

# UNIDAD 3



## Diseño y realización de pruebas

### Objetivos

- Comprender la filosofía de las pruebas del software.
- Emplear las estrategias adecuadas para la realización de pruebas a lo largo del ciclo de vida del software.
- Conocer y aplicar técnicas de diseño de casos de prueba de caja blanca y de caja negra.
- Conocer de qué manera se documentan las pruebas.
- Utilizar herramientas de depuración en un entorno de desarrollo.
- Realizar pruebas automáticas en un entorno de desarrollo.

### Contenidos

- 3.1. Filosofía de las pruebas del software
- 3.2. Estrategia de pruebas del software
- 3.3. Técnicas de diseño de casos de prueba
- 3.4. Documentación de las pruebas
- 3.5. Herramientas de depuración
- 3.6. Pruebas automáticas

# Introducción

Cuando un equipo de desarrollo acepta el encargo de crear una aplicación informática, desea que el resultado de su trabajo sea de calidad, esto es, que la aplicación responda a las necesidades del cliente y que tenga el menor número posible de errores. El objetivo de la etapa de pruebas, posterior a la programación, es descubrir dichos defectos antes de que el software sea entregado al cliente.

En esta unidad, se explica cómo abordar esta importante tarea y las diferentes técnicas de diseño de casos de prueba. También se verá de qué manera un entorno de desarrollo puede asistir a las personas que desarrollan programas informáticos en las tareas de depuración o corrección de errores encontrados en esta fase. Para finalizar, se describe el empleo de una herramienta que sirve para la realización de pruebas unitarias de manera automatizada.

## 3.1. Filosofía de las pruebas del software

Las pruebas constituyen una de las tareas del ciclo de vida del software y tienen como objetivo fundamental la detección de defectos que se han cometido inadvertidamente durante el proceso de construcción del software.

Cabe hacer hincapie aquí que el objetivo de esta tarea es la detección de defectos y no precisamente la demostración de que el software no tiene ninguno. Y es que, aunque se trate de una pequeña aplicación, es imposible probar exhaustivamente el software, es decir, es imposible encontrar todos sus errores. El objetivo debe ser pues realizar pruebas que no supongan un esfuerzo excesivo y que haya una elevada probabilidad de que se detecten los fallos, es decir, se trata de intentar encontrar un equilibrio entre el esfuerzo requerido para esta tarea y la capacidad de detección de defectos. En este sentido, también es necesario desmitificar una creencia común: el hecho de que se detecten defectos durante este proceso es un indicativo de la negligente actuación de los desarrolladores de software. Nada más lejos de la realidad: como seres humanos que son, se equivocan, lo que tiene como consecuencia que el resultado de su trabajo (en este caso, el software) puede contener errores. Así pues, el objetivo debe ser detectar estos fallos antes de que el software sea entregado al cliente, porque es probable que si no los encuentran quienes han desarrollado el programa, los detecte el cliente una vez que el producto ha sido entregado, y esto, sin lugar a dudas, no es deseable.

Otro factor que juega en contra de la consideración adecuada de las pruebas es que si bien las tareas de análisis, diseño y programación se consideran tareas constructivas, las pruebas pueden ser consideradas por parte del equipo desarrollador como psicológicamente destructivas. En consecuencia, el desarrollador o la desarrolladora actuará con cuidado, y diseñará y ejecutará pruebas que demostrarán que el programa funciona, en lugar de descubrir errores. Sin embargo, dado que las personas que desarrollan software no son infalibles, es probable que haya errores, y por ello, es conveniente hacer un esfuerzo por encontrarlos antes de que el cliente los descubra, lo que sería lamentable.

Piattini *et al.* (2007), en relación con la actitud necesaria para acometer las pruebas, nos indican que «un buen caso de prueba es aquel que tiene una gran probabilidad de encontrar un defecto no descubierto aún y que el éxito de una prueba consiste en detectar un defecto no encontrado antes. El contraste con la visión tradicional de las pruebas ('Vamos a probar un par de opciones para comprobar que funciona y ya está') es radical».

Por estas razones, es necesario cambiar la mentalidad en relación con las pruebas del software. Lo indicado anteriormente no quiere decir que las desarrolladoras y desarrolladores no deban probar sus propios programas. De hecho, siempre suelen ser ellas y ellos mismos las personas encargadas de realizar las pruebas de más bajo nivel o las pruebas de unidades individuales. No obstante, en el caso de proyectos grandes y sobre todo en pruebas a mayor nivel, se suele involucrar en esta tarea a un equipo de pruebas independiente, con el que deben trabajar conjuntamente las personas encargadas del desarrollo del software para garantizar que se realicen pruebas exhaustivas del programa.

El objetivo de las pruebas puede percibirse desde dos puntos de vista:

1. Mediante las pruebas se pretende, por un lado, comprobar si se está construyendo el producto correcto, es decir, el que desea el cliente, lo cual se relaciona con el concepto de **validación**.
2. Por otro lado, se trata de comprobar si el producto que se está desarrollando funciona correctamente, es decir, si lo que hace lo realiza de manera correcta. Esto se refiere al concepto de **verificación**.

Así pues, mediante las pruebas se debe comprobar tanto si el producto es el que quiere el cliente (validación) como si está construido correctamente, de manera que lo que hace lo hace bien (verificación).

Asimismo, en relación con las pruebas, hay dos aspectos estratégicos, relacionados con su puesta en práctica, que es preciso considerar:

1. La **estrategia de aplicación de las pruebas**, es decir, debe establecerse qué partes del software deben someterse a pruebas a medida que se va desarrollando. Este aspecto se analizará a fondo en el Apartado 3.2.
2. Las **técnicas de diseño de casos de prueba** que se van a emplear. O dicho de otro modo, una vez seleccionada la parte del software que se someterá a prueba, se deberá decidir de qué manera se debe llevar a cabo esta, es decir, habrá que seleccionar la técnica apropiada. Estas técnicas se estudiarán en detalle en el Apartado 3.3.

El IEEE (por sus siglas en inglés, Institute of Electrical and Electronics Engineers) proporciona la siguiente definición para caso de prueba: «un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular como, por ejemplo, ejercitarse un camino concreto de un programa o verificar el cumplimiento de un determinado requisito». Es decir, un caso de prueba consiste básicamente en indicar qué datos de entrada se deben suministrar al programa, qué datos de salida se deben obtener y cuál es el fin de llevar a cabo dicho caso de prueba. Una vez ejecutada la prueba, se comprueba si la salida obtenida coincide con la salida esperada. La coincidencia indicará que no se ha detectado defecto alguno para ese caso de prueba; en caso contrario,

se realiza una tarea conocida con el nombre de **depuración**. La depuración es, por tanto, el resultado de una prueba exitosa y consiste en localizar el error y corregirlo, para lo que puede ser necesario efectuar nuevos casos de prueba.

### Recuerda



El hecho de que se encuentren errores a lo largo de la fase de pruebas de una aplicación no debe entenderse como un fracaso del trabajo previo realizado. Al contrario, se debe considerar como algo positivo el hecho de que se encuentren errores después de terminada la programación, pues esto permite eliminarlos antes de entregar la aplicación y evita que estos sean descubiertos por el cliente. No se debe olvidar que «todos cometemos errores» y, como dice el dicho popular, «rectificar es de sabios».

## 3.2. Estrategia de pruebas del software

Las pruebas siempre comienzan por los componentes más pequeños, es decir, se empieza probando pequeñas porciones de software y se va incrementando de manera progresiva el alcance de la prueba. Se suelen llevar a cabo cuatro tipos de pruebas secuenciales que se exponen en los siguientes subapartados.

### 3.2.1. Prueba de unidad

Se comienza probando o bien módulos individuales de software, en caso de que se desarrolle software convencional, o bien clases, si se trata de software orientado a objetos. En este último caso, la prueba de unidad conlleva realizar también pruebas a nivel de método, es decir, para cada método no trivial habrá que comprobar si su comportamiento es el adecuado. Es obvio pues que para probar correctamente una clase entera, es necesario realizar pruebas con cada uno de sus métodos.

### Argot técnico



Cuando se habla de **software convencional**, ello se refiere al software creado siguiendo el **paradigma estructurado**, según el cual una aplicación consta de varios componentes o módulos (procedimientos o funciones), de manera que existe un módulo de control principal, que llama a otros módulos, y estos a otros y así sucesivamente.

Recuérdese que, por el contrario, en el **paradigma orientado a objetos**, una aplicación consta de varias clases, y que cada una de estas se compone de atributos y operaciones, llamadas métodos. A partir de una clase, se pueden crear objetos o instancias de ella. La ejecución de la aplicación se consigue mediante el envío de mensajes de unos objetos a otros.

Puesto que, hoy en día, se emplea mayoritariamente el paradigma orientado a objetos, este libro se centra casi exclusivamente en este.

## 3.2.2. Prueba de integración

El objetivo de estas pruebas, que se realizan a continuación de las pruebas de unidad, es comprobar si las clases de las que consta el software funcionan correctamente cuando cooperan entre ellas con el objetivo de realizar las tareas encomendadas al programa.

Existen dos estrategias para las pruebas de integración en los sistemas orientados a objetos:

1. **Prueba basada en hebra:** integra el conjunto de clases requeridas para responder a una entrada o evento del sistema. Cada hebra se integra y prueba de manera individual.
2. **Prueba basada en uso:** se comienza probando las **clases independientes**, esto es, aquellas que usan muy pocas clases de servidor o ninguna. Después de probar estas, se examina la siguiente capa de clases, llamadas **clases dependientes**, que son utilizadas por las primeras. Esta secuencia de pruebas para las capas de clases dependientes continúa hasta que se construye todo el sistema.

## 3.2.3. Prueba del sistema

Como indican Piattini et al. (2007), este tipo de prueba consiste en:

«el proceso de prueba de un sistema integrado de hardware y software para comprobar si cumple con los requisitos especificados, los cuales son:

- Cumplimiento de todos los requisitos funcionales, considerando el producto software final al completo, en un entorno de sistema.
- El funcionamiento y rendimiento de las interfaces hardware, software, de usuario y de operador.
- Adecuación de la documentación de usuario.
- Ejecución y rendimiento en condiciones límite y de sobrecarga».

Es posible distinguir varios tipos de pruebas del sistema:

- **Pruebas de recuperación:** se fuerza al software a fallar y se verifica que la recuperación se realice de manera adecuada. Este tipo de prueba debe aplicarse a aquellos sistemas que deben ser tolerantes a fallos y en los que la recuperación debe realizarse de manera rápida.
- **Pruebas de seguridad:** para algunos sistemas, por su naturaleza, es importante protegerlos de acciones de personas que deseen realizar daños o que pretendan obtener información confidencial. Las pruebas de seguridad intentan verificar que los mecanismos de protección del sistema lo protegerán contra accesos ilegales. Se trata de que la persona que lleva a cabo la prueba «ponga en jaque» el sistema, intentando conseguir contraseñas, provocando errores del sistema, etc. El objetivo es comprobar el comportamiento del sistema antes estas situaciones y determinar en qué medida la dificultad que supone el intento de penetración en el sistema no compensa los beneficios que pueda conseguir la persona que lo intenta penetrar.

- **Pruebas de esfuerzo:** en este tipo de prueba se trata de exponer al sistema a situaciones extremas para comprobar su comportamiento ante estas circunstancias, que, aunque no sean las normales, pueden tener lugar en la vida real. Alguna de las situaciones que se pueden plantear para realizar este tipo de pruebas son: ejecutar casos de prueba que requieran memoria máxima y otros recursos, que provoquen una búsqueda excesiva de datos residentes en disco, etcétera.
- **Pruebas de rendimiento:** se realizan con los sistemas para los que no es suficiente el cumplimiento de una serie de requisitos funcionales, sino que también es necesario que se cumplan ciertos requisitos de rendimiento, como que se dé respuesta a un determinado evento en un tiempo limitado. Por lo tanto, están diseñadas para probar el rendimiento del sistema.
- **Pruebas de despliegue:** también llamadas *pruebas de configuración*, estas se realizan cuando es necesario que la aplicación se ejecute en diversas plataformas y sistemas operativos. Consisten en probar el funcionamiento del sistema en las diferentes plataformas en las que se debe poder utilizar. Por ejemplo, en el caso del desarrollo de una aplicación web, puede ser necesario probar su funcionamiento en diferentes navegadores y en distintos sistemas operativos, incluyendo todas las posibles combinaciones de navegador y sistema operativo.

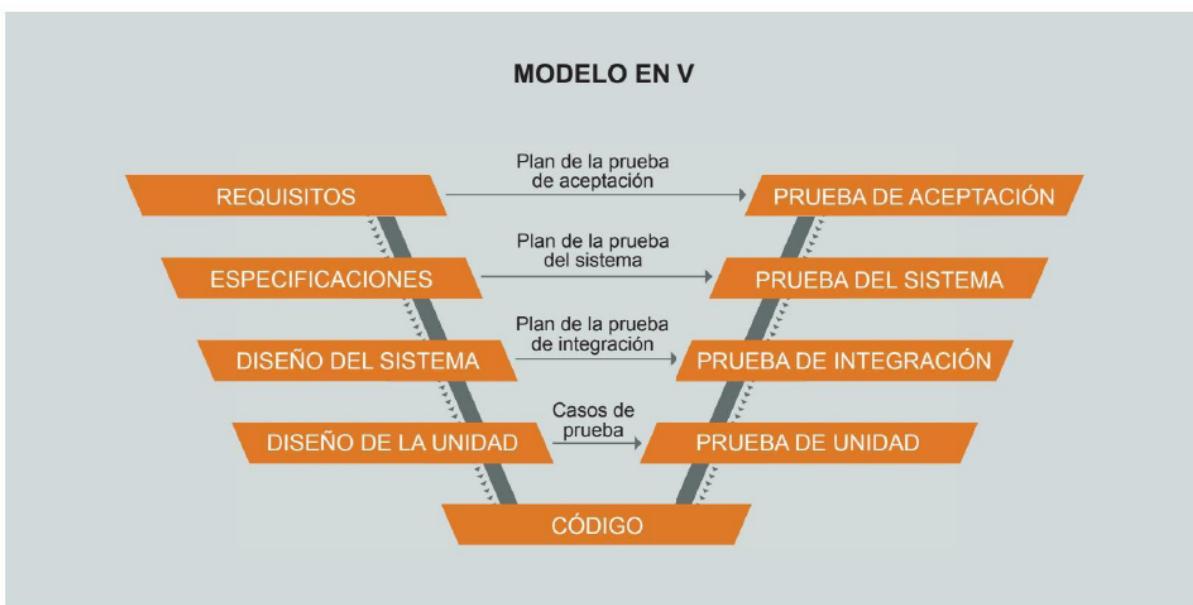
### 3.2.4. Prueba de validación

El objetivo de la prueba de validación, también llamada *prueba de aceptación*, es determinar si el software es considerado válido por parte la persona usuaria y si está preparado para su implantación en el entorno de las personas que lo vayan a usar.

La consideración de la validez del software se debe basar en una serie de criterios de validación o aceptación previamente establecidos, que deben formar parte de la especificación de requisitos del software. Por ello, es muy importante que estos criterios de validación aparezcan de manera explícita en la ERS (especificación de requisitos del software). Se ha de tener en cuenta que, según Piattini et al. (2007), la calidad del software se puede definir como «la concordancia del software producido con los requisitos explícitamente establecidos, con los estándares de desarrollo expresamente fijados y con los requisitos implícitos no establecidos formalmente, que desea el usuario». Es deseable que no existan requisitos implícitos, es decir, no incluidos en la ERS, porque ello podría originar problemas a la hora de considerar el software preparado para su entrega al cliente.

En este tipo de pruebas, participan activamente aquellas personas que lo van a utilizar, las cuales, ayudadas por el equipo de pruebas, deben ejecutar las pruebas. Esta es la fase final del proceso software, en la que la persona usuaria da su consentimiento para su uso o explotación.

Todos estos tipos de pruebas se muestran en el modelo en V, que es una variante del modelo en cascada. En su representación, los diferentes niveles de pruebas se colocan formando una V (véase la Figura 3.1) para mostrar la relación de cada fase del ciclo de vida con su fase de pruebas asociada.



**Figura 3.1.** Representación del modelo en V, en el que la fase posterior a la programación (las pruebas) se divide en varias subfases que corresponden a las etapas del ciclo de vida en cascada, si bien las etapas de análisis y diseño se dividen cada una de ellas en dos.

## 3.3. Técnicas de diseño de casos de prueba

Las técnicas de diseño de casos de prueba son las diferentes formas en que se pueden generar casos de prueba para probar el software. Dependiendo del enfoque que se adopte, se puede aplicar uno de los dos siguientes tipos de técnicas de diseño de casos de prueba (véase la Figura 1.10 de la Unidad 1):

1. **Pruebas de caja blanca o pruebas estructurales:** examinan los detalles de cada módulo, para lo que se debe disponer del código fuente. A través de dicho código, se prueban los diferentes caminos, los bucles, las variables, etcétera.
2. **Pruebas de caja negra o pruebas funcionales:** se considera al software como una caja negra que recibe una serie de entradas y proporciona una serie de salidas. El objetivo de estas pruebas es validar los requisitos funcionales.

No obstante, lo habitual es emplear los dos tipos de técnicas de diseño de casos de prueba, pues se pueden combinar y complementar perfectamente.

### Argot técnico



El nombre de estos dos tipos de pruebas está relacionado con la percepción del probador acerca de la porción del software que prueba.

En el caso de las **pruebas de caja negra**, para la persona que prueba el software, este se puede considerar una caja negra porque no es necesario conocer ningún detalle acerca de este más que la función que realiza y, por tanto, la salida que genera a partir de la entrada proporcionada.

Sin embargo, en el caso de las **pruebas de caja blanca**, se parte del código fuente y, de su examen, se derivan los casos de prueba. En este caso, sí es necesario, por consiguiente, conocer el interior (el código fuente) del software que se somete a prueba. De hecho, las pruebas de caja blanca también reciben el nombre de **pruebas de caja de cristal** porque el interior del software debe ser visible para el probador.

A continuación, se presentan las distintas técnicas de diseño de casos de prueba y, por último, se propone una estrategia para aplicar estas técnicas.

### ■ ■ ■ 3.3.1. Pruebas estructurales o de caja blanca

El objetivo de las pruebas de caja blanca es crear casos de prueba que permitan revisar detalles internos del software. Así, de estas pruebas se derivan casos de prueba que:

1. Garantizan que todas las rutas independientes dentro de un módulo se revisaron al menos una vez.
2. Revisan todas las decisiones lógicas en sus lados verdadero y falso.
3. Ejecutan todos los bucles en sus fronteras y dentro de sus fronteras operativas.
4. Revisan estructuras de datos internas para garantizar su validez.

Existen diferentes técnicas de diseño de casos de prueba de caja blanca, que se exponen seguidamente.

#### ■ ■ ■ Prueba del camino básico

Esta técnica también recibe el nombre de **prueba de ruta básica** y fue propuesta por Thomas J. McCabe. Esta técnica permite generar una métrica de la complejidad del módulo probado llamada **complejidad ciclomática de McCabe**, que es un número entero que indica el número de caminos independientes que hay en el código y, en consecuencia, el número de casos de prueba que hay que ejecutar para garantizar que se recorren todas las sentencias del código al menos una vez. Para calcular esta métrica y, en consecuencia, generar los casos de prueba, es necesario representar el código fuente que se va a probar en un **grafo de flujo**.

Para crear un grafo de flujo, se deben llevar a cabo los siguientes pasos:

1. Se asigna un número único a cada sentencia del código y a cada condición. No obstante, si se dispone de varias instrucciones en secuencia, es posible asignar un número único a todas ellas y estas serán tratadas como un bloque. Cada uno estos elementos (sentencia o grupo de sentencias en secuencia o condición) constituirá un **nodo** del grafo de flujo, que se representará mediante un círculo en cuyo interior aparecerá el número asignado.
2. Se unen los nodos mediante flechas llamadas **aristas** que representan el flujo de control y son similares a las flechas de un diagrama de flujo. Se deben representar las estructuras básicas de programación tal y como se muestra en Figura 3.2.

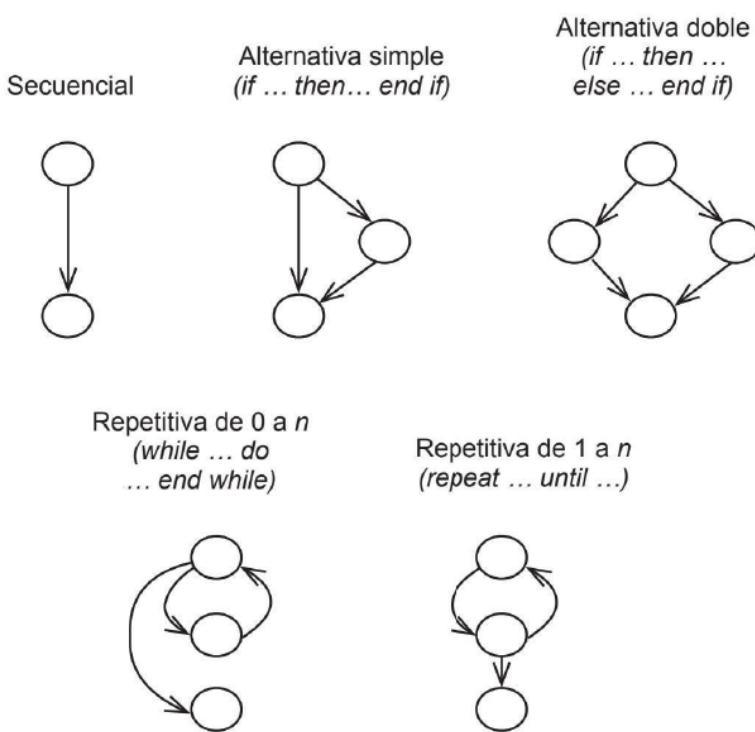


Figura 3.2. Representación de los grafos de flujo para diferentes estructuras de control.

Una vez creado el grafo de flujo, se puede calcular la complejidad ciclomática de McCabe  $V(G)$  de cualquiera de las tres formas siguientes:

- Número de regiones del grafo, considerando una región como una zona del grafo cerrada. No obstante, también se debe considerar como región la región externa (no cerrada).
- De acuerdo con la fórmula:

$$V(G) = A - N + 2$$

donde  $A$  es el número de aristas del grafo, y  $N$ , el número de nodos.

- Segundo la ecuación:

$$V(G) = NP + 1$$

donde  $NP$  es el número de nodos predicados del grafo de flujo, es decir, el número de nodos que contienen condiciones.

El valor  $V(G)$  indica el número de caminos independientes del grafo y, por tanto, el número de casos de prueba que hay que generar para garantizar que se han recorrido todas las instrucciones y condiciones del código.

El último paso de este método es generar casos de prueba que recorran cada uno de los caminos, para lo que se deben elegir los datos de entrada que fuercen cada uno de esos caminos. Puede ocurrir que alguno de los caminos no pueda recorrerse de manera independiente, sino como parte de otro camino.

Para cada caso de prueba, se deben comparar los resultados obtenidos con los esperados. La no coincidencia indicará la existencia de algún defecto y, en consecuencia, se debe llevar a cabo un proceso de depuración.

## Actividad resuelta 3.1

### Prueba del camino básico

En este ejercicio se aplica la prueba del camino básico a un método que lee una serie de números hasta que se introduce un cero y muestra la media de los números positivos introducidos y la media de los números negativos.

### Solución

La primera tarea que se debe a llevar a cabo es asignar un número entero a cada secuencia de instrucciones y a cada condición:

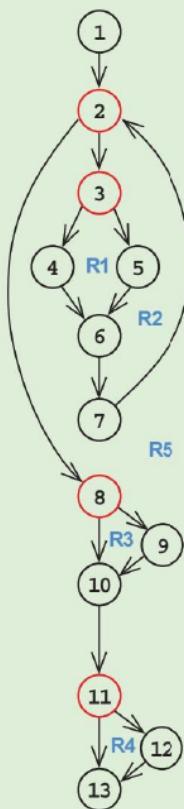
```

begin
    int num = 0, cont_pos=0, cont_neg=0, suma_pos=0, suma_neg=0;
    float media_pos=0,media_neg=0;
    System.out.print("Introduce número: ");
    num=Entrada.entero();

    while ((num != 0))
        if ((num > 0))
            cont_pos++;
            suma_pos+=num;
        else
            cont_neg++;
            suma_neg+=num;
        endif;
        System.out.print("Introduce número: ");
        num=Entrada.entero();
    endwhile;
    if ((cont_pos != 0))
        media_pos = (float)suma_pos/cont_pos;
        System.out.println("Media de los positivos: "+ media_pos);
    endif;
    if ((cont_neg !=0))
        media_neg= (float)suma_neg/cont_neg;
        System.out.println("Media de los negativos: "+ media_neg);
    endif;
end;
```

Es posible incluir dentro un mismo nodo varias instrucciones en secuencia, como es el caso de las instrucciones incluidas dentro del nodo 1. También se puede incluir, dentro del mismo nodo, la instrucción que indica el fin de una instrucción condicional y la instrucción o las instrucciones siguientes en secuencia, como se ha hecho para el nodo 6. Así mismo, es posible incluir, dentro de un mismo nodo, la instrucción que indica el fin de una estructura repetitiva junto con las instrucciones anteriores dentro del bucle si son instrucciones en secuencia. Por este motivo, aunque no se ha hecho, también se podrían haber incluido en el mismo nodo las instrucciones asignadas a los nodos 6 y 7.

El siguiente paso es construir el grafo de flujo de acuerdo con las estructuras mostradas en la Figura 3.2. El grafo de flujo quedaría como el que se muestra en la Figura 3.3. En el grafo, se han marcado en rojo los nodos predicados y se ha asignado un número a cada región en color azul.



**Figura 3.3.** Grafo de flujo correspondiente al código de esta actividad.

Una vez que se tiene el grafo de flujo, ya se puede calcular la complejidad ciclomática,  $V(G)$ , de las tres formas que se ha indicado antes:

$$V(G) = \text{número de regiones} = 5$$

$$V(G) = A - N + 2 = 16 - 13 + 2 = 5$$

$$V(G) = NP + 1 = 4 + 1 = 5$$

Como  $V(G)$  toma el valor 5, se deduce que hay cinco caminos independientes en el código examinado y, por tanto, el número de casos de prueba que hay que crear para cubrir todas las instrucciones y condiciones del programa es 5. La siguiente tarea es encontrar esos cinco caminos independientes. Para ello, hay que tener en cuenta que cada camino independiente incluye una ruta nueva en relación con los anteriores:

Camino 1: 1-2-8-10-11-13

Camino 2: 1-2-3-4-6-7-2-8-10-11-13

Camino 3: 1-2-3-4-6-7-2-8-9-10-11-13

Camino 4: 1-2-3-5-6-7-2-8-10-11-13

Camino 5: 1-2-3-5-6-7-8-10-11-12-13

Con objeto de probar el camino 1, se debe introducir como primer número el 0 con el fin de que no se entre en el bucle ni una sola vez y se pase directamente del nodo 2 al nodo 8. En este caso, el programa no debe mostrar nada, ya que no se ha introducido ningún número ni positivo, ni negativo.

El camino 2 no se puede probar como tal, ya que al introducir un número positivo (para pasar por el nodo 4), se debería pasar por el nodo 9, que muestra la media de los números positivos. Por consiguiente, este camino debe probarse como parte de otro.

Para probar el camino 3, hay que suministrar al programa, en primer lugar, un número positivo (para pasar por el nodo 4), por ejemplo, el número 8, y un 0 a continuación (para no pasar más por el bucle). El programa deberá indicar que la media de los números positivos es 8. Al probar el camino 3, ya se prueba el camino 2 como parte de él.

El camino 4 tampoco se puede probar como tal, pues al introducir un número negativo (para pasar por el nodo 5), se debe pasar por el nodo 12 para mostrar la media de los números negativos.

Por último, a fin de probar el camino 5, se debe suministrar al programa un número negativo (para pasar por el nodo 5), por ejemplo, el -4, y un 0 a continuación. El programa deberá indicar que la media de los números negativos es -4. Al probar el camino 5, se prueba también el camino 4 como parte de él.

Para cada caso de prueba, se comprueba que el resultado mostrado coincide con el esperado, de forma que la no coincidencia indicará la existencia de algún defecto, por lo que habrá que proceder a detectar y corregir dicho error (mediante un proceso de depuración).

La complejidad ciclomática de McCabe es una métrica de la complejidad del módulo probado. Así, cuando  $V(G)$  es mayor que 10, la probabilidad de defectos en el módulo o en el programa crece bastante si dicho valor alto no se debe a sentencias case o similares. En estos casos, es recomendable replantearse el diseño modular obtenido, de modo que sería conveniente dividir el módulo complejo en varios módulos para reducir la complejidad del software.

## Actividad propuesta 3.1

### Prueba del camino básico

El siguiente programa escrito en Java pide un número entero por teclado e indica si es un número perfecto, es decir, si es igual a la suma de sus divisores propios. Un divisor propio es un entero positivo distinto del número en sí mismo y que divide al número de forma exacta, es decir, sin resto.

Construye el grafo de flujo para este programa y calcula su complejidad ciclomática. Además, indica un conjunto de caminos independientes y, para cada camino, el caso de prueba correspondiente, incluyendo la entrada proporcionada y la salida esperada.

```

1 public static void main(String[] args) {
2     int num, divisor, sumadivisores;
3     divisor = 1;
4     sumadivisores = 0;
5     Scanner entrada = new Scanner(System.in);
6     System.out.print ("Introduzca un número mayor que 0: ");
7     num = entrada.nextInt();
8     while (divisor <= num/2)
9     {
10         if (num % divisor == 0)
11             sumadivisores = sumadivisores + divisor;
12         divisor++;
13     }
14     if (num == sumadivisores)
15         System.out.println ("El número " + num + " es un número
16         perfecto");
17     else
18         System.out.println ("El número " + num + " no es un número
19         perfecto");
20 }
```

## ■■■ Prueba de bucles

Los bucles o estructuras repetitivas constituyen una estructura de control fundamental en todo lenguaje de programación. La prueba de bucles se centra exclusivamente en este tipo de estructuras. Los casos de prueba que se deben generar para cada tipo de bucle son los siguientes:

■ **Bucles simples:** si el número máximo de iteraciones por el bucle es  $n$ , se deben probar los siguientes casos:

- Saltarse completamente el bucle.
- Realizar una iteración por el bucle.
- Realizar dos iteraciones por el bucle.
- Realizar  $m$  iteraciones por el bucle, siendo  $m < n$ .
- Realizar  $n-1$ ,  $n$  y  $n+1$  iteraciones por el bucle.

■ **Bucles anidados:** se deben llevar a cabo los siguientes pasos:

1. Comenzar por el bucle más interno, estableciendo valores mínimos para todos los demás bucles.
2. Realizar pruebas de bucle simple para el bucle más interno, manteniendo los bucles exteriores en sus valores mínimos.

3. Trabajar de fuera hacia dentro y probar el siguiente bucle manteniendo los otros bucles exteriores en valores mínimos y los otros bucles interiores en valores típicos.

4. Continuar hasta que se hayan probado todos los bucles.

- **Bucles concatenados:** si los bucles son independientes unos de otros, se pueden probar empleando la estrategia indicada para los bucles simples. Si los bucles concatenados son dependientes porque el segundo bucle usa el contador del primer bucle como valor inicial para el segundo, se recomienda emplear el enfoque aplicado a los bucles anidados.

## Prueba de flujo de datos

Esta técnica de diseño de casos de prueba selecciona caminos de un programa de acuerdo con las ubicaciones de las definiciones y usos de variables en dicho programa.

Aquí es necesario considerar que, en una instrucción, se define una variable si se le asigna valor, mientras que se puede afirmar que esa variable se usa si se consulta o utiliza su valor de algún modo. Así, en una condición, siempre se usan una o varias variables.

Si se asigna el número  $I$  a una instrucción, el conjunto  $DEF$  para esa instrucción  $I$  estará formado por el conjunto de variables que se definen en esa instrucción. Su conjunto  $USO$  estará formado por el conjunto de variables cuyo valor se usa en esa instrucción. Esto quedaría ilustrado así:

$$DEF(I) = \{X \mid I \text{ contiene una definición de } X\}$$

$$USO(I) = \{X \mid I \text{ contiene un uso de } X\}$$

La definición de una variable  $X$  en la instrucción  $I$  está viva en la instrucción  $I'$  si hay un camino entre las instrucciones  $I$  e  $I'$  que no contiene ninguna otra definición de  $X$ .

Mediante esta técnica, se encuentran las **cadenas DU** para una variable. Estas cadenas tienen la forma  $[X, I, I']$ , donde  $X$  es una variable,  $I$  e  $I'$  son instrucciones tal que  $X$  está en  $DEF(I)$  y en  $USO(I')$ , y la definición de  $X$  en  $I$  está viva en  $I'$  porque existe algún camino entre  $I$  e  $I'$  que no incluye ninguna otra definición de  $X$ .

### Argot técnico



Las **cadenas DU** de esta técnica reciben este nombre porque se trata de **cadenas de definición-uso**, más concretamente, cadenas en las que primero se define una variable y luego se usa esa misma variable sin que haya entre la instrucción donde se define y la instrucción donde se usa ninguna otra definición de la variable.

Debido a que el objetivo es encontrar cadenas DU, esta técnica también recibe el nombre de **prueba DU**.

Esta técnica requiere llevar a cabo los siguientes pasos:

1. Asignar un número a cada instrucción y condición de un programa.

2. Calcular el conjunto *DEF* y *USO* para cada variable que se usa en ese programa.
3. Encontrar las cadenas *DU* para todas las variables.
4. Generar el número mínimo de casos de prueba que incluyan caminos para todas las cadenas *DU*.
5. Generar datos de entrada que fuercen cada uno de los caminos encontrados en el paso anterior.

## Actividad resuelta 3.2

### Prueba de flujo de datos

En este ejercicio se aplica la prueba de flujo de datos al método de la Actividad resuelta 3.1 para el que se obtuvo el grafo de flujo de la Figura 3.3.

### Solución

El primer paso ya está hecho porque ya se ha asignado un número a cada instrucción y condición.

Seguidamente, se calculan las cadenas *DU* solo para la variable *num*. A tal fin, se definen los conjuntos *DEF* y *USO* para esta variable. Como se puede observar, la variable *num* se define en las instrucciones 1 y 6, mientras que se usa en las que tienen asignados los números 2, 3, 4 y 5. Esto se indica del siguiente modo:

$$\begin{aligned} \text{DEF}(1) &= \{\text{num}\} \\ \text{USO}(2) &= \{\text{num}\} \\ \text{USO}(3) &= \{\text{num}\} \\ \text{USO}(4) &= \{\text{num}\} \\ \text{USO}(5) &= \{\text{num}\} \\ \text{DEF}(6) &= \{\text{num}\} \end{aligned}$$

El siguiente paso es buscar las cadenas *DU* para esta variable, que son las siguientes numeradas:

$$\begin{aligned} \text{DU } [\text{num}, 1, 2] &(1) \\ \text{DU } [\text{num}, 1, 3] &(2) \\ \text{DU } [\text{num}, 1, 4] &(3) \\ \text{DU } [\text{num}, 1, 5] &(4) \\ \text{DU } [\text{num}, 6, 2] &(5) \\ \text{DU } [\text{num}, 6, 3] &(6) \\ \text{DU } [\text{num}, 6, 4] &(7) \\ \text{DU } [\text{num}, 6, 5] &(8) \end{aligned}$$

A continuación, se debe encontrar el número mínimo de caminos que incluyan todas las cadenas *DU*. Estos caminos son:

- Camino 1: 1-2-3-4-6-7-2-3-4-6-7-2-3-5-6-7-2-8-9-10-11-12-13. Este camino cubre las cadenas DU: (1), (2), (3), (5), (6), (7) y (8).
- Camino 2: 1-2-3-5-6-7-2-8-10-11-12-13. Este camino cubre la cadena DU (4).

En este caso, con dos caminos es suficiente para cubrir todas las cadenas DU encontradas. En la primera iteración del bucle del camino 1, se cubren las cadenas DU (1), (2) y (3); en la segunda iteración, se cubren las cadenas DU (5), (6) y (7) y, en la tercera iteración, se cubre la cadena DU (8). Pero, en el camino 1, no es posible cubrir la cadena DU con el número (4), es decir, DU [num, 1, 5] porque por el nodo 1 solo se pasa una vez. En las iteraciones, no se puede pasar por los nodos 4 y 5 a la vez y, para volver a uno de los nodos 4 o 5, hay que pasar por el nodo 6, que contiene otra definición de la variable num. Por este motivo, hay que establecer otro camino para cubrir esta cadena DU.

Ya solo quedaría generar los datos de entrada para recorrer cada uno de estos dos caminos. Para el camino 1, habría que introducir en el programa dos números positivos, luego uno negativo y un cero para finalizar. Por ejemplo, si se suministra un 4, un 8 y un -9, el programa nos debe indicar que la media de los números positivos es 6 y la de los negativos, -9. Para forzar el camino 2, habría que introducir en el programa un número negativo y un cero para finalizar. Si, por ejemplo, se proporciona como entrada el número -10 y luego un 0, el programa debe indicar que la media de los números negativos es -10.

## Actividad propuesta 3.2

### Prueba de flujo de datos

Tomando el programa de la Actividad propuesta 3.1, obtén las cadenas DU para la variable *sumadivisores* y encuentra el número mínimo de caminos de prueba que recorren todas estas cadenas.

### 3.3.2. Pruebas funcionales o de caja negra

El objetivo de las pruebas de caja negra es crear casos de prueba que permitan revisar todos los requisitos funcionales de un programa. A tal efecto, existen diferentes técnicas de diseño de casos de prueba de caja negra que se exponen a continuación.

#### Particiones o clases de equivalencia

Por medio de esta técnica, se identifican un conjunto de clases de equivalencia o tipos de valores que se deben asignar a los datos de entrada. El primer paso para identificar las clases de equivalencia es determinar las condiciones de entrada del programa. Una condición de entrada puede ser:

- Un valor específico.
- Un rango de valores permitido, por ejemplo, entre 18 y 65.
- Un conjunto de valores permitidos, por ejemplo, 18, 36 y 72.

- Una condición lógica o booleana, por ejemplo, es español o no es español.

Hay una serie de reglas que ayudan a identificar las clases de equivalencia en función del tipo de la condición de entrada:

- Si la condición de entrada es un valor específico, se debe crear una clase válida (con ese valor) y dos no válidas (con dos valores distintos).
- Si la condición de entrada consiste en un rango de valores permitidos (por ejemplo, un número entre 18 y 65), se debe crear una clase válida ( $18 \leq \text{número} \leq 65$ ) y dos no válidas ( $\text{número} < 18$  y  $\text{número} > 65$ ).
- Si la condición de entrada consiste en un conjunto de valores permitidos y se sabe que el programa trata de manera diferente cada uno de ellos (por ejemplo, el idioma es inglés, francés o italiano), se identifica una clase válida por cada valor (en este caso, tres valores) y una clase no válida (por ejemplo, alemán).
- Si la condición de entrada consiste en una condición lógica (por ejemplo, la persona debe tener nacionalidad española), se identifica una clase de equivalencia válida (es española) y otra no válida (no es española).
- Si se sabe que algunos elementos de una clase no se tratan de igual forma que el resto de sus elementos, debe dividirse dicha clase de equivalencia en varias clases menores.

Para la creación de los casos de prueba, una vez conocidas estas reglas, se llevan a cabo los siguientes tres pasos:

1. Asignar un número único a cada clase de equivalencia.
2. Crear el número mínimo de casos de prueba que incluyan todas las clases válidas.
3. Crear un caso de prueba por cada clase no válida. El caso de prueba debe incluir una clase no válida, y todas las demás clases deben ser válidas.

## Actividad resuelta 3.3

### Prueba de particiones o clases de equivalencia

En una aplicación, la persona usuaria debe introducir su edad, que debe ser un número entre 18 y 65; su NIF, que debe constar de 8 números y una letra; y, además, debe indicar si tiene o no nacionalidad española, de forma que tener nacionalidad española es un requisito.

#### Solución

En la Tabla 3.1, se muestran las condiciones de entrada y, por cada una de ellas, las clases válidas y las no válidas, numerándose cada una de ellas.

**Tabla 3.1.** Condiciones de entrada para el supuesto, junto con las clases de equivalencia válidas y no válidas de acuerdo con las normas especificadas

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)

En la Tabla 3.2, se muestran los casos de prueba válidos y, en la Tabla 3.3, los no válidos. Por cada caso de prueba, se indica, en la última columna, las clases de equivalencia incluidas. Recuerda que, para el caso de las clases válidas, se debe crear el número mínimo de casos de prueba que incluyan todas las clases válidas; en este caso, se requiere un único caso de prueba.

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

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

Para las clases no válidas, hay que crear un caso de prueba por cada una de ellas. Como se puede observar en la Tabla 3.1, al haber 8 clases de equivalencia no válidas, se deben crear 8 casos de prueba (Tabla 3.3).

**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)

## Actividad propuesta 3.3

### Prueba de particiones o clases de equivalencia

La consejería de sanidad de una región desea crear una aplicación para posibilitar la solicitud de citas previas de los pacientes con sus médicos. Los datos que deben introducir los pacientes para acceder a esta aplicación son:

- a) Número de tarjeta sanitaria (TIS): debe ser un número entero de 8 dígitos.
- b) Primer apellido: debe ser una cadena de entre 2 y 30 letras, pudiendo incluir algún espacio en blanco.
- c) Año de nacimiento: debe ser un número entero entre 1901 y el año actual.

Crea una tabla de clases de equivalencia. Además, genera los casos de prueba correspondientes usando la técnica de particiones o clases equivalencia, indicando en cada caso las clases cubiertas.

## Análisis de valores límite

Se trata de una técnica de diseño de casos de prueba complementaria a la de clases de equivalencia. Las diferencias que presenta con relación a esta son:

- En vez de seleccionar cualquier elemento dentro de una clase de equivalencia, se selecciona aquel o aquellos que se hallan justo en los límites de la clase.
- En vez de centrarse únicamente en las condiciones de entrada, en esta técnica, los casos de prueba se generan teniendo en cuenta también el dominio de salida.

Las reglas que se siguen para generar los casos de prueba son las siguientes:

1. Si una condición de entrada especifica un rango de valores, por ejemplo, entre -5 y 10, se deben generar casos válidos para los extremos del rango (-5 y 10) y casos no válidos justo más allá de los extremos (-5,01 y 10,01, en caso de que se admitan dos decimales).
2. Si una condición de entrada especifica un número de valores (entre 1 y 100), hay que generar casos de prueba para los valores mínimo y máximo (1 y 100), uno menos que el mínimo (0) y uno más que el máximo (101).
3. Se emplea la regla 1 para cada condición de salida que especifique un rango de valores. Si, por ejemplo, el salario mínimo es 1.200,00 € y el máximo es 3.600,00 €, se deben generar casos de prueba que den como resultado salarios de 1.200,00 €, 3.600,00 €, 1.199,99 € y 3.600,01 €.
4. Se usa la regla 2 para cada condición de salida que especifique un número de valores. Si, por ejemplo, el programa especifica que se puede hacer uso de uno a tres descuentos, habrá que generar casos de prueba para lograr cero, uno, tres y cuatro descuentos.
5. Si la entrada o salida de un programa es un conjunto ordenado (por ejemplo, una tabla o un archivo secuencial), los casos de prueba se deben centrar en el primer y último elemento.

## Actividad resuelta 3.4

### Prueba de análisis de valores límite

Partiendo de la condición de entrada *edad* de la Actividad resuelta 3.3, indica qué clases de equivalencia adicionales habría que crear, y genera los casos de prueba correspondientes empleando la técnica de análisis de valores límite.

#### Solución

Dado que esta condición de entrada especifica un número de valores (entre 18 y 65), hay que generar clases de equivalencia para los valores mínimo y máximo (18 y 65), uno menos que el mínimo (17) y uno más que el máximo (66), como se muestra en la Tabla 3.4, en la que se ha asignado a las clases de equivalencia una numeración que sigue la de la Actividad resuelta 3.3:

**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)

Una vez generada estas clases de equivalencia adicionales, se trata de generar los casos de prueba que abarquen estas clases. Para ello, se debe tener en cuenta que hay que generar un caso de prueba válido que abarque todas las clases válidas. Como, en este caso, para la edad, hay dos clases válidas, se deben generar dos casos de prueba válidos (véase Tabla 3.5). A la edad, se deben añadir las otras condiciones de entrada que se incluyeron en la Actividad resuelta 3.3.

**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)

Hay que generar, además, un caso de prueba nuevo por cada clase no válida. Como se han añadido dos nuevas clases no válidas, se generará un caso de prueba no válido por cada una de ellas (Tabla 3.6).

**Tabla 3.6.** Casos de prueba con clases de equivalencia no válidas que se generan empleando la técnica de análisis de valores límite

Edad	NIF	Nacionalidad	Clases incluidas
17	32323267G	Española	(14), (5), (10)
66	32323267G	Española	(15), (5), (10)

## Conjetura de errores

Esta técnica consiste en generar una lista de errores que suelen cometer quienes desarrollan aplicaciones y de las situaciones que suelen dar lugar a dichos errores, para luego generar un conjunto de casos de prueba basándose en esta lista. Se trata de errores que se cometen comúnmente y no relacionados con aspectos funcionales.

Se muestran a continuación una serie de ejemplos típicos que reflejan el empleo de esta técnica:

- El valor 0 tanto en la entrada como en la salida es una situación propensa a que se cometan errores.
- En situaciones en las que se introduce un número variable de valores, como por ejemplo una lista, conviene considerar los siguientes casos: no introducir ningún valor, introducir un solo valor y un conjunto de valores en el que todos son iguales.
- La usuaria o usuario de la aplicación puede introducir datos erróneos, por ejemplo, que en un campo numérico introduzca algún carácter no numérico.
- La persona encargada de la programación pueda haber interpretado incorrectamente algo en la especificación.

### 3.3.3. Estrategia de aplicación de las técnicas de diseño de casos de prueba

El proceso de prueba de una aplicación completa se debe ejecutar de acuerdo con los siguientes pasos:

- Se comienza realizando la prueba de unidad de cada clase. Se puede comenzar con la técnica de análisis de valores límite, añadir clases válidas y/o no válidas no contempladas con la técnica de clases de equivalencia y finalizar con la técnica de conjectura de errores.
- Si no se ha alcanzado la cobertura deseada (por ejemplo, el recorrido de todos los posibles caminos a lo largo del código fuente), se deben añadir los casos necesarios mediante técnicas de pruebas de caja blanca (es decir, la prueba del camino básico, la de bucles y la de flujo de datos).
- Se continúa con las pruebas de integración, aunque también se pueden combinar estas con pruebas de unidad. Las pruebas de integración se deben centrar en la comprobación de los mecanismos de interacción entre las diferentes clases y se deben usar preferentemente técnicas de caja blanca. Se debe ir incrementando progresivamente el alcance de estas pruebas hasta abarcar el software completo.
- La prueba del sistema debe emplear casos de prueba de caja negra para comprobar el cumplimiento de los objetivos requeridos por el sistema.
- En la prueba de validación, deben tomar parte las personas usuarias finales y se pueden reutilizar casos de prueba del sistema. La finalización exitosa de estas pruebas supone la aceptación final del software por parte del usuario o usuaria.

**Recuerda**

Una buena estrategia de prueba debe combinar los dos tipos de técnicas de diseño de casos de prueba estudiadas (pruebas de caja blanca y de caja negra) para lograr la cobertura deseada.

## 3.4. Documentación de las pruebas

La documentación de las pruebas es conveniente para una buena organización de estas y para asegurar su reutilización. Se deben documentar tanto el diseño de las pruebas como el resultado o ejecución de las pruebas.

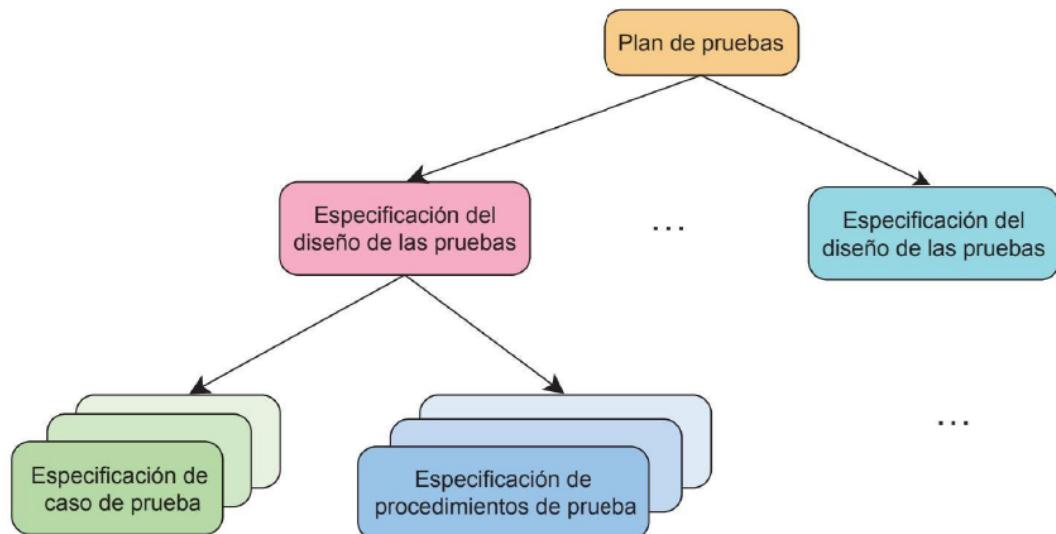
En cuanto a la **documentación del diseño de las pruebas**, es conveniente crear los siguientes documentos:

- **Plan de pruebas:** es un documento con la planificación general de las pruebas para la aplicación que se está creando. Se debe indicar el enfoque general de las pruebas, los recursos requeridos, las actividades de pruebas que se van a llevar a cabo, los elementos que se deben probar, el personal responsable y los riesgos asociados.
- **Especificación del diseño de las pruebas:** es un informe que detalla el plan de pruebas porque en él se indica cada uno de los elementos concretos que se van a probar y los procedimientos de prueba. A partir de este documento se generan los siguientes:
  - *Especificaciones de casos de prueba:* por cada prueba que se va a realizar, se deben indicar los datos de entrada que se van a suministrar, los resultados esperados y las dependencias entre casos de prueba.
  - *Especificaciones de procedimientos de prueba:* especifica los pasos para la ejecución de uno o de un conjunto de casos de prueba. En este documento, se deben indicar la lista de casos de prueba correspondientes al procedimiento, los requisitos para la ejecución de los casos de prueba (entorno, personal, etc.) y los pasos del procedimiento.

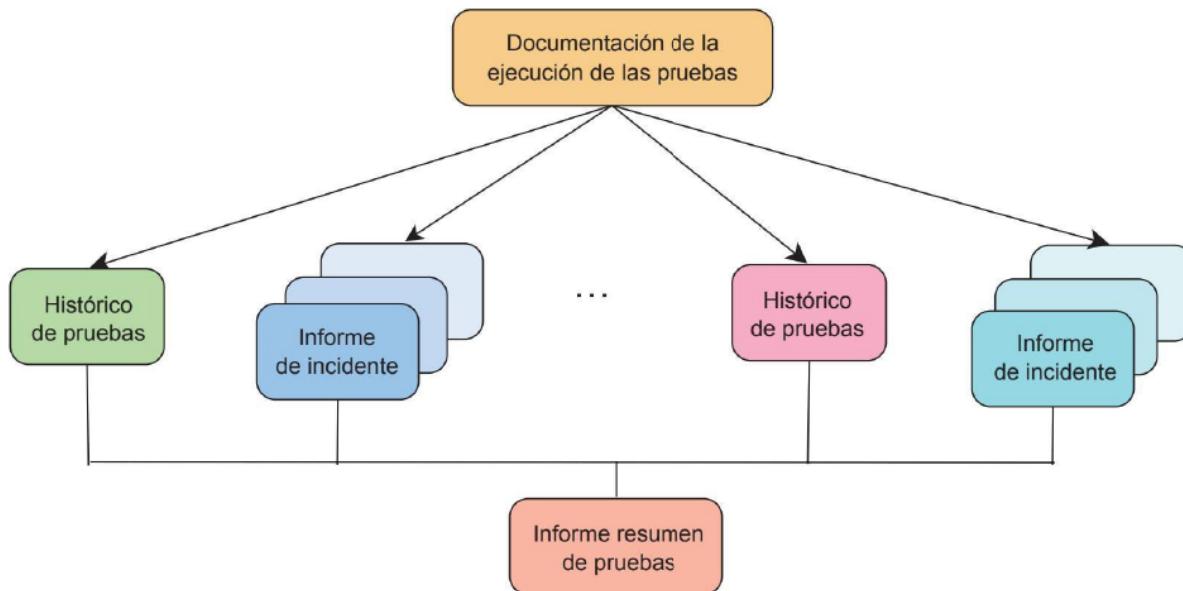
En cuanto a la **documentación de la ejecución de las pruebas**, esta es importante para la eficacia en la detección y corrección de defectos y para dejar constancia de los resultados de las pruebas. La ejecución de las pruebas toma como entrada las especificaciones de los casos de prueba que se van a usar y las especificaciones de los procedimientos de prueba, y debe generar dos tipos de documentos:

1. **Histórico de pruebas:** registra todos los hechos relevantes ocurridos durante la ejecución: elementos probados, entorno de pruebas, resultados obtenidos, fecha y hora y referencia al informe de incidentes, si es el caso.
2. **Informe de incidentes:** registra cada incidente ocurrido durante la prueba que requiera una posterior investigación.

La ejecución de las pruebas correspondiente a cada especificación del diseño de las pruebas genera asimismo un **informe resumen de pruebas**, que incluye: resumen de la evaluación de los elementos probados, variaciones del software como resultado de las pruebas, valoración de la cobertura de la prueba, resumen de los resultados obtenidos y resumen de las actividades de prueba (incluyendo el consumo de recursos).



**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.



**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.

La valoración de los procesos de prueba es importante porque puede ayudar en proyectos posteriores. Por ello, es conveniente realizar un **análisis causal**, cuyo objetivo es

proporcionar información sobre la naturaleza de los defectos encontrados. Por cada defecto encontrado, se debería registrar la siguiente información: cuándo se cometió, quién lo cometió, qué se hizo mal, cómo se podría haber evitado, por qué no se detectó antes, cómo se podría haber detectado antes y cómo se encontró el error.

El objetivo del análisis causal es formar al personal sobre los errores cometidos para que se puedan prevenir en el futuro. La información generada se puede emplear para predecir los fallos futuros del software.

## 3.5. Herramientas de depuración

La depuración es la tarea que hay que realizar tras una prueba exitosa, es decir, tras una prueba que detecta un fallo o, más bien, el síntoma de un defecto. En ese caso, lo que se hace es localizar el defecto en el software y corregirlo, proceso que se conoce con el nombre de **depuración**.

A veces, la localización del defecto es una tarea sencilla, pero otras veces, no tanto, y para ello, pueden resultar útiles las herramientas de depuración que proporcionan los entornos de desarrollo. Puede ocurrir incluso que para la detección del defecto sea necesario crear nuevos casos de prueba.

Una vez detectado el defecto, habrá que corregirlo y tener en cuenta que, al llevar a cabo las modificaciones que supuestamente corregirán el fallo, se pueden introducir nuevos defectos, por lo que será conveniente ejecutar pruebas de regresión, esto es, se repetirán los casos de prueba que se ejecutaron antes de las modificaciones para estar seguros de que los cambios no hayan originado nuevos defectos en el software.

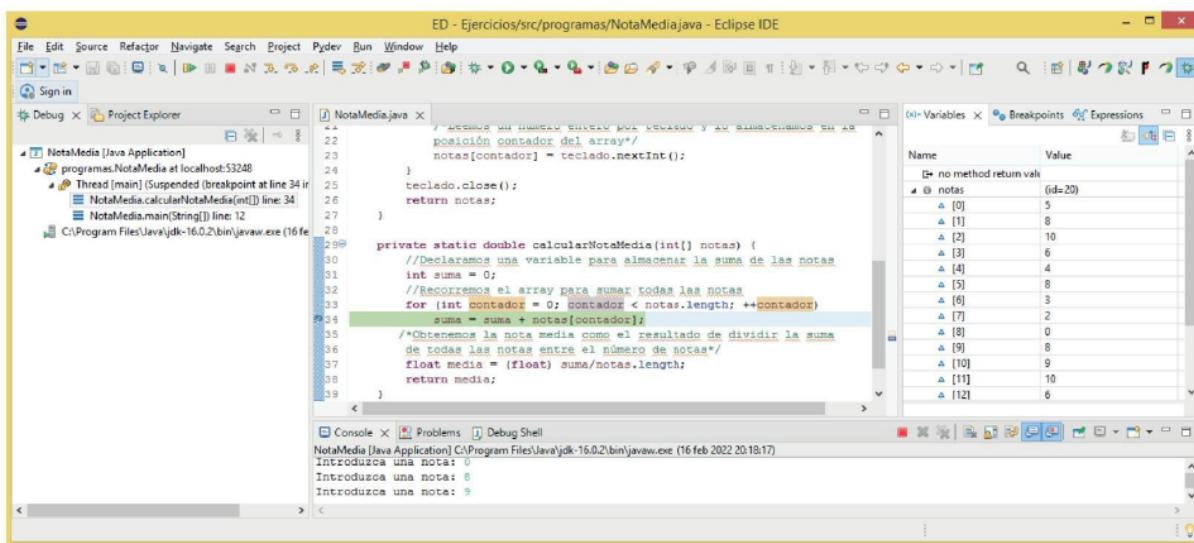
A modo de ejemplo, aquí se verá cómo el depurador o *debugger* del IDE Eclipse asiste en esta tarea a las personas que desarrollan software. *Grosso modo*, las ayudas que proporciona esta herramienta son las siguientes:

- Ejecuta el código paso a paso.
- Establece puntos de ruptura o interrupción para que el código se ejecute hasta esos puntos sin ejecutarse paso a paso.
- Suspende la ejecución del programa.
- Examina los valores de las variables a lo largo de la ejecución.

Para ejecutar un programa en modo depuración, se puede realizar cualquiera de las siguientes acciones:

- Activar la opción de menú *Run → Debug*.
- Seleccionar la clase que se desea ejecutar y, en el menú contextual, elegir la opción *Debug As → Java Application*.
- Hacer clic en el icono  y seleccionar la opción *Debug As → Java Application*.

Es conveniente también abrir la vista depuración seleccionando la opción de menú *Window → Perspective → Open Perspective → Debug*. Aparecerá entonces una pantalla como la que se muestra en la Figura 3.6.



**Figura 3.6.** Vista Depuración mientras se está ejecutando una aplicación en modo de depuración.

En esta pantalla, es posible distinguir las siguientes áreas:

- **Área de depuración o debug:** esta área se encuentra a la izquierda y en ella se muestran los hilos de ejecución del programa que se está ejecutando, en este caso, solo uno. Se indica la línea en la que está parada la ejecución del programa.
- **Área de edición:** situada en la parte central, en ella se muestra el código fuente del programa que se está ejecutando.
- **Área de consola:** esta se encuentra en la parte inferior, donde se muestra el resultado de la ejecución del programa.
- **Área de inspección:** se ubica a la derecha y dispone de varias pestañas. En la pestaña *Variables*, se muestran los valores de las variables a lo largo de la ejecución del programa. Estos valores se pueden modificar simplemente cambiando el valor en la columna *Value*. En la pestaña *Breakpoints*, se pueden ver los puntos de ruptura creados, que se pueden editar (eliminar, añadir nuevos puntos de ruptura, etc.). Por último, en la pestaña *Expressions*, se pueden escribir expresiones en la columna *Name* y ver su valor a lo largo de la ejecución del programa en la columna *Value*.

En modo depuración, al hacer clic en la opción de menú *Run*, aparecen nuevas opciones, algunas de las cuales se explican a continuación:

- **Resume** (o tecla F8): reanuda un hilo suspendido ejecutando instrucciones de este hasta llegar a un punto de ruptura.
- **Suspend**: suspende el hilo seleccionado.
- **Terminate** (o teclas Ctrl + F2): termina el proceso de depuración.
- **Step into** (o tecla F5): ejecuta el código paso a paso. En caso de que en el código haya una llamada a un método, también se ejecutarán paso a paso cada instrucción de este.
- **Step over** (o tecla F6): ejecuta el código paso a paso, pero en caso de que en el código haya una llamada a un método, el código del método no se ejecutará paso a paso.

- *Step return* (o tecla F7): si la ejecución está dentro de un método, el depurador sale del método en cuestión.
- *Run* (o teclas Ctrl + F11): ejecuta el programa normalmente.
- *Debug* (o tecla F11): ejecuta el programa en modo depuración.
- *Skip all breakpoints* (teclas Ctrl + Alt + B): se saltan todos los puntos de ruptura como si estos no existiesen.
- *Remove all breakpoints*: elimina todos los puntos de ruptura.
- *Add Java exception breakpoint*: permite añadir una excepción como punto de ruptura.

A continuación, se ofrece un ejemplo de uso del depurador a partir del código que se expone abajo, por medio del cual, se crea un *array* de 15 números que representan las notas (números enteros) obtenidas por 15 alumnos. Se usa el método *leerNotas* para leer por teclado las 15 notas e introducirlas en el *array* y el método *calcularNotaMedia* para leer todas las notas y calcular la nota media, que es mostrada por pantalla en el método *main*.

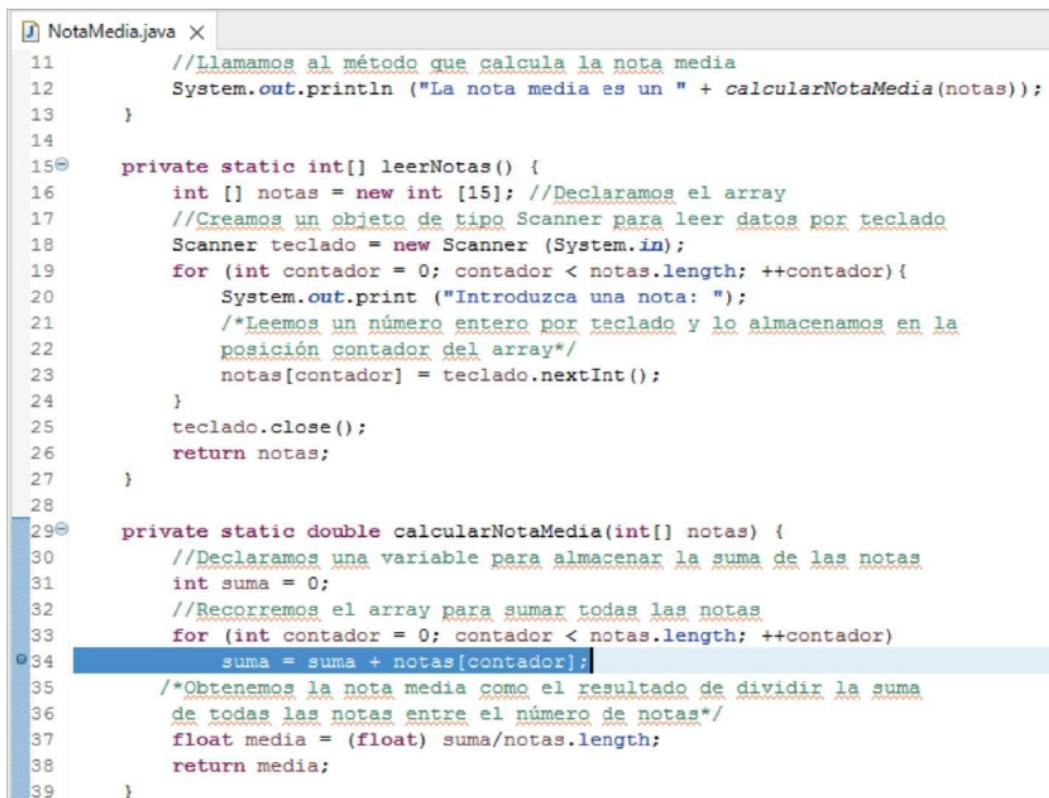
```
1 package programas;
2 import java.util.Scanner;
3 public class NotaMedia {
4     public static void main(String[] args) {
5         int [] notas = new int [15]; //Declaramos el array
6         //Llamamos al método leerNotas para leer las notas por teclado
7         notas = leerNotas();
8         //Llamamos al método que calcula la nota media
9         System.out.println ("La nota media es un " +
10            calcularNotaMedia(notas));
11    }
12    private static int[] leerNotas() {
13        int [] notas = new int [15]; //Declaramos el array
14        //Creamos un objeto de tipo Scanner para leer datos por
15        //teclado
16        Scanner teclado = new Scanner (System.in);
17        for (int contador = 0; contador < notas.length; ++contador){
18            System.out.print ("Introduzca una nota: ");
19            /*Leemos un número entero por teclado y lo almacenamos
20            en la posición contador del array*/
21            notas[contador] = teclado.nextInt();
22        }
23        teclado.close();
24        return notas;
25    }
26    private static double calcularNotaMedia(int[] notas) {
27        //Declaramos una variable para almacenar la suma de las notas
28        int suma = 0;
29        //Recorremos el array para sumar todas las notas
30        for (int contador = 0; contador < notas.length; ++contador)
```

```

28         suma = suma + notas[contador];
29         /*Obtenemos la nota media como el resultado de dividir la
30         suma de todas las notas entre el número de notas*/
31         float media = (float) suma/notas.length;
32         return media;
33     }

```

Seguidamente, se ejecuta en modo depuración este programa. Para ello, se abre la vista de depuración seleccionando la opción de menú *Window* → *Perspective* → *Open Perspective* → *Debug* y se establece un punto de ruptura o interrupción en la instrucción del método *calcularNotaMedia* donde se incrementa la variable *suma* con el valor de cada nota del array (línea 34). Para ello, se hace doble clic en el margen izquierdo del editor y aparecerá un circulito en dicho margen, como se puede observar en la Figura 3.7. Además, en la pestaña *Breakpoints* del área de inspección, aparecerá dicho punto de ruptura.



The screenshot shows a Java code editor window titled "NotaMedia.java X". The code is as follows:

```

11     //Llamamos al método que calcula la nota media
12     System.out.println ("La nota media es un " + calcularNotaMedia(notas));
13 }
14
15@ private static int[] leerNotas() {
16     int [] notas = new int [15]; //Declaramos el array
17     //Creamos un objeto de tipo Scanner para leer datos por teclado
18     Scanner teclado = new Scanner (System.in);
19     for (int contador = 0; contador < notas.length; ++contador){
20         System.out.print ("Introduzca una nota: ");
21         /*Leemos un número entero por teclado y lo almacenamos en la
22         posición contador del array*/
23         notas[contador] = teclado.nextInt();
24     }
25     teclado.close();
26     return notas;
27 }
28
29@ private static double calcularNotaMedia(int[] notas) {
30     //Declaramos una variable para almacenar la suma de las notas
31     int suma = 0;
32     //Recorremos el array para sumar todas las notas
33     for (int contador = 0; contador < notas.length; ++contador)
34         suma = suma + notas[contador];
35     /*Obtenemos la nota media como el resultado de dividir la suma
36     de todas las notas entre el número de notas*/
37     float media = (float) suma/notas.length;
38     return media;
39 }

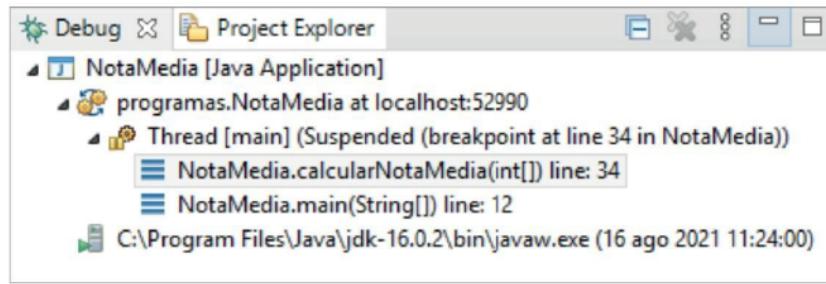
```

A small blue circle is visible on the margin to the left of the line 34 code, indicating a breakpoint has been set there.

**Figura 3.7.** Punto de ruptura o interrupción en la instrucción de