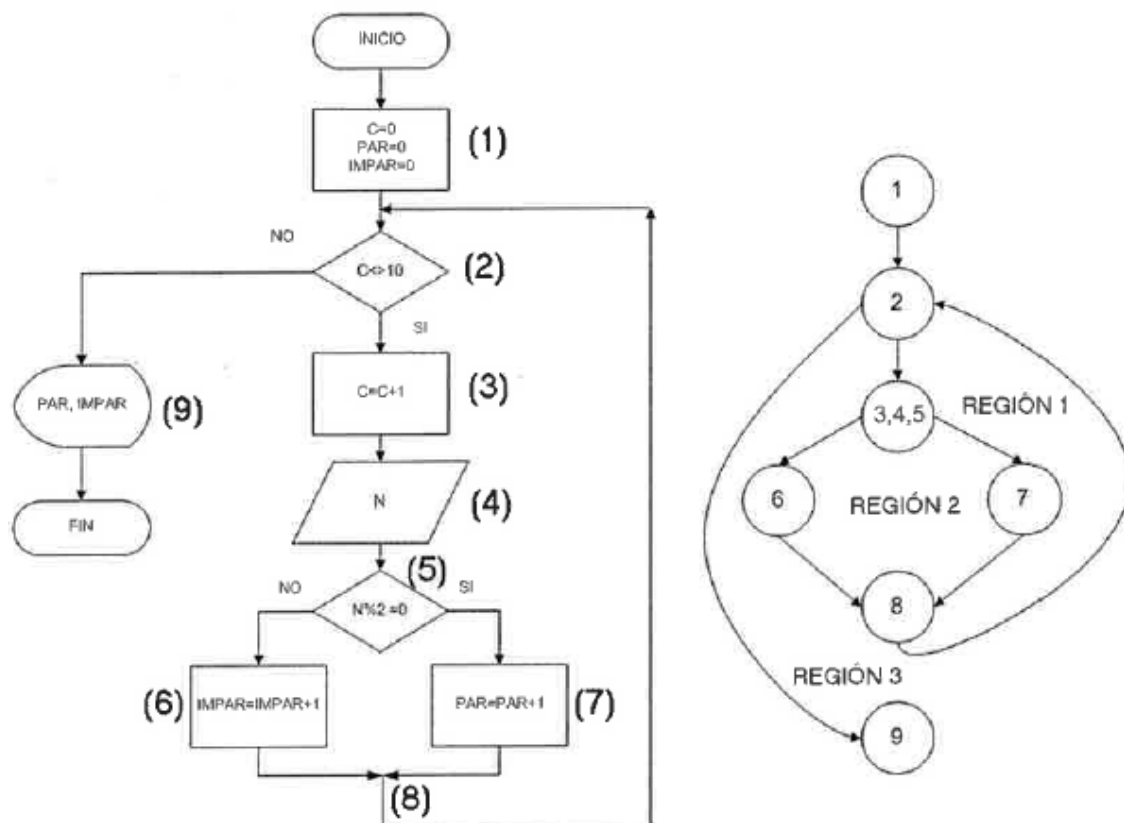
**Otro caso:**  
<Instrucciones>

**Fin según**



Ejemplo: se muestra el diagrama de flujo y el grafo de flujo para un programa que lee 10 números de teclado y muestra cuántos de los números leídos son pares y cuántos son impares.

Para comprobar si el número es par o impar utilizamos el operador % de Java (devuelve el resto de la división) que devuelve 0 si es par. La estructura principal corresponde a un MIENTRAS (o WHILE) y dentro hay una estructura SI (o IF).



Cada círculo del grafo de flujo se llama nodo. Representa una o más sentencias procedimentales. Un solo nodo se puede corresponder con una secuencia de símbolos del proceso y un rombo de decisión. Un ejemplo es el nodo numerado como 3, 4, 5.

Las flechas del grafo de flujo se denominan aristas o enlaces y representan el flujo de control, como en el diagrama de flujo. Una arista termina en un nodo, aunque el nodo no tenga ninguna sentencia procedimental; es el caso del nodo numerado como 8.

El nodo que contiene una condición se llama nodo predicado y se caracteriza porque de él salen dos o más aristas. En el ejemplo se muestran 2 nodos predicado, el representado por el número 2 y el representado por 3, 4 ,5. Únicamente de estos nodos pueden salir dos aristas.

#### COMPLEJIDAD CICLOMÁTICA

La complejidad ciclomática es una métrica del software que proporciona una medida cuantitativa de la complejidad lógica de un programa. En el contexto del método de prueba del camino básico, la complejidad ciclomática establece el número de caminos independientes del conjunto básico de caminos de ejecución de un programa, y por lo tanto, el número de casos de prueba que se deben ejecutar para asegurar que cada sentencia se ejecuta al menos una vez.

La complejidad ciclomática  $V(G)$  se calcula de la siguiente forma:

$$V(G) = \text{Aristas} - \text{Nodos} + 2.$$

Por lo que, para el ejemplo anterior, la complejidad ciclomática sería:

$$V(G) = 8 - 7 + 2 = 3$$

En función de la complejidad ciclomática de un código, se establecen distintos niveles de riesgo:

- Entre 1 y 10: Programas o métodos sencillos, sin mucho riesgo.
- Entre 11 y 20: Programas o métodos más complejos, riesgo moderado
- Entre 21 y 50: Programas o métodos complejos, alto riesgo.
- Mayor que 50: Programas o métodos no testeables, muy alto riesgo.

## OBTENCIÓN DE LOS CASOS DE PRUEBA

El último paso de la prueba del camino básico es construir los casos de prueba que fuerzan la ejecución de cada camino. Con el fin de comprobar cada camino, debemos escoger los casos de prueba de forma que las condiciones de los nodos predicado estén adecuadamente establecidas.

Una forma de representar el conjunto de casos de prueba es como se muestra en la siguiente tabla.

Camino	Caso de prueba	Resultado esperado
1	Escoger algún valor de C tal que NO se cumpla la condición $C < 10$ $C = 10$	Visualizar el número de pares y el de impares
2	Escoger algún valor de C tal que SÍ se cumpla la condición $C < 10$ . Escoger algún valor de N tal que NO se cumpla la condición $N \% 2 = 0$ $C = 1, N = 5$	Contar números impares
3	Escoger algún valor de C tal que SÍ se cumpla la condición $C < 10$ . Escoger algún valor de N tal que SÍ se cumpla la condición $N \% 2 = 0$ $C = 2, N = 4$	Contar números pares

## 5.5.2. Partición o clases de equivalencia

Las clases de equivalencia, es un tipo de prueba de caja negra, en donde cada caso de prueba pretende cubrir el mayor número de entradas posible.

El dominio de valores de entrada se divide en número finito de clases de equivalencia. Como la entrada está dividida en un conjunto de clases de equivalencia, la prueba de un valor representativo de cada clase permite suponer que el resultado que se obtiene con él será el mismo que con cualquier otro valor de la clase.

Cada clase de equivalencia debe cumplir:

- Si un parámetro de entrada debe estar comprendido entre un determinado rango, hay tres clases de equivalencia: por debajo, en y por encima.
- Si una entrada requiere un valor entre los de un conjunto, aparecen dos clases de equivalencia: en el conjunto o fuera de él.

- Si una entrada es booleana, hay dos clases: sí o no.
- Los mismos criterios se aplican a las salidas esperadas: hay que intentar generar resultados en todas y cada una de las clases.

En este ejemplo:

```
1 public boolean numValido (double x)
2 {
3     if( x > 0 && x < 100)
4         return true
5     else
6         return false;
7 }
8
```

Las clases de equivalencia serían:

1. Por debajo:  $x \leq 0$
2. En:  $x > 0$  y  $x < 100$
3. Por encima:  $x \geq 100$

y los respectivos casos de prueba, podrían ser:

1. Por debajo:  $x=0$
2. En:  $x=50$
3. Por encima:  $x=100$

### 5.5.3. Valores límite

En el código Java adjunto, aparecen dos funciones que reciben el parámetro  $x$ . En la función `numVálido1`, el parámetro es de tipo real y en la función `numValido2`, el parámetro es de tipo entero:



```
1 public boolean numValido1 (double x)
2 {
3     if( x > 5)
4         return true
5     else
6         return false;
7 }
8
9 public boolean numValido2 (int x)
10 {
11     if( x > 5)
12         return true
13     else
14         return false;
15 }
16
```

Como se aprecia, el código de las dos funciones es el mismo, sin embargo, los casos de prueba con valores límite va a ser diferente.

La experiencia ha demostrado que los casos de prueba que obtienen una mayor probabilidad de éxito son aquellos que trabajan con valores límite.

Esta técnica, se suele utilizar como complementaria de las particiones equivalentes, pero se diferencia, en que se suelen seleccionar, no un conjunto de valores, sino unos pocos, en el límite del rango de valores aceptado por el componente a probar.

Cuando hay que seleccionar un valor para realizar una prueba, se escoge aquellos que están situados justo en el límite de los valores admitidos.

Por ejemplo, supongamos que queremos probar el resultado de la ejecución de una función, que recibe un parámetro x:

Si el parámetro x de entrada tiene que ser mayor estricto que 5, y el valor es real, los valores límite pueden ser 4,99 y 5,01.

Si el parámetro de entrada x está comprendido entre -4 y +4, suponiendo que son valores enteros, los valores límite serán -5, -4, -3, 3, 4 y 5.

#### 5.5.4. Herramientas para análisis estático de código

Es un tipo de análisis de software que se realiza sin ejecutar el programa. En la mayoría de los casos, el análisis se realiza en alguna versión del código fuente y en otros casos se realiza en el código objeto. El término se aplica generalmente a los análisis realizados por una herramienta automática.

Una de las herramientas más empleadas es SonarQube, con ella podremos recopilar, analizar, y visualizar métricas del código fuente

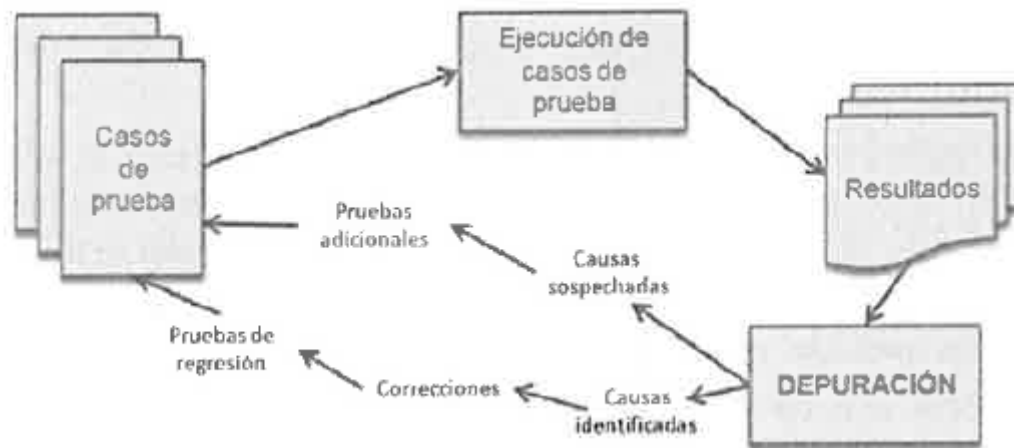
- Informa sobre código duplicado, estándares de codificación, pruebas unitarias, cobertura de código, complejidad ciclomática, errores potenciales, comentarios y diseño de software.
- Aunque pensado para Java, acepta otros lenguajes mediante extensiones.
- Se integra con Maven, Ant, y herramientas de integración continua como Atlassian Bamboo, Jenkins y Hudson.

#### 5.6. HERRAMIENTAS DE DEPURACIÓN

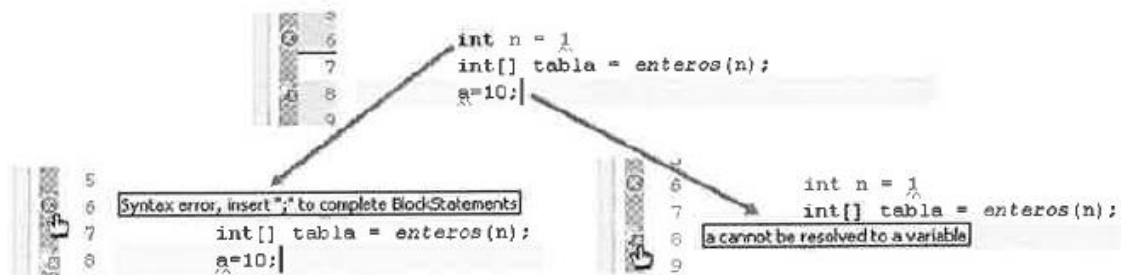
El proceso de depuración comienza con la ejecución de un caso de prueba.

Se evalúan los resultados de la ejecución y fruto de esa evaluación se comprueba que hay una falta de correspondencia entre los resultados esperados y los obtenidos realmente. El proceso de depuración siempre tiene uno de los dos resultados siguientes:

1. Se encuentra la causa del error, se corrige y se elimina.
2. No se encuentra la causa del error. En este caso, la persona encargada de la depuración debe sospechar la causa, diseñar casos de prueba que ayuden a confirmar sus sospechas y volver a repetir las pruebas para identificar los errores y corregirlos (pruebas de regresión, repetición selectiva de pruebas para detectar fallos introducidos durante la modificación).



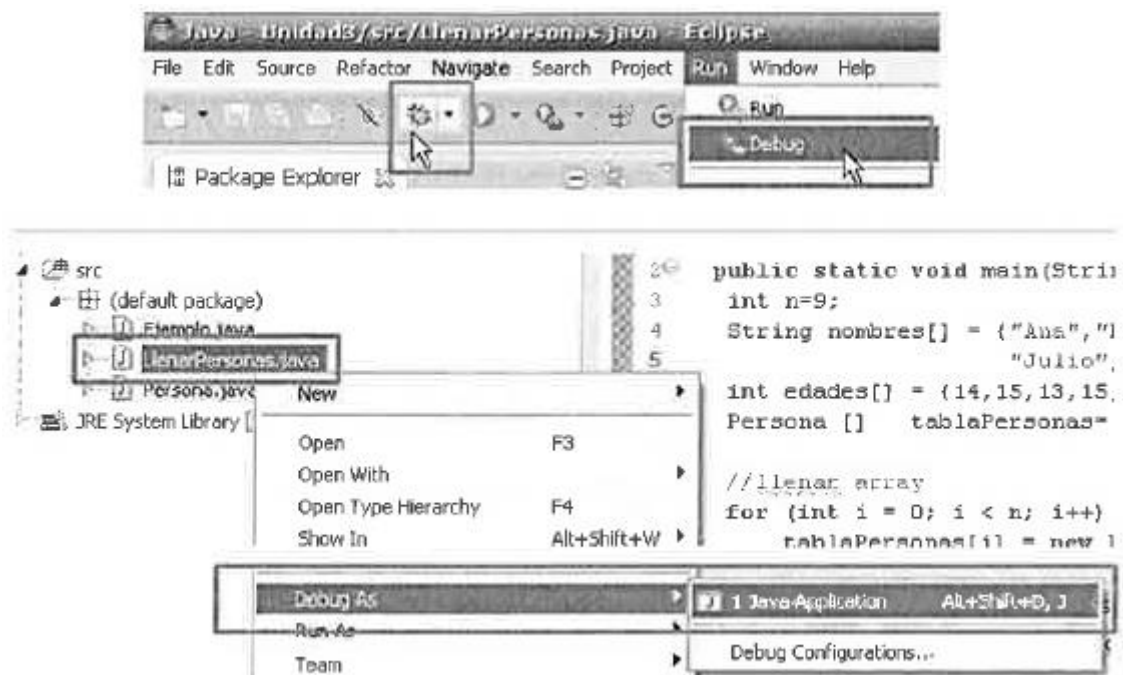
Al desarrollar programas cometemos dos tipos de errores: errores de compilación y errores lógicos. Los primeros son fáciles de corregir ya que normalmente usamos un IDE para codificar y al escribir las sentencias, si por ejemplo nos olvidamos el punto y coma, o usamos una variable inexistente, el entorno proporciona información de la localización del error y cómo solucionarlo.



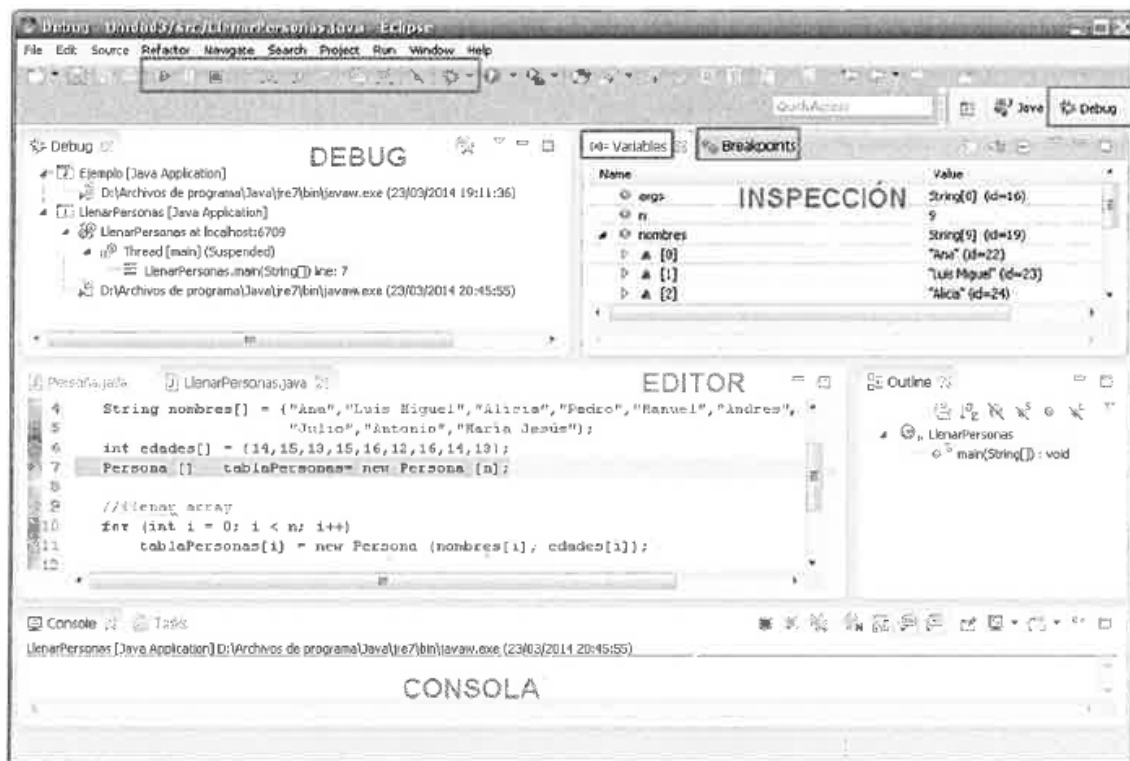
Los errores de tipo lógico son más difíciles de detectar ya que el programa se puede compilar con éxito (es decir, no hay errores sintácticos) y sin embargo, su ejecución puede devolver resultados inesperados o erróneos. A estos errores de tipo lógico se le suele llamar bugs.

Los entornos de desarrollo incorporan una herramienta conocida como depurador (o debugger) para ayudarnos a resolver este tipo de errores. El depurador nos permite analizar el código del programa mientras se ejecuta. Permite establecer puntos de interrupción o de ruptura, suspender la ejecución del programa, ejecutar el código paso a paso y examinar el contenido de las variables.

En Eclipse podemos lanzar el depurador de varias formas: pulsando en el botón Debug, seleccionando el menú Run->Debug o mediante el menú contextual que se muestra al hacer clic con el botón derecho del ratón en la clase que se va a ejecutar y seleccionando Debug As -> Java Application. En cualquiera de esos casos la clase se ejecuta.



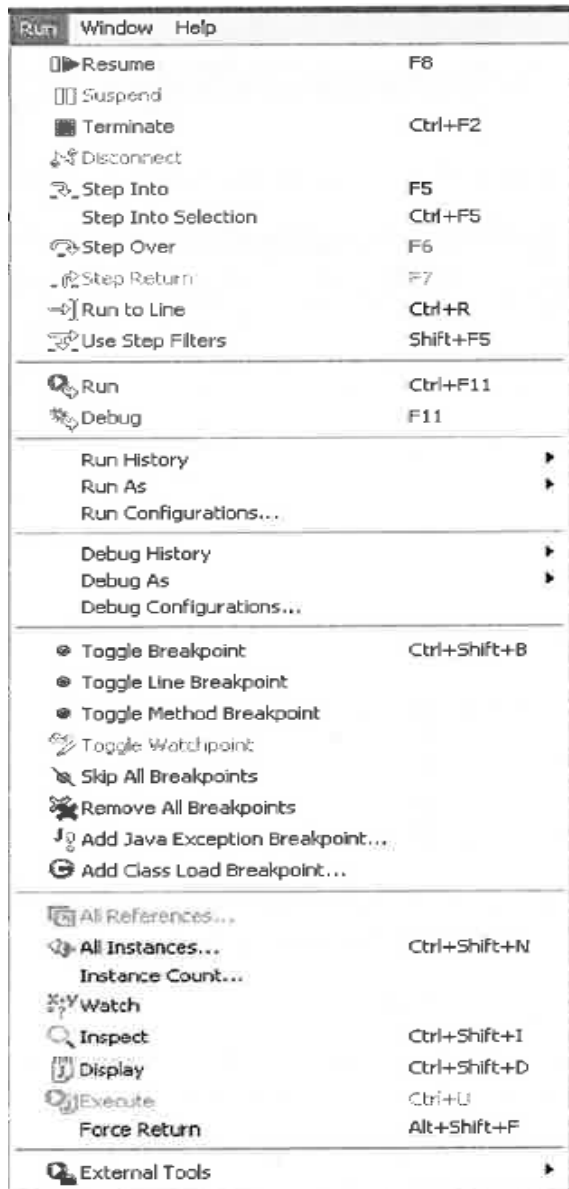
También es conveniente abrir la vista de depuración desde el menú Window -> Open Perspective -> Debug, en la que se muestra la información relativa al programa que se está ejecutando. No obstante, al ejecutar el programa en modo depuración, si la vista no está abierta, se muestra un mensaje desde el que confirmaremos su apertura. Al cambiar a la vista depuración se muestra en la barra de herramientas una serie de botones: para continuar la ejecución, suspenderla, pararla, para meterse dentro de la primera línea de un método, avanzar un paso la ejecución, avanzar el programa hasta salir del método actual, etc.



En esta vista se pueden ver varias zonas:

- En la vista EDITOR se va marcando la traza de ejecución del programa mostrándose una flechita azul en el margen izquierdo de la línea que se está ejecutando.
- La vista DEBUG muestra los hilos de ejecución, en este caso solo muestra un hilo (Thread[main]) y debajo la clase en la que está parada la ejecución mostrando el número de línea.
- Las vistas de INSPECCIÓN permiten ver los valores de las variables y de los puntos de ruptura (o breakpoints) que intervienen en el programa en un instante determinado. Desde aquí se puede modificar el valor de las variables, basta con hacer clic en el valor y cambiarlo; el nuevo valor se usará en los siguientes pasos de ejecución. También desde la pestaña Breakpoints se puede activar o desactivar un breakpoint, eliminarlo, configurarlo para que la ejecución se detenga cuando se pase por él un determinado número de veces, etc.
- Por último, la vista CONSOLA muestra la consola de ejecución del programa que se está depurando. Es la vista sobre la que se redirecciona tanto la entrada como la salida estándar.

Desde el menú Run de la perspectiva de depuración se pueden observar varias opciones, algunas de ellas similares a los botones que aparecen de la barra de herramientas y en la barra de la vista de inspección (véase Figura 3.14):



- **Resume (o tecla F8).** Reanuda un hilo suspendido. Se utiliza cuando no queremos analizar instrucción por instrucción y deseamos que el depurador se pare en la siguiente línea donde hay un breakpoint.
- **Suspend.** Suspende el hilo seleccionado.
- **Terminate (o Ctrl+F2).** Finaliza el proceso de depuración.

- **Step Into (o tecla F5).** Se ejecuta paso a paso cada instrucción. Si el depurador encuentra una llamada a un método o función, al pulsar Step Into se irá a la primera instrucción de dicho método.
- **Step Over (o pulsar F6).** Se ejecuta paso a paso cada instrucción, pero si el depurador encuentra un método, al pulsar Step Over se irá a la siguiente instrucción, sin entrar en el código del método.
- **Step Return (o pulsar F7).** Si nos encontramos dentro de un método, el depurador sale del método actual.
- **Run to Line (o pulsar Ctrl+R).** Se reanuda la ejecución del código a partir de la línea seleccionada.
- **Run (o pulsar Ctrl+R).** Ejecutar el programa.
- **Debug (o pulsar F11).** Ejecutar en modo depuración.
- **Skip All Breakpoints.** Se omiten todos los breakpoints.
- **Remove All Breakpoints.** Elimina todos los puntos de ruptura.
- **Add Java exception Breakpoints.** Permite añadir una excepción Java como breakpoint.
- **All instances.** Abre un cuadro de diálogo emergente que muestra una lista de todas las instancias del tipo Java seleccionado.
- **Watch.** Permite crear nuevas expresiones basadas en las variables del programa.
- **Inspect.** Crea una nueva expresión para la variable seleccionada y la agrega a la vista de inspección.
- **Disptay.** Muestra el resultado de evaluar la expresión seleccionada.



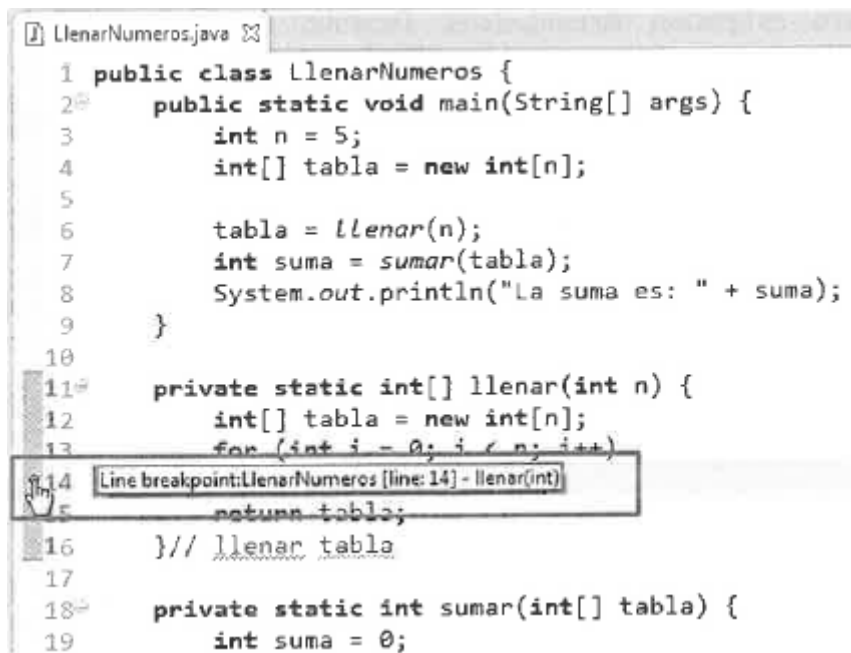
### 5.6.1. Puntos de ruptura y seguimiento

Partimos de la siguiente clase para empezar a utilizar el depurador:

```
1 public class LlenarNumeros {
2
3     public static void main(String[] args) {
4
5         int n=5;
6         int [] tabla = new int [n] ;
7         tabla = llenar(n) ;
8         int suma = sumar(tabla) ;
9         System.out.println("La suma es: " + suma);
10    }
11
12    private static int[] llenar(int n) {
13        int[] tabla = new int[n];
14        for (int i = 0; i<n; i++)
15            tabla[i] = i * 10;
16        return tabla;
17    }
18
19    private static int sumar(int[] tabla) {
20        int suma = 0;
21        int n = tabla.length;
22        for (int i = 0; i<n; i++)
23            suma = suma + tabla[i];
24        return suma;
25    }
26
27 }
```

Donde se definen dos métodos además del método main). El primer método llamado `llenar()` recibe el parámetro entero `n` y devuelve un array con `n` enteros. El segundo método llamado `sumar()`, recibe un array de enteros, suma sus elementos y devuelve la suma.

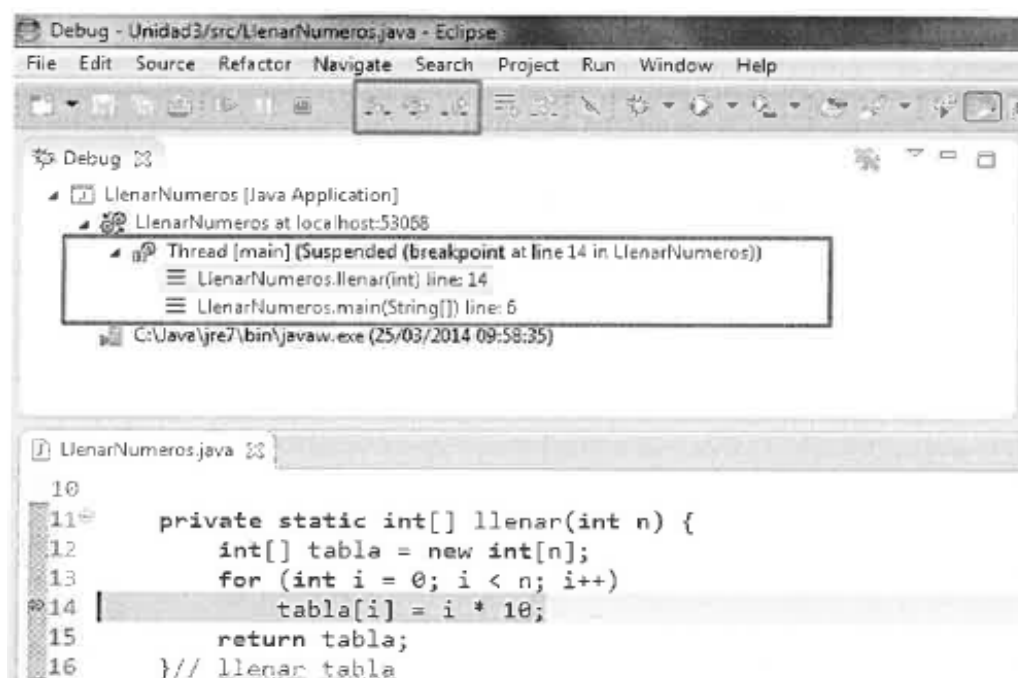
En primer lugar, para empezar el proceso de depuración abrimos la vista de depuración (menú Window -> Open perspective -> Debug). Una vez ahí, para empezar a depurar una clase establecemos un punto de ruptura o breakpoint. Se trata de seleccionar una línea en nuestro código donde queramos que la ejecución se detenga, así podremos ver los valores que tienen las variables en ese momento. Para poner un breakpoint hacemos doble clic en el margen izquierdo del editor, justo en la línea donde queremos que se detenga la ejecución, aparecerá un circulito a la izquierda, tal como se puede ver en la siguiente figura, en la que se establece un punto de ruptura en la línea 14 (`tabla[i] = i * 10`), donde se va llenando el array.



```
1 public class LlenarNumeros {
2     public static void main(String[] args) {
3         int n = 5;
4         int[] tabla = new int[n];
5
6         tabla = llenar(n);
7         int suma = sumar(tabla);
8         System.out.println("La suma es: " + suma);
9     }
10
11     private static int[] llenar(int n) {
12         int[] tabla = new int[n];
13         for (int i = 0; i < n; i++)
14             Line breakpoint: LlenarNumeros [line: 14] - llenar(int)
15             return tabla;
16     } // llenar tabla
17
18     private static int sumar(int[] tabla) {
19         int suma = 0;
```

Una vez establecido el punto de ruptura ejecutamos el programa en modo depuración, para ello pulsamos en el botón Debug. El programa se ejecutará de forma normal hasta que la ejecución llegue al punto de ruptura establecido, en ese momento se detendrá.

En la ventana Debug aparece la pila de llamadas, como se puede apreciar en la siguiente figura, donde se ven cada uno de los hilos de ejecución (en este caso solo hay uno, Thread main[]). Debajo de esta línea se muestra la clase con el método donde está ahora la ejecución parada: la clase LlenarNumeros, método llenar). Se muestra también la línea donde está parada la ejecución, la 14.



```
10
11     private static int[] llenar(int n) {
12         int[] tabla = new int[n];
13         for (int i = 0; i < n; i++)
14             tabla[i] = i * 10;
15         return tabla;
16     } // llenar tabla
```

La siguiente línea muestra quién ha llamado a este método, en este caso la llamada se realiza desde la clase LlenarNumeros y dentro del método main() en la línea 6, en esta línea está la sentencia `tabla = llenar(n)`. Al hacer clic en estas líneas se muestra la línea de código que se está ejecutando.

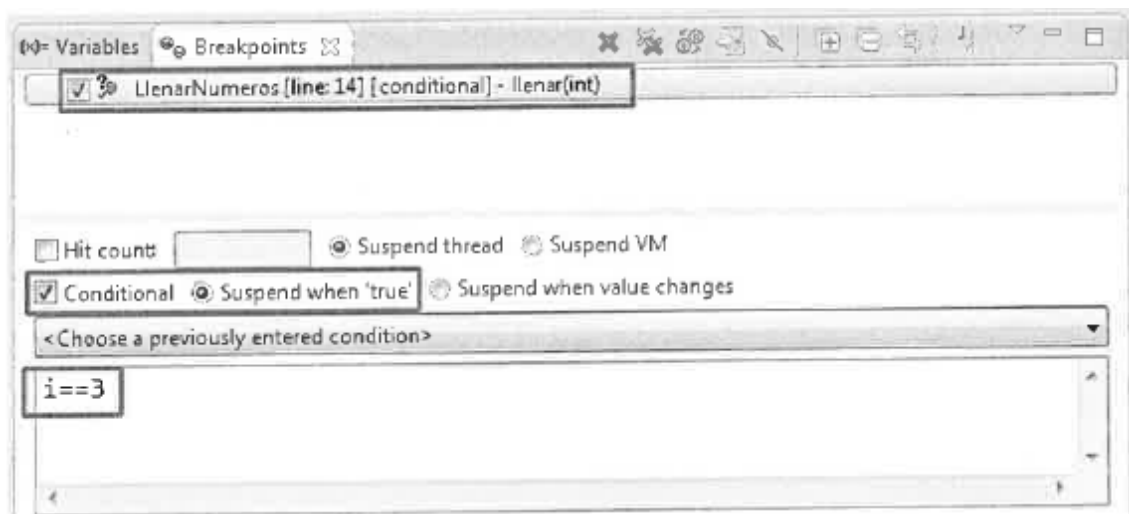
A continuación, podemos usar los botones:

- Step Into: para ejecutar el programa paso a paso instrucción por instrucción.
- Step Over: para ejecutar paso a paso cada instrucción, pero si encuentra un método saltarlo.
- Step Return: este caso si nos encontramos dentro de un método, al pulsarlo el depurador hace avanzar la ejecución del programa saliendo del método.

En cualquier momento podemos finalizar el depurador pulsando en el botón Terminate.

Una flechita al lado de la línea, nos indica la línea que se está ejecutando. Para quitar el punto de ruptura de alguna línea hacemos doble clic sobre el circulito.

Podemos establecer puntos de ruptura condicionales, por ejemplo, con el punto de ruptura establecido en la línea 14 podemos determinar que la ejecución se detenga cuando el valor de la `i` sea 3, tal como se puede ver en la siguiente imagen.

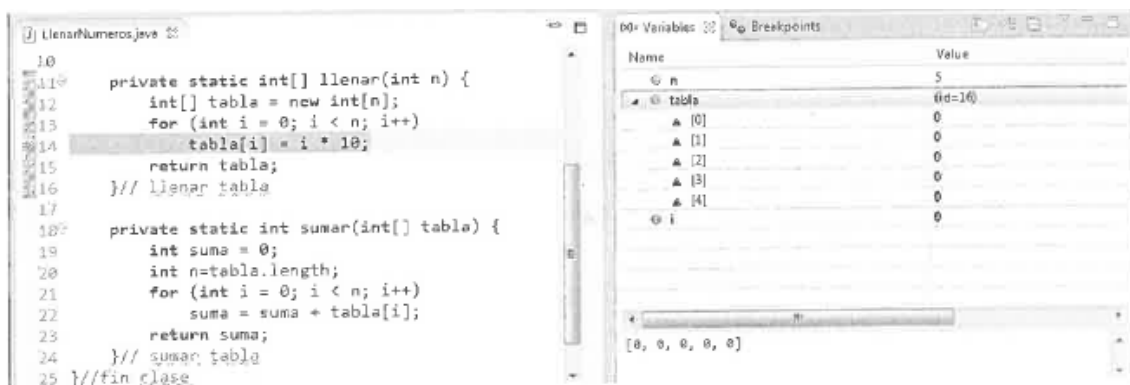


Desde la vista de INSPECCIÓN y desde la pestaña Breakpoints marcamos la casilla Conditional, seleccionamos Suspend when 'true' y escribimos la condición a evaluar (en este caso que la `i` sea igual a 3, `i==3`) que debe devolver un valor

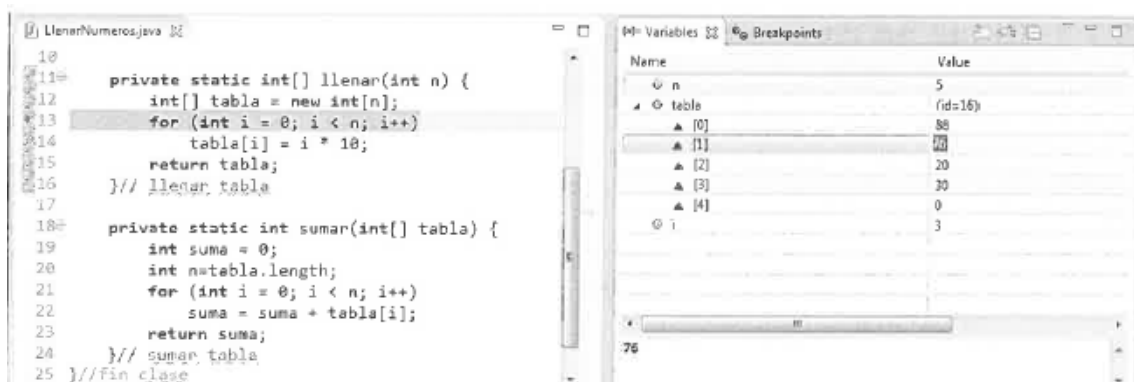
booleano. Entonces cuando la *i* sea 3 se detiene la ejecución y podemos empezar a usar los botones para ejecutar instrucción por instrucción.

### 5.6.2. Examinar y modificar variables

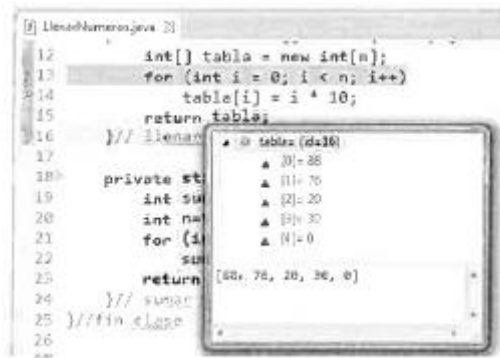
Desde la vista de INSPECCIÓN y desde la pestaña de Variables podemos inspeccionar las variables definidas en el punto en el que el programa está detenido en este momento tal como se puede observar en la siguiente imagen. En este punto se muestra el valor de la variable *n*, los elementos de la tabla inicializados a 0 y el valor inicial de la *i* que es 0. En la parte inferior se muestra el valor de la variable seleccionada, en este caso se muestran los valores de la tabla [0, 0, 0, 0, 0].



Desde aquí se puede modificar el valor de una variable tan solo haciendo clic sobre ella y escribiendo el valor deseado.



Otra forma de ver el contenido de la variable es pasando el puntero del ratón por ella, se abre una ventanita mostrándonos la información.



Como resumen podemos decir que el depurador nos permite ejecutar un programa de forma controlada con el fin de probarlo, encontrar la causa de un error o incluso conocer mejor su funcionamiento.

Proporciona las siguientes funciones:

- Ejecutar el programa paso a paso. Tras cada paso, el usuario recupera el control.
- Detener la ejecución del programa cuando alcance una determinada línea del código o cuando se cumpla una condición.
- A cada paso de ejecución se puede conocer el valor de las variables o expresiones.

## 5.7. PRUEBAS UNITARIAS CON JUNIT

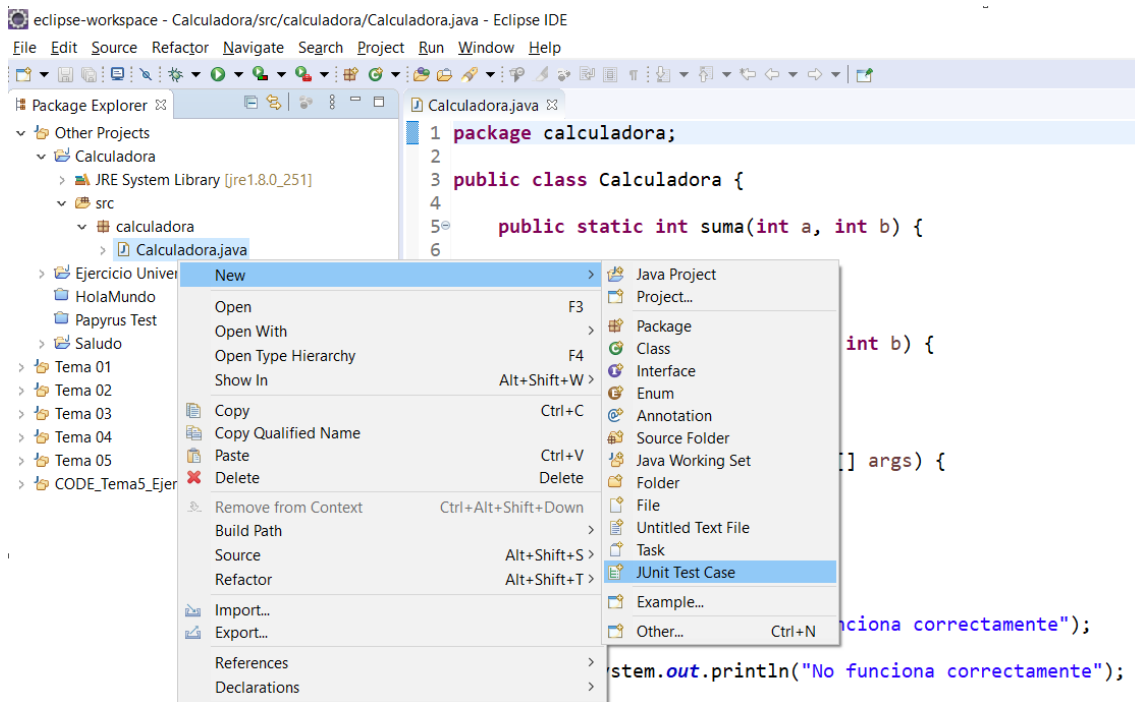
Hasta el momento, la realización de las pruebas la llevábamos a cabo de forma manual. Sin embargo, este método presenta varios problemas, veámoslo con un ejemplo:

```
1 package calculadora;
2
3 public class Calculadora {
4
5     public static int suma(int a, int b) {
6
7         return a + b;
8     }
9
10    public static int resta(int a, int b) {
11
12        return a - b;
13    }
14
15    public static void main(String[] args) {
16
17        int a = 3, b = 2;
18        int res = suma(a, b);
19        int esperado = 5;
20
21        if (res == esperado)
22            System.out.println("Funciona correctamente");
23        else
24            System.out.println("No funciona correctamente");
25    }
26 }
```

- El código de las pruebas va a molestar en el código principal del programa y en muchas ocasiones nos veremos forzados a moverlo o incluso eliminarlo.
- Puede suceder que, durante la propia realización de las pruebas, al ser éstas manuales, cometamos algún error en la comprobación de los resultados sin ser conscientes de ello.

Por este motivo existen herramientas como JUnit, que nos permiten realizar pruebas unitarias de manera automatizada. Además, Junit se encuentra integrado en la mayoría de IDE (Eclipse, NetBeans, IntelliJ, etc.), por lo que no tendremos que descargar ningún paquete adicional para utilizarlo.

Vamos a ver como se crearía una prueba unitaria con Junit. Para ello, en el explorador de paquetes, Con la clase Calculadora seleccionada pulsamos el botón derecho del ratón y seleccionamos New->JUnit Test Case:



Se abrirá una ventana de diálogo, desde aquí debemos seleccionar New JUnit 4 test y cubrir el campo Package para que las pruebas se creen en un paquete de pruebas distinto del paquete en el que tenemos nuestro código.

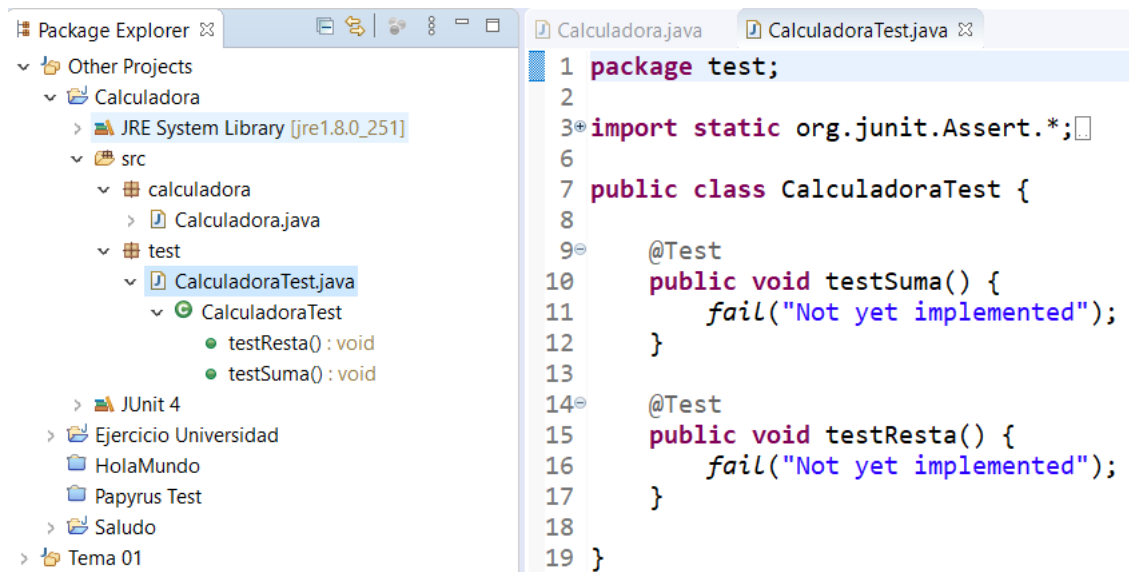
Nota: Antes de continuar, a través de los siguientes enlaces puedes consultar las principales características de Junit 5, basado en muchas de las novedades de Java 8, como pueden ser las lambdas:

<https://www.adictosaltrabajo.com/2016/11/24/primeros-pasos-con-junit-5/>

<https://howtodoinjava.com/junit5/junit-5-vs-junit-4/>

Para el resto de las opciones dejamos los valores por defecto, como nombre de clase se generará el nombre CalculadoraTest. Pulsamos el botón Next. A continuación, hemos de seleccionar los métodos que queremos probar, marcamos los 2 métodos y pulsamos Finish. Se abre una ventanita indicándonos que la librería JUnit 4 no está incluida en nuestro proyecto y pulsamos el botón OK para que se incluya.

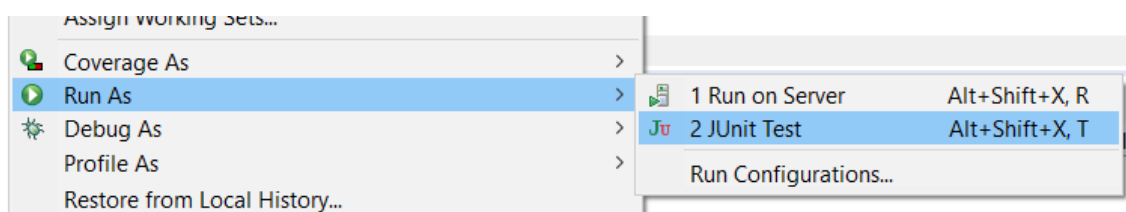




La clase de prueba se crea automáticamente, se observan una serie de características:

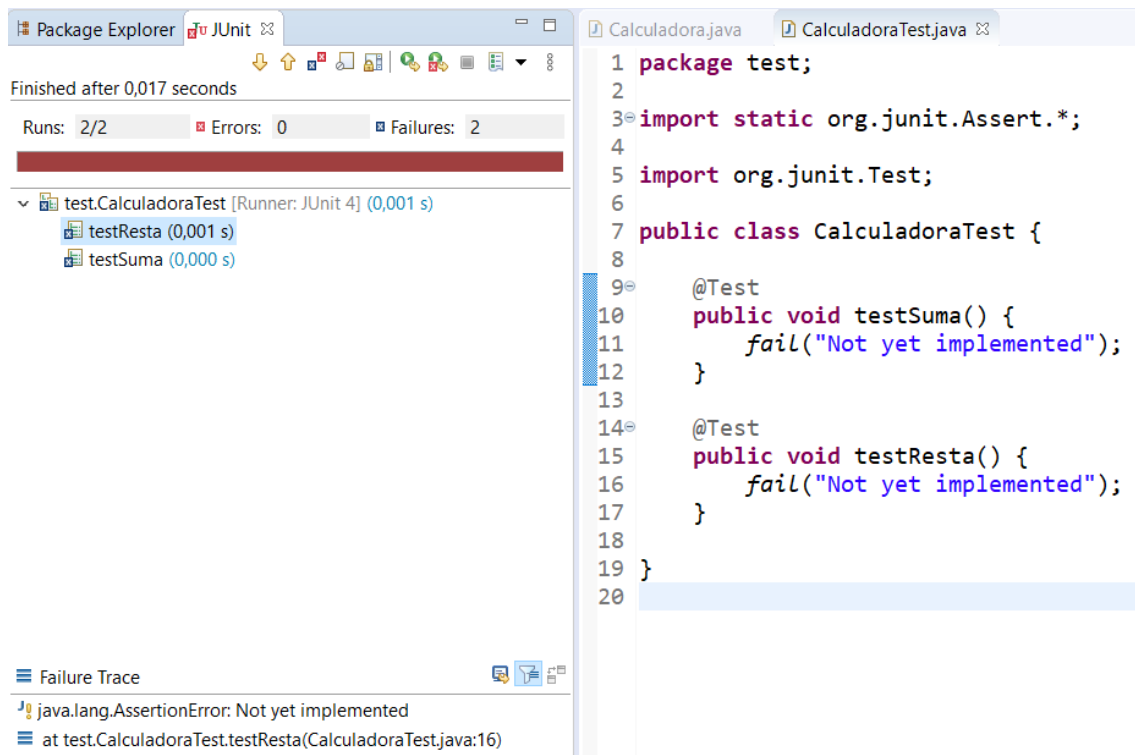
- Se crean 2 métodos de prueba, uno para cada método seleccionado anteriormente.
- Los métodos son públicos, no devuelven nada y no reciben ningún argumento.
- El nombre de cada método incluye la palabra test al principio testSuma(), testResta().
- Sobre cada uno de los métodos aparece la anotación @Test que indica al compilador que es un método de prueba.
- Cada uno de los métodos de prueba tiene una llamada al método fail() con un mensaje indicando que todavía no se ha implementado el método. Este método hace que el test termine con fallo lanzando el mensaje.

Si ejecutamos nuestra clase CalculadoraTest.java como un JUnit Test:



Se nos abrirá en Eclipse una pestaña Junit, en la que se nos mostrará el resultado de nuestro test. En este caso, obtenemos dos fallos, uno por cada método. Esto es debido a que cada método tiene una llamada al método fail que por defecto implementa Junit cuando creamos un test, y que provoca

automáticamente un fallo. Se puede comprobar que los errores se encuentran en las líneas de código en las que están situados dichos métodos.



Llegados a este punto, antes de implementar los métodos de prueba `testSuma` y `testResta`, vamos a ver una serie de métodos de Junit.

En primer lugar, vamos a ver la función de los Inicializadores y Finalizadores. El método `SetUp` y el método `tearDown`, se utilizan para inicializar y finalizar las condiciones de prueba, como puede ser la creación de un objeto, inicialización de variables, etc. En algunos casos, no es necesario utilizar estos métodos, pero siempre se suelen incluir.

El **método `setUp`** es un método de inicialización de la prueba y se ejecutan antes de cada caso de prueba, en la clase de prueba. Este método no es necesario para ejecutar pruebas, pero si es necesario para inicializar algunas variables antes de iniciar la prueba.

El **método `tearDown`** es un método finalizador de prueba, y se ejecutará después de cada test en la clase prueba. Un método finalizador no es necesario para ejecutar las pruebas, pero si necesitamos un finalizador para limpiar algún dato que fue requerido en la ejecución de los casos de prueba.

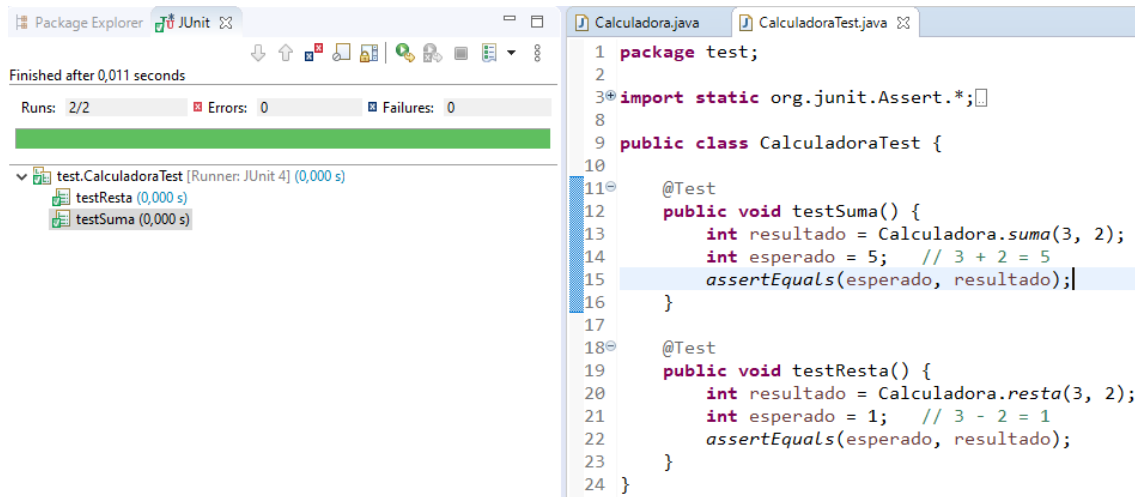
En segundo lugar, es necesario conocer las aserciones. Los métodos `assertXXX()`, se utilizan para hacer las pruebas. Estos métodos, permiten comprobar si la salida del método que se está probando, concuerda con los valores esperados. Las principales son:

- **`AssertTrue()`** evalúa una expresión booleana. La prueba pasa si el valor de la expresión es true.
- **`assertFalse()`** evalúa una expresión booleana. La prueba pasa si el valor de la expresión es false.
- **`AssertNull()`** verifica que la referencia a un objeto es nula.
- **`assertNotNull()`** verifica que la referencia a un objeto es no nula.
- **`AssertSame()`** compara dos referencias y asegura que los objetos referenciados tienen la misma dirección de memoria. La prueba pasa si los dos argumentos son el mismo objeto o pertenecen al mismo objeto.
- **`assertNotSame()`** Compara dos referencias a objetos y asegura que ambas apuntan a diferentes direcciones de memoria. La prueba pasa si los dos argumentos supuestos son objetos diferentes o pertenecen a objetos distintos.
- **`assertEquals()`** Se usa para comprobar igualdad a nivel de contenidos. La igual de tipos primitivos se compara usando "=", la igual entre objetos se compara con el método `equals()`. La prueba pasa si los valores de los argumentos son iguales.
- **`fails()`** causa que la prueba falle inmediatamente. Se puede usar cuando la prueba indica un error o cuando se espera que el método que se está probando llame a una excepción.

Nota: Si quieres conocer más información de estos métodos, así como de las anotaciones `@Before`, `@After`, `@Test`, etc., consulta el siguiente enlace:

<http://junit.sourceforge.net/javadoc/org/junit/package-summary.html>

Pasamos a implementar los métodos de prueba `testSuma` y `testResta`. Para ello usaremos `assertEquals(long expected, long actual)` ya que el casteo de `int` a `long` se puede realizar sin problemas.



Como vemos en la imagen anterior, en la que se puede observar que ya se ha ejecutado el test, el resultado es correcto.

Ejercicio: Intercambia las líneas “return a + b” y “return a - b” de la clase calculadora y ejecuta de nuevo el test. Comprobarás que Junit detecta 2 errores.

Nota: Es muy útil volver a ejecutar pruebas ya superadas cuando realizamos cambios parciales sobre nuestro código, ya que nos pueden servir para darnos cuenta de que algo se ha roto a partir de algún cambio determinado y probablemente nos interese volver a una versión anterior de nuestro código.

Ahora nos vamos a fijar en los posibles iconos que se muestran en la pestaña Junit, al lado de cada prueba que ejecutamos:



Indica prueba exitosa.



Indica fallo. Por ejemplo, no se cumple una aserción.



Indica error. Causa una excepción, por ejemplo, un null pointer.

Ejercicio: Implementa en la clase calculadora un método multiplicación y otro método división. Realiza un test para cada uno de estos métodos, de manera que obtengas un fallo en la multiplicación y un error en la división.

A continuación, vamos a ver las anotaciones @Before y @After.

Para ello creamos una nueva clase CalculadoraV2:

```
1 package calculadora;
2
3 public class CalculadoraV2 {
4
5     private int ans;
6
7     public CalculadoraV2() {
8         ans = 0;
9     }
10
11     public int suma(int a, int b) {
12         ans = a + b;
13         return ans;
14     }
15
16     public int resta(int a, int b) {
17         ans = a - b;
18         return ans;
19     }
20
21     public int suma(int a) {
22         ans += a;
23         return ans;
24     }
25
26     public int resta(int a) {
27         ans -= a;
28         return ans;
29     }
30
31     public int ans() {
32         return ans;
33     }
34
35     public void clear() {
36         ans = 0;
37     }
38 }
```

Para la que crearemos un test con Junit, dentro del paquete test.

Como resultado obtenemos el siguiente código:

```
1 package test;
2
3+ import static org.junit.Assert.*;
8
9 public class CalculadoraV2Test {
10
11-   @Test
12   public void testSuma() {
13       CalculadoraV2 calcV2 = new CalculadoraV2();
14       int resultado = calcV2.suma(3, 2);
15       int esperado = 5;
16       assertEquals(esperado, resultado);
17   }
18
19-   @Test
20   public void testAnsSuma() {
21       CalculadoraV2 calcV2 = new CalculadoraV2();
22       calcV2.suma(3, 2);
23       int resultado = calcV2.ans();
24       int esperado = 5;
25       assertEquals(esperado, resultado);
26   }
27
29+   public void testResta() {
36
38+   public void testAnsResta() {
45 }
```

Ejercicio: Termina la implementación de la clase `CalculadoraV2Test`.

Como podemos observar, hay código que estamos repitiendo en las distintas pruebas, para solucionar estos problemas existen las anotaciones `@Before` y `@After`:

- Los métodos anotados con `@Before`, se ejecutan antes de cada test.
- Los métodos anotados con `@After`, se ejecutan después de cada test.

Vamos a crear una nueva clase `CalculadoraV2Test2` en la que emplearemos estas anotaciones para dejar un código más limpio.

Ejercicio: En la clase CalculadoraV2Test2, añade un `System.out.println()` con un mensaje significativo como primera línea de cada método. Ejecuta la clase de prueba y comprueba si los mensajes que muestra la consola se corresponden con el funcionamiento esperado del código, como consecuencia de usar las anotaciones `@Before` y `@After`.

La clase CalculadoraV2TestV2 quedaría como sigue:

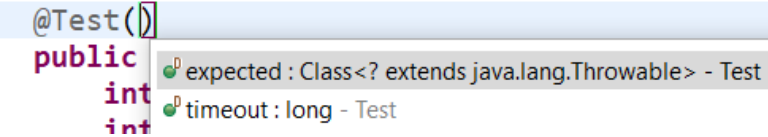
```
1 package test;
2
3+ import static org.junit.Assert.*;
10
11 public class CalculadoraV2TestV2 {
12
13     CalculadoraV2 calcV2;
14
15     @Before
16     public void before() {
17         System.out.println("before()");
18         calcV2 = new CalculadoraV2();
19     }
20
21     @After
22     public void after() {
23         System.out.println("after()");
24         calcV2.clear();
25     }
26
27     @Test
28     public void testSuma() {
29         System.out.println("testSuma()");
30         int resultado = calcV2.suma(3, 2);
31         int esperado = 5;
32         assertEquals(esperado, resultado);
33     }
34
35     @Test
36     public void testAnsSuma() {
37         System.out.println("testAnsSuma()");
38         calcV2.suma(3, 2);
39         int resultado = calcV2.ans();
40         int esperado = 5;
41         assertEquals(esperado, resultado);
42     }
43 }
```

Ejercicio: Termina la implementación de la clase CalculadoraV2Test.



No se deben confundir las anotaciones `@Before` y `@After` con `@BeforeClass` y `@AfterClass`, que veremos más adelante.

Vamos a hablar ahora de los parámetros de la notación `@Test`



```
@Test(  
public  
int  
int  
int  
expected : Class<? extends java.lang.Throwable> - Test  
timeout : long - Test
```

- Expected: se usa para indicar que se espera que el método lance una excepción determinada. Si durante su ejecución, el test no lanza una excepción del tipo que le hemos indicado, el resultado del test será fallo.
- Timeout: se usa cuando queremos limitar el tiempo máximo que puede durar una prueba. La prueba finalizará correctamente si termina antes del tiempo límite, de lo contrario terminará con fallo.

Ejercicio:

Crea un nuevo método `divisionMejorada(int a, int b)` en la clase `Calculadora.java` que devuelva una `ArithmeticException` en caso de que se intente dividir entre cero.

Añade dos métodos a la clase `CalculadoraTest.java`, que prueben el método `divisionMejorada()`. En uno de ellos deberás emplear la notación `@Test` con parámetro `expected` y en el otro sin parámetros. En ambas llamadas utilizarás los valores `a=3` y `b=0`.

¿Cuál ha sido el resultado de la ejecución de cada test?

En el test con el parámetro `expected`, cambia el parámetro `b` a 1

¿Cuál es ahora el resultado del test?

Ejercicio:

Crea un método factorial (iterativo) en la clase `Calculadora.java`

Añade un método a la clase `CalculadoraTest.java` que prueba dicho método. Emplea el parámetro `timeout` con el valor 1 (1 milisegundo) y comprueba a partir de qué número aproximado, el test termina con fallo.

Añade un método factorial (recursivo) en la clase `Calculadora.java` y su test correspondiente a `CalculadoraTest.java`. Compara la velocidad de ejecución de método iterativo y el recursivo.

Otra anotación interesante de Junit es `@Ignore`. Esta anotación debe situarse en la línea anterior a `@Test` y evita que el test en cuestión se ejecute, lo cual puede ser útil en distintos casos:

- El código a probar ha cambiado y el caso de uso no ha sido todavía adaptado.
- El tiempo de ejecución del método de test es demasiado largo para ser incluido.
- El método de test no está aun completamente implementado.
- La prueba accede a una base de datos a la cual no se dispone de conexión por el momento.
- Etc.

Como parámetro puede tener un texto que indique el motivo por el cual se ignora la prueba:

```
@Ignore("Se desactiva esta prueba hasta que tengamos privilegios de conexión con la base de datos")
@Test()
public void altaRegistroTablaClientes() {
```

Pasamos ahora a ver las anotaciones `@BeforeClass` y `@AfterClass`, que ya han sido mencionadas anteriormente.

Estas anotaciones tienen un comportamiento similar a `@Before` y `@After`, con la particularidad de que sólo se van a ejecutar una vez:

- `@BeforeClass`: Se va a ejecutar antes de cualquier prueba unitaria que se ejecute dentro de la clase.
- `@AfterClass`: Se va a ejecutar después de cualquier prueba unitaria que se ejecute dentro de la clase.

Vamos a modificar nuestra clase `CalculadoraV2` de manera que tengamos un método anotado con `@BeforeClass` que sea el encargado de crear el objeto `Calculadora` y un método anotado con `@Before`, que se ejecutará por tanto antes de cada método y que llame al método `clear()` encargado de resetear el valor de la variable `ans`, de manera que nos aseguremos que la calculadora está limpia antes de cada prueba.

Vamos a añadir también un método anotado con `@AfterClass`, al que añadiremos una traza, simplemente para que cuando ejecutemos los test podáis comprobar por consola que es el último método en ejecutarse.

Ejercicio: Como habrás notado, los métodos anotados con `@BeforeClass` y `@AfterClass` tienen que ser definidos como estáticos. Si tienes curiosidad por saber el motivo, te de animo a que lo investigues. Te dejo un par de enlaces para ayudarte que deberían ser de ayuda:

<https://stackoverflow.com/questions/1052577/why-must-junits-fixturesetup-be-static>

<https://martinfowler.com/bliki/JunitNewInstance.html>

En la siguiente captura se puede comprobar el orden de ejecución de los métodos de una clase de pruebas:

```
<terminated> CalculadoraV2TestV2 [JUnit]
beforeClass()
before()
testSuma()
after()
before()
testAnsSuma()
after()
afterClass()
```

En ocasiones puede interesarnos variar el orden de ejecución de nuestros test y para esto disponemos de la anotación `@FixMethodOrder` que nos permite definir tres métodos de ordenación de nuestros tests unitarios:

- **DEFAULT:** Ordenación determinista pero no predecible.
- **JVM:** Deja los métodos de prueba en el orden devuelto por la JVM.
- **NAME\_ASCENDING:** Ordenación según los nombres de los métodos.

Esta anotación se coloca en la línea anterior a la de la definición de la clase:

```
12 @FixMethodOrder(MethodSorters.JVM)
13 public class CalculadoraTest {
```

### Pruebas parametrizadas

Cuando realizamos pruebas unitarias siempre queremos que las mismas sean lo más fuertes posibles, para lo cual resulta indispensable repetir las pruebas con distintos parámetros y probar ese test con varios casos para asegurarnos de que en realidad funciona de manera correcta.

Una prueba unitaria parametrizada no es más que un método de prueba en el cual deben realizarse como si de un bucle for se tratara una prueba por cada grupo de parámetros, esto hace la prueba unitaria mucho más fuerte ya que estamos verificando que funcione correctamente en todos los casos posibles.

Para ello vamos a utilizar la clase Parametized, la cual con simples anotaciones nos permite correr nuestros test fácilmente y para verlo mediante un ejemplo vamos a crear una nueva clase llamada SumaTest.java que se va encargar de hacerle la prueba unitaria al método suma(int a, int b) de la clase CalculadoraV2.

### ¿Cómo funcionan?

1. Al utilizar el runner `@RunWith` con el parámetro `Parametized.Class`, se le indica a JUnit que invoque a la clase referenciada en la anotación para lanzar el test, en lugar de al lanzador por defecto.

Con este Runner, JUnit instanciará la clase de test (SumaTest.java) tantas veces como juego de datos tenga.

2. Se define un constructor que recibe los parámetros de cada juego de datos.

Cada vez que el Runner instancia la clase le pasa un nuevo juego de datos por el constructor.

3. Se define un método con la anotación `@Parameters`. De esta manera se indica a Junit cual es el método que va a devolver los parámetros a utilizar por el runner.

Este método debe devolver una colección iterable de arrays de objetos, donde cada array debe tener tantos elementos como parámetros de entrada tenga el método que vamos a anotar con `@Test` y dispuestos además en el mismo orden que los recibe el método.

4. Por último, se crea el método anotado como `@Test`, el cual, se ejecutará una vez por cada juego de datos. Es posible definir más de un método `@Test`.

Siguiendo estas indicaciones, el código de la clase `SumaTest.java`, quedaría:

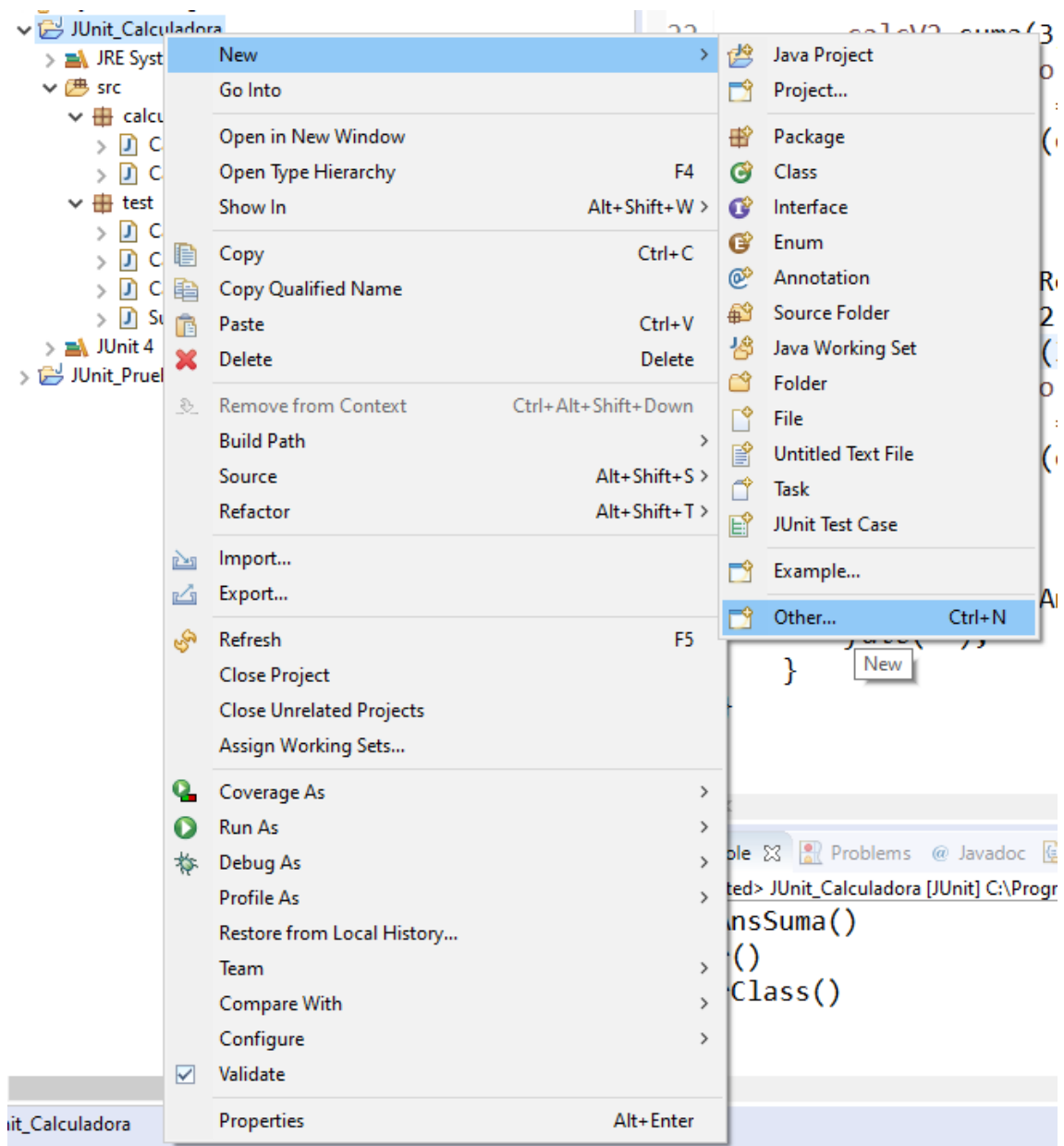
```
1 package test;
2
3 import static org.junit.Assert.*;
12
13 @RunWith(value = Parameterized.class)
14 public class SumaTest {
15
16     @Parameters
17     public static Iterable<Object[]> getData() {
18         List<Object[]> obj = new ArrayList<>();
19         obj.add(new Object[] { 3, 1, 4 });
20         obj.add(new Object[] { 2, 3, 5 });
21         obj.add(new Object[] { 3, 3, 6 });
22         return obj;
23     }
24
25     private int a, b, esperado;
26
27     public SumaTest(int a, int b, int esperado) {
28         this.a = a;
29         this.b = b;
30         this.esperado = esperado;
31     }
32
33     @Test
34     public void sumaTest() {
35         CalculadoraV2 calculadora = new CalculadoraV2();
36         int resultado = calculadora.suma(a, b);
37         assertEquals(esperado, resultado);
38     }
39
40 }
```

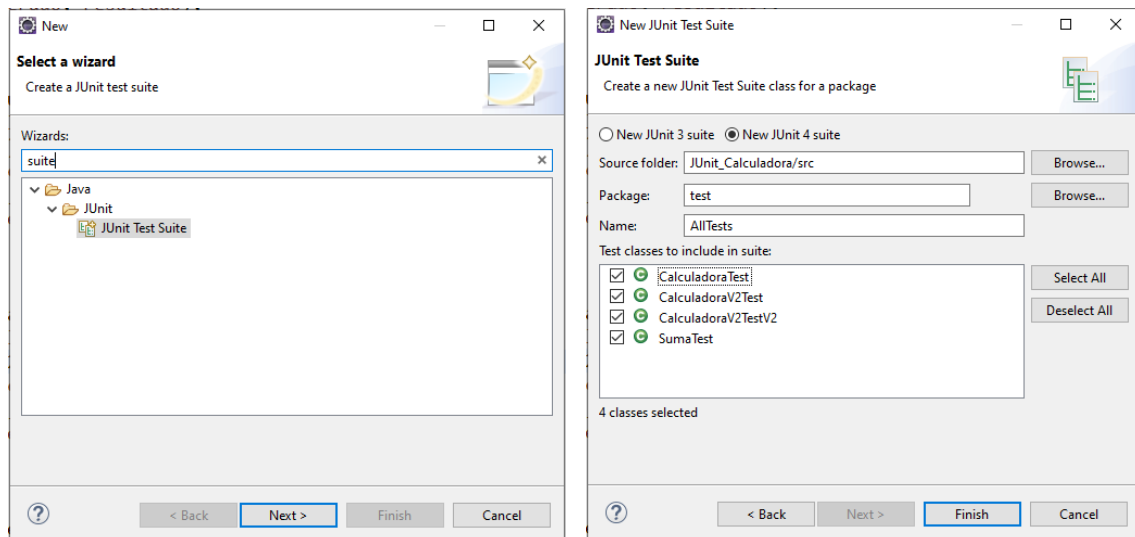
Ejercicio: Simplifica el método `getData()` empleando el método `Arrays.asList()`

## Suites

Los Test Suite, son un conjunto de test agrupados, generalmente los test automatizados se encuentran separados en clases lo cual hace que sea bastante engorroso realizar las pruebas una a una, de ahí surge la idea de agruparlos y ejecutar varias clases de prueba una detrás de otra en el orden que nosotros determinemos.

Para crear una suite de pruebas JUnit en Eclipse, podemos hacerlo de manera sencilla haciendo clic derecho sobre nuestro proyecto y en New->Other, elegiremos JUnit Test Suite.





Una vez hecho esto aparecerá una ventana donde podremos dar un nombre a nuestra suite y escoger las clases de prueba que queremos que incluya.

Vamos a echar ahora un ojo al código de la suite:

```

1 package test;
2
3 import org.junit.runner.RunWith;
4
5
6
7 @RunWith(Suite.class)
8 @SuiteClasses({ CalculadoraTest.class,
9                 CalculadoraV2Test.class,
10                 CalculadoraV2TestV2.class,
11                 SumaTest.class })
12 public class AllTests {
13
14 }

```

Como vemos, la clase incluye las notaciones `@RunWith`, que ya vimos junto con las pruebas parametrizadas y la notación `@SuiteClasses` que como parámetro recibe un array con todas las clases de prueba que queremos que se ejecuten. Podemos variar el orden de ejecución de las pruebas simplemente cambiando el orden de las clases en el array.

**Ejercicio:** Siguiendo el ejemplo de `SumaTest.java`, crea una clase de pruebas parametrizadas para `CalculadoraV2.rest()`, de nombre `RestaTest.java` y añade ambas clases a una suite de pruebas.