TAREA ED03

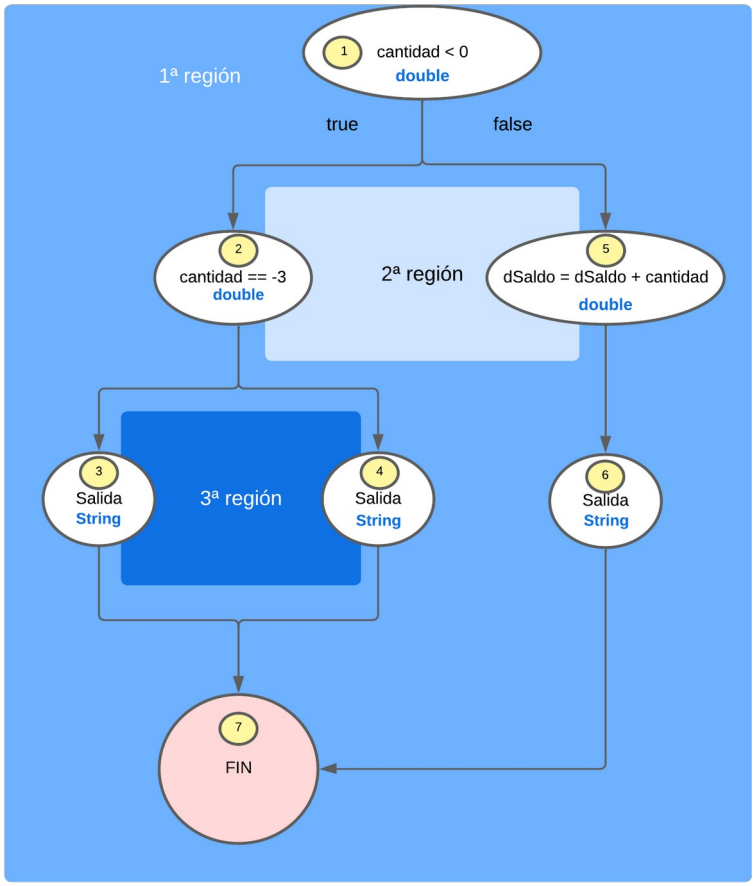
1.Realiza un análisis de caja blanca completo del método ingresar.
Son pruebas estructurales que permiten verificar la estructura interna de cada componente de la aplicación.
Para ello se realizan pruebas del software con el fin de ejecutar todas las lineas del programa.
No pretenden asegurar la corrección de los resultados producidos, sino que comprueban que se ejecuten todas las instrucciones del programa y los camino lógicos que va a recorrer el programa.
Para ello, se utilizan los siguientes criterios de cobertura:

- Cobertura de sentencias: Comprobamos, mediante casos de prueba, que cada sentencia se ejecuta, al menos, una vez.
- Cobertura de decisiones: Comprobamos, mediante casos de prueba, que cada opción de resultado se evalúe, al menos, una vez a cierto y otra a falso.
- Cobertura de condiciones: Comprobamos, mediante casos de prueba, que cada elemento de una condición se evalúe, al menos, una vez a cierto o falso.
- Cobertura de condiciones y de decisiones: Comprobamos, mediante casos de prueba, que cumplen ambas simultáneamente.
- Cobertura del camino de prueba: Comprobamos, mediante casos de prueba, que se ejecuta el programa, al menos una vez, desde la sentencia inicial a la final (lo que se denomina camino), pudiéndose realizar en dos variantes: una indica que cada bucle debe ejecutarse sólo una vez (hacerlo más veces no aumenta la efectividad) y otra que recomienda que se pruebe cada bucle tres veces (una sin entrar en su interior, otra ejecutándolo una vez y otra dos veces).

Para **determinar los casos de prueba de caja blanca**, que garantice que los caminos definidos en el código se llegan a recorre, se realizan los siguientes pasos:

1. **Crear un grafo** que represente el código a probar.
   Los grafos se construyen a partir de **nodos** (secuencias de instrucciones consecutivas donde no hay alternativas en la ejecución o condiciones a evaluar) y **aristas** (encargadas de unir los nodos). En el caso de que las decisiones tengan **múltiples condiciones**, se separa **cada condición en un nodo**.
   Estos grafos han sido realizados con el programa Lucidchart.

Método ingresar



En el método ingresar se pasa como argumento una variable (“cantidad”) de tipo double; si esta variable es menor a 0 y es igual a -3, se produce una salida de tipo String y se devuelve el código de error 2, y sino, produce una salida de tipo String y devuelve un código de error 1, y si mayor que 0, se suma dicha variable a la variable “dSaldo”, de tipo double, y se produce una salida de tipo String (mediante llamada al método println de la clase System).

2. **Calcular la complejidad ciclomática (de McCABE)** a partir del grafo obtenido. Se puede realizar mediante **3 métodos diferentes**:
   - **V(G) = a - n + 2**, siendo a el número de aristas del grafo y n el número de nodos.
   - **V(G) = c + 1**, siendo c el número de nodos de condición.
   - **V(G) = r**, siendo r el número de regiones cerradas del grafo (incluida la externa).

   En el método ingresas obtenemos la siguiente complejidad:

   V(G) de ingresar = 7 arcos – 6 nodos + 2 = 3.

   V(G) de ingresar = 2 nodos de condición + 1 = 3.

   V(G) de ingresar = 3 regiones (una de ellas externa).

3. **Determinar tantos caminos o recorridos del grafo** como la complejidad ciclomática calculada.
   El **número de caminos de prueba** debe ser igual a la complejidad calculada. Consiste en hacer recorridos desde el inicio hasta el final del método, registrando los nodos por los que va pasando la ejecución del camino.
   Cada nuevo camino deberá aportar el **paso por nuevas aristas/nodos** del grafo. La definición de caminos se hará **desde los más sencillos a los más complicados**.

   - Camino 1: 1-2-3-6
   - Camino 2: 1-2-4-6
   - Camino 3: 1-5-6

4. **Generar un caso de uso para cada camino**, determinando datos de entrada y resultados esperados.

   Definimos los datos de entrada al programa que nos permitan recorrer los caminos definidos en el apartado anterior. El **conjunto de entradas al programa** utilizados en la ejecución de cada camino se denomina **caso de uso**.
   Además **definimos los resultados previstos** para que cuando posteriormente se lance la ejecución del programa para cada caso de uso, los resultados obtenidos serán comparados con los esperados y así determinar la corrección del código.

Método	Camino/Caso uso	Cantidad	Resultado esperado
ingresar	1	-3	Mensaje: “Error detectable en pruebas de caja blanca”. Devuelve el código de error 2.
	2	-2	Mensaje: “No se puede ingresar una cantidad negativa”. Devuelve el código de error 1.
	3	10	Incrementa el valor de la variable saldo en la cantidad indicada. Devuelve el código de error 0.

5. **Lanzar una ejecución del programa por cada uso** y comprobar si los resultados obtenidos son los esperados comprobando la corrección de código.

2.Realiza un análisis de caja negra, incluyendo valores límite y conjetura de errores del método retirar. Debes considerar que este método recibe como parámetro la cantidad a retirar, que no podrá ser menor a 0. Además en ningún caso esta cantidad podrá ser mayor al saldo actual. Al tratarse de pruebas funcionales no es necesario conocer los detalles del código pero te lo pasamos para que lo tengas.

El propósito de las pruebas de caja negra o funcionales es **comprobar si las salidas que devuelve la aplicación son las esperadas** en función de los parámetros de entrada. En este tipo de prueba **no se considera la implementación del código**.

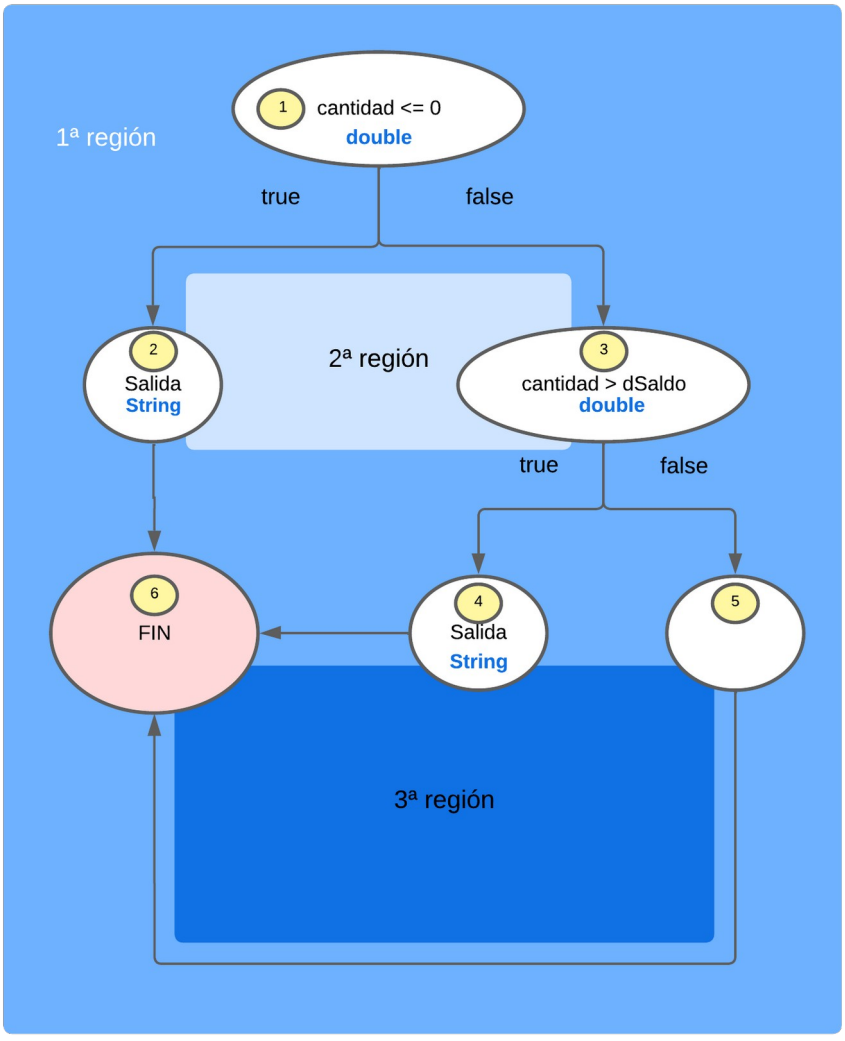
La técnica para determinar los casos de prueba de caja negra se realiza completando los siguientes pasos:

- Determinar las clases de equivalencia: se considerar el **menor número posible de casos de prueba** (cada caso de prueba tiene que abarcar el mayor número posible de entradas diferentes). Lo que se pretende, es **crear un conjunto de clases de equivalencia** donde la prueba de un valor representativo de la misma, en cuanto a la verificación de errores, **sería extrapolable** al que se conseguiría probando cualquier valor de la clase.
  Los **pasos a seguir para identificar las clases de equivalencia** son:

  - **Identificar las condiciones de las entradas del programa**, es decir, restricciones de formato o contenido de los datos de entrada.
  - A partir de ellas, identificar clases de equivalencia que pueden ser:
    - De **datos válidos**.
    - De **datos no válidos** o erróneos.

  Si se sospecha que ciertos elementos de una clase no se tratan igual que el resto de la misma, deben dividirse en clases menores.

Método retirar



Método	Condición de entrada	Clases válidas	Clases no válida
retirar	cantidad < saldo	(1) cantidad <= 100 (Probando con un saldo de partida de 100)	(2) cantidad >100
			(3) El parámetro recibe char o String en lugar de double
	cantidad > 0	(4) cantidad >=0	(5) cantidad < 0
			(6) El parámetro recibe char o String en lugar de double

- Análisis de valores límite (AVL): se eligen como **valores de entrada aquellos que se encuentra en el límite** de las clases de equivalencia. En las pruebas AVL también habría que generar c**asos de prueba atendiendo a clases de equivalencia** de los datos de salida:

  - ✔ En el caso de prueba (1): cantidad = 100
  - ✔ En el caso de prueba (2): cantidad = 101
  - ✔ En el caso de prueba (4): cantidad = 0
  - ✔ En el caso de prueba (5): cantidad = -1

- Pruebas aleatorias: consiste en generar **entradas aleatorias**. Para ello se suelen utilizar generadores de prueba, que son capaces de crear un volumen de casos de prueba al azar, con los que será alimentada la aplicación.

- Conjetura de errores: trata de generar **casos de prueba que la experiencia ha demostrado generan típicamente errores**. Se trata de una técnica menos métodica que las anteriores, tiene mucho más que ver con la intuición y experiencia del programador. En valores numéricos, un ejemplo típico es comprobar si funciona correctamente con el valor 0.

- Casos de uso, resultados y análisis: Ahora toca **definir los datos de entrada al programa**. Al conjunto de entradas al programa utilizados para cada ejecución se le denomina **caso de uso**. Los casos de uso **se generarán a partir de las clases de equivalencia, valores límite y conjeturas de errores obtenidos** en los apartados anteriores.
  Este proceso consta de las siguientes fases:

  - Numerar las clase de equivalencia.
  - Crear casos de uso que cubran todas las clases de equivalencia válidas. Se intentará agrupar en cada caso de uso tantas clases de equivalencia como sea posible.
  - Crear un caso de uso para cada clase de equivalencia no válida.

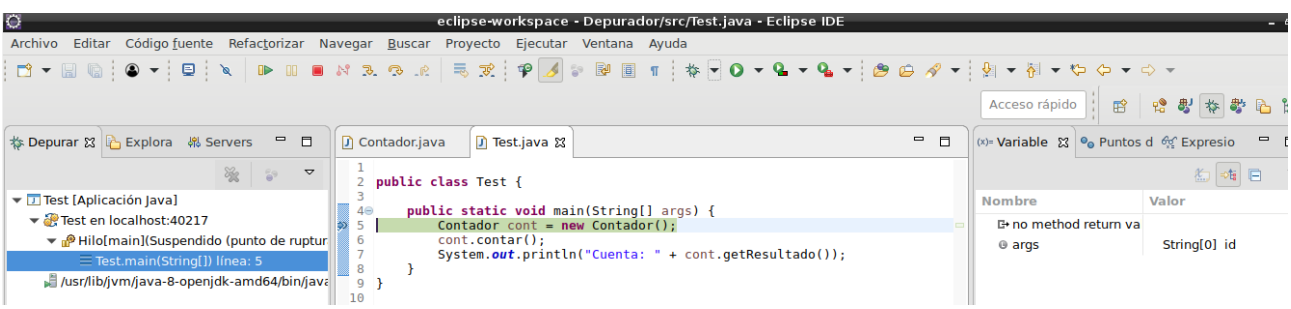
  Además toca definir los r**esultados previstos en cada ejecución**. Cuando posteriormente se lance la ejecución del programa para cada caso de uso, **los resultados obtenidos serán comparados con los esperados y así determinar la corrección del código.**

Método	Caso	Cantidad	Resultado esperado
retirar	1	100	El valor del saldo se reduce en 100
	2	101	El programa genera un mensaje de saldo insuficiente
	3	‘H’	El programa genera una excepción controlando el error provocado al recibir como parámetro un char en lugar de un double
	4	0	El valor del saldo se reduce en 0
	5	-1	El programa genera un mensaje advirtiendo de que la cantidad solicitada para retirar es menor de 0
	6	‘H’	El programa genera una excepción controlando el error provocado al recibir como parámetro un char en lugar de un double

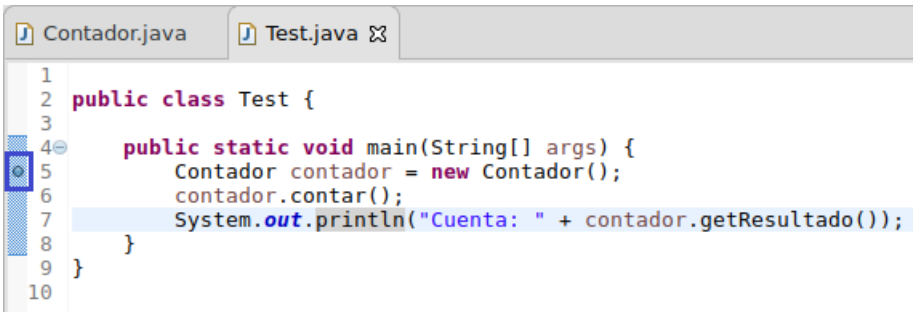
3.Crea la clase CCuentaTest del tipo Caso de prueba JUnit en Eclipse que nos permita pasar las pruebas unitarias de caja blanca del método ingresar. Los casos de prueba ya los habrás obtenido en el primer apartado del ejercicio. Copia el código fuente de esta clase en el documento.

Existen varias alternativas para **lanzar la ejecución de un programa en modo debug(depuración)**. Una de ellas es pulsando con el botón derecho del ratón sobre la clase de inicio del proyecto (implementa el método main), y seleccionar en el menú contextual que aparece "Depurar como => Aplicación Java".

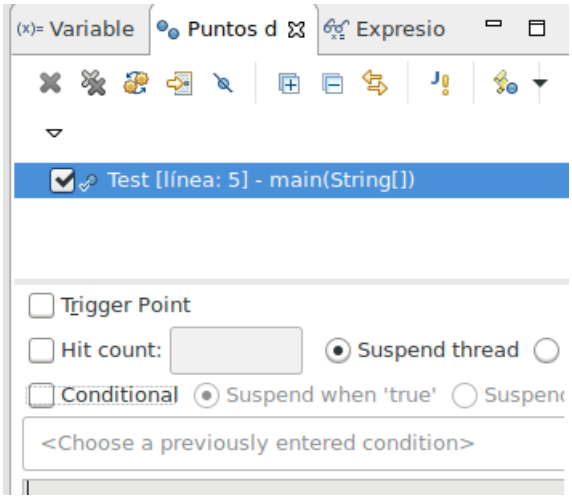
En **modo depuración**, Eclipse da la opción de trabajar con la perspectiva depurar, que ofrece una serie de vistas muy interesantes para este tipo de ejecución, tales como: la **vista de visualización y cambio de variables, la vista de puntos de parada establecidos o la pila de llamadas** entre otras:



- Puntos de ruptura: Al solicitar una ejecución en modo debug, **si no hemos establecido puntos de parada en el código, éste ejecutará hasta el final** del mismo modo que lo haría en una ejecución normal.
  Para **establecer un punto de parada**, basta con hacer doble clic en el margen izquierdo de la línea de código donde se va a establecer.



El punto de parada es dado de alta y ya **aparece en la vista de puntos de parada**.

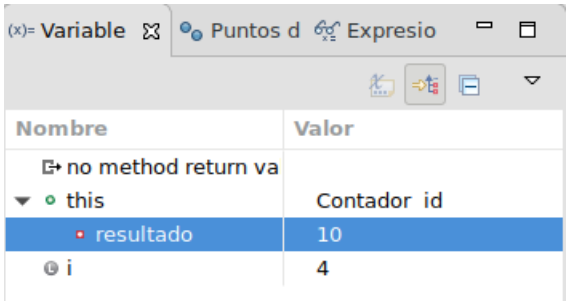


Eclipse **permite definir puntos de parada condicionales** para personalizar cuándo o porqué se para el programa. Si por ejemplo, seleccionamos en un punto de parada la **opción Hit count(impac tos)** y determinamos un valor, **el programa parará cuando haya pasado por este punto el número de veces indicado**.

Es posible crear condiciones más elaboradas dependientes de otras variables disponibles en el contexto de la ejecución. **El programa sólo parará cuando la condición definida se cumpla y la ejecución pase por esa línea de código.**

En el momento que tenemos detenido el programa, se pueden realizar diferentes labores: por un lado, se pueden **examinar las variables**, y **comprobar que los valores que tienen asignados son correctos**, o se pueden **iniciar una depuración paso a paso** e ir comprobando el camino que toma el programa a partir del punto de ruptura. Una vez realizada la comprobación, podemos abortar el programa, o continuar la ejecución normal del mismo.

- Examinadores de variables: Durante el proceso de implementación y prueba de software, una de las maneras más comunes de comprobar que la aplicación funciona de manera adecuada, es **chequear que las variables vayan tomando los valores adecuados en cada momento**. Eclipse nos proporciona la **vista variables** donde podemos ir comprobando el valor que van tomando las variables activas en la zona de código donde está el programa parado.

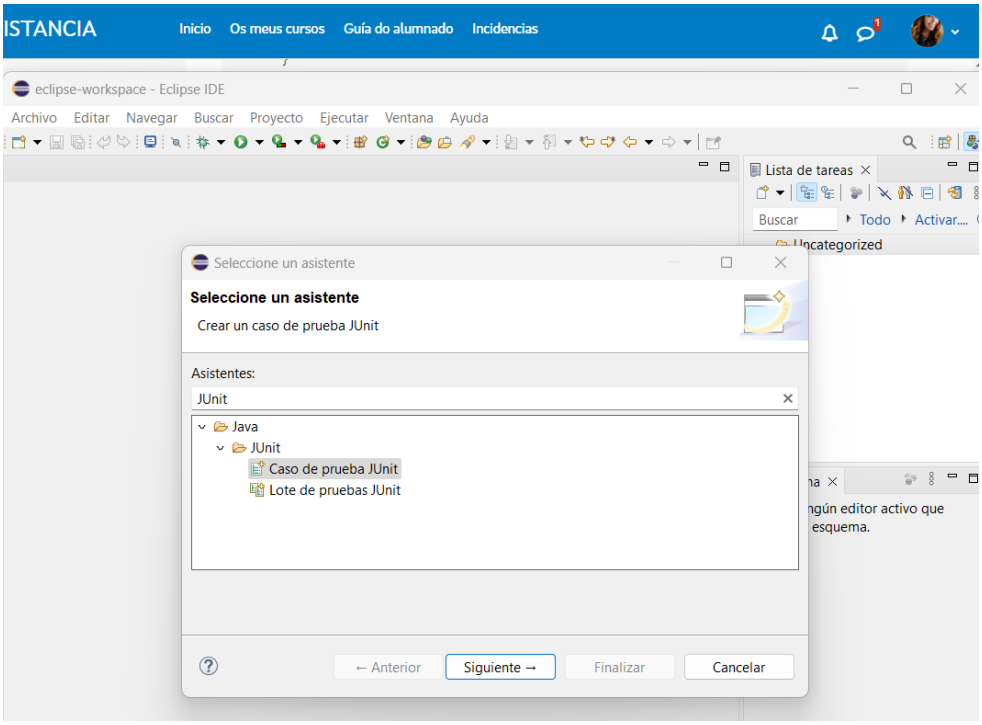


Además, pulsando sobre el valor de cualquiera de estas variables es posible modificarlas. Esto nos permite **evaluar nuevos escenarios de prueba con datos diferentes**.

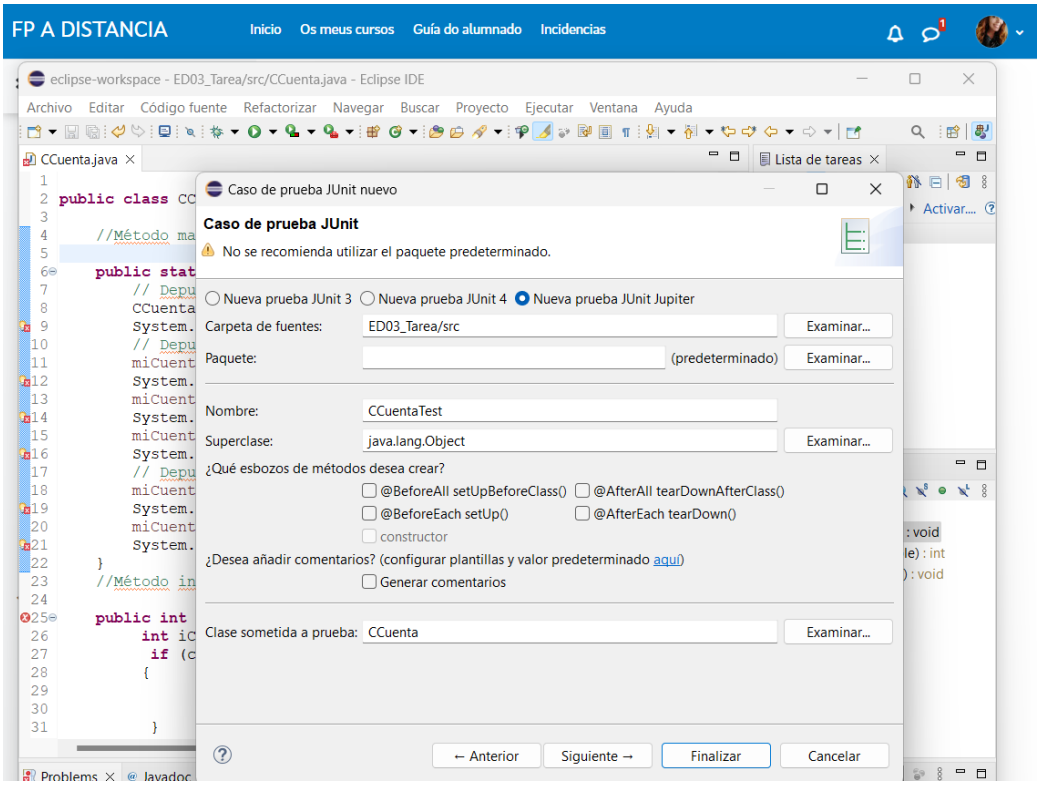
- Botones de depuración: Cuando estamos en el proceso de depuración de un programa con la perspectiva Depurar, Eclipse nos ofrece una serie de **botones en su barra de herramientas:**

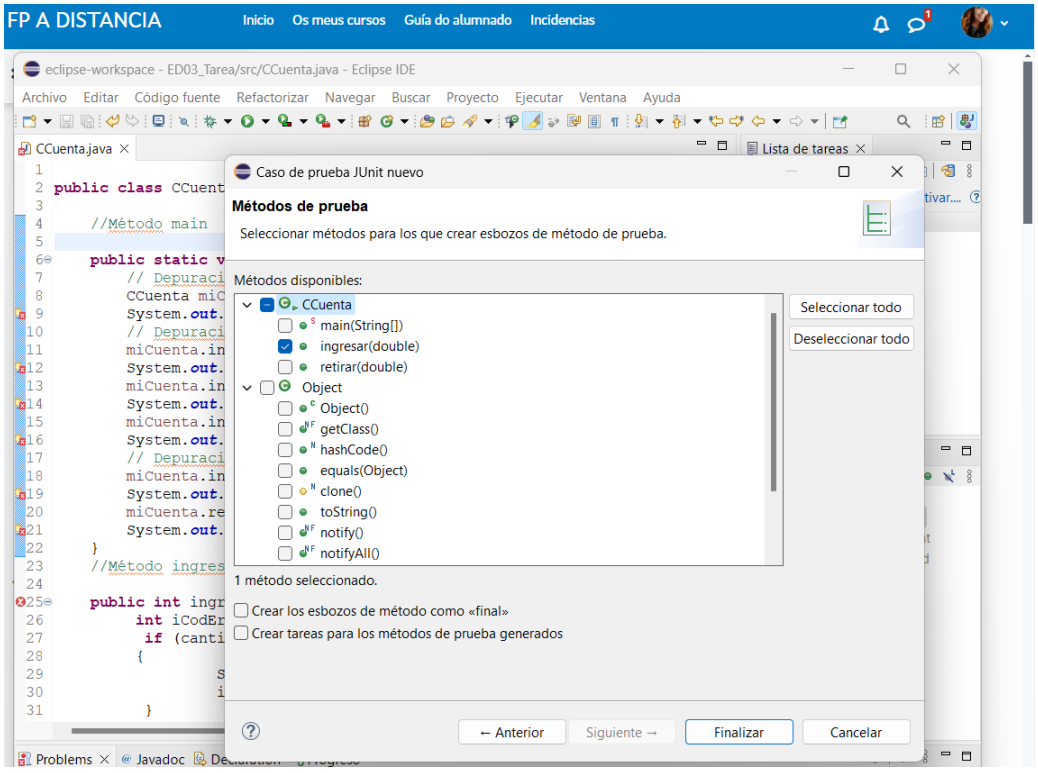
  - Desactiva temporalmente todos los puntos de parada del código.
  - Continua la ejecución del programa. Se detendrá en el siguiente punto de parada.
  - Pausa/detiene la ejecución del programa en el punto de código donde se encuentre al ser pulsado.
  - Finaliza la ejecución del programa.
  - Función no considerada en este manual. Consultar manual de Eclipse.
  - Si el programa se encuentra detenido en la llamada a un método, al pulsar este botón la ejecución pasa a la primera línea del mismo.
  - Si el programa se encuentra detenido en la llamada a un método, al pulsar este botón la ejecución del método se hace por completo, sin depurar su implementación.
  - Avanza la ejecución del programa hasta que nos salimos del método actual y vamos hasta el sitio donde fue llamado.

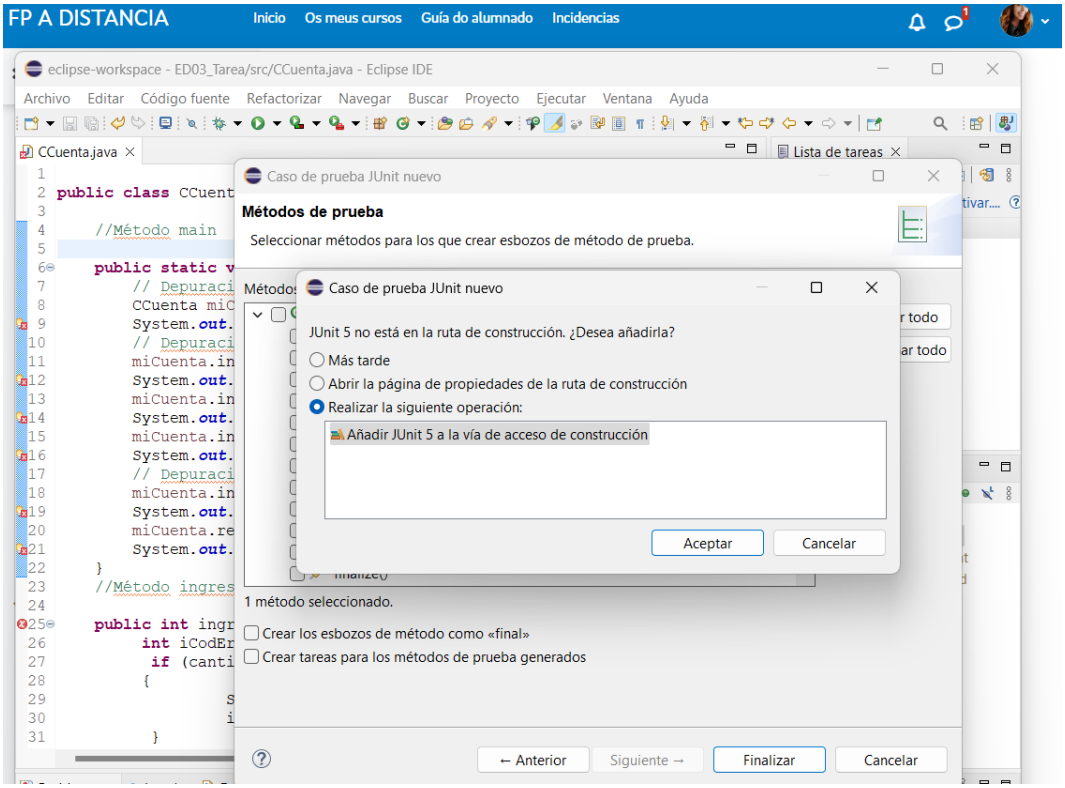
Creamos una clase CcuentaTest del **tipo Caso de Prueba JUnit en Eclipse** (Archivo > Nuevo > Otros > Unit > Caso de prueba Junit.



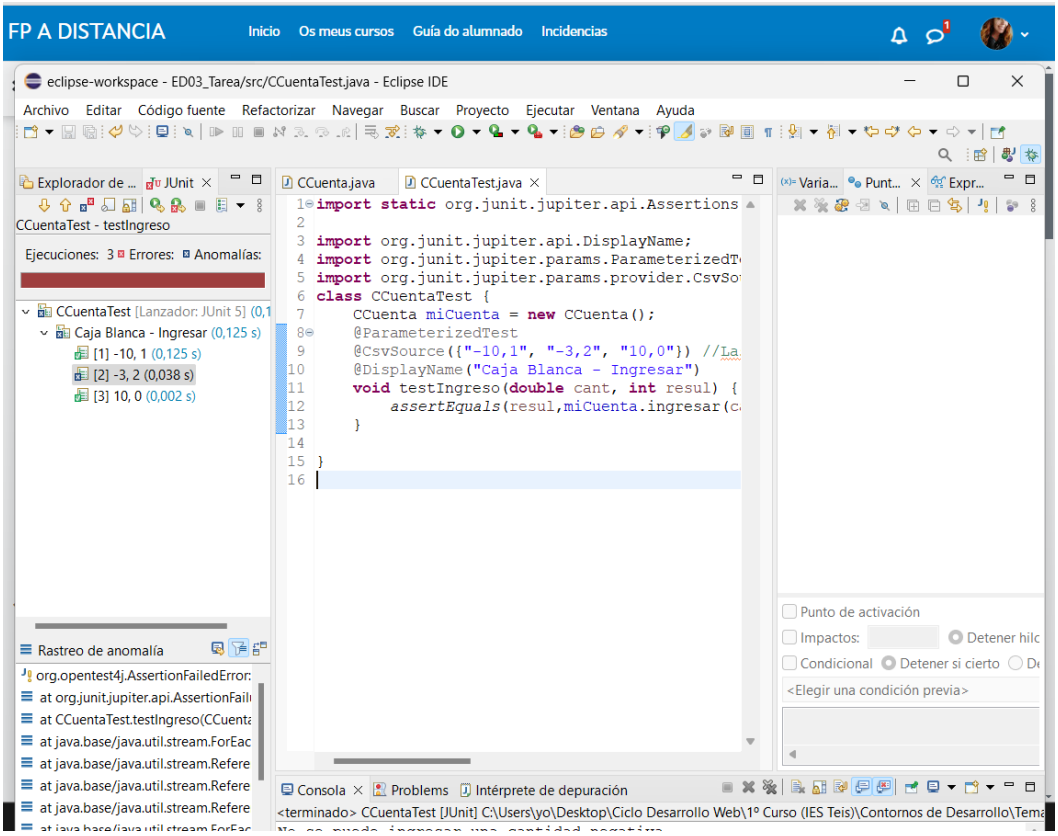
Apareciendo el asistente donde **indicamos el nombre de la Clase de Prueba, la versión de Junit** que se va a utilizar (“New Junit jupiter test”), **seleccionar el método** (ingresar) para ser probado y **aceptar que la librería Junit sea incluida**.







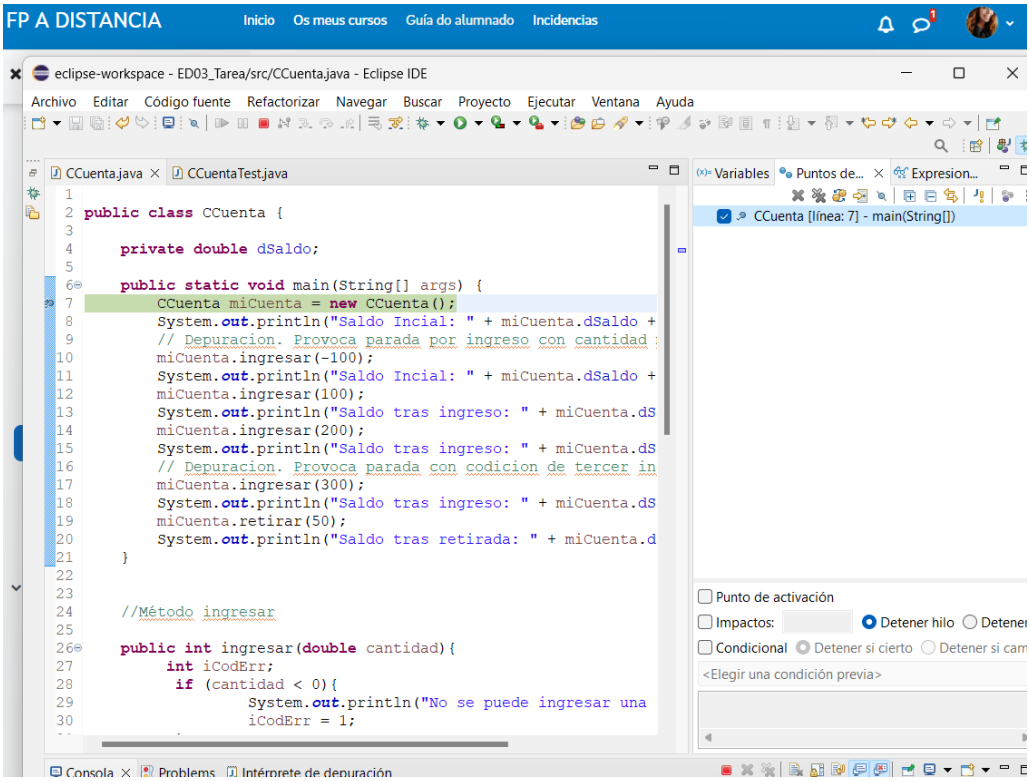
Una vez aceptado, Eclipse **crea la estructura de la clase de prueba** y lo actualizaremos de la siguiente manera, **para después ejecutarlo**:



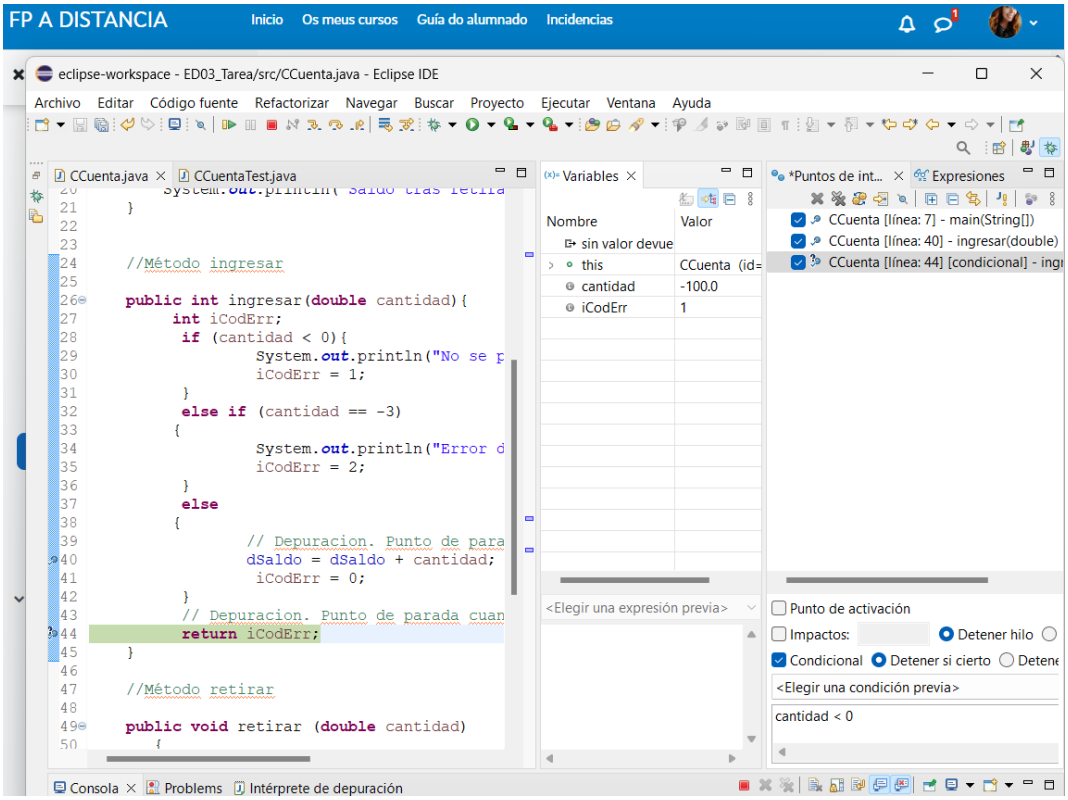
Ejecutamos el la Clase CCuentaTest y comprobamos que los casos de prueba 1 y 3 son satisfactorios y en el caso 2 advierte de fallo, como muestra la imagen.

4.Genera los siguientes puntos de ruptura para validar el comportamiento del método ingresar en modo depuración.
Conmutamos los siguientes puntos de parada y depuramos:

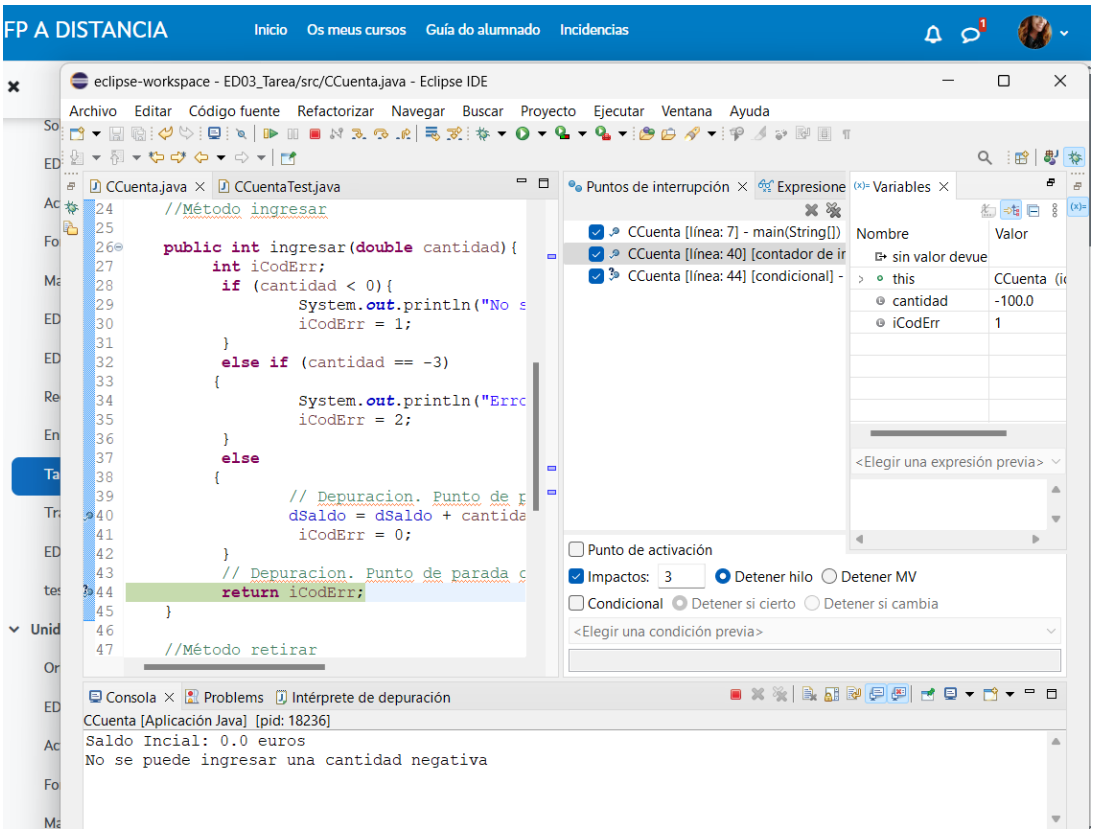
- **Punto de parada sin condición al crear el objeto miCuenta en la función main** (Línea 7 del código del método main).



- **Punto de parada en la instrucción return del método ingresar sólo si la cantidad a ingresar es menor de 0** (Línea 44 del código del método ingresar). Activamos el punto de ruptura e incluimos la condicion cantidad < 0.
Depuramos y observamos que durante la depuración el programa se ha parado y en la Vista de Variables el valor de la cantidad es menor que 0 (-100).



- **Punto de parada en la instrucción donde se actualiza el saldo**, sólo deberá parar la tercera vez que sea actualizado (Línea 40 del código del método ingresar).



Activamos el puntos de ruptura y el valor del campo Impactos a 3.
Depuramos y observamos que el programa nos informa en la consola de dos ingresos anteriores.
Por lo tanto ha parado en la 3ª actualización de la variable dSaldo mediante ingreso.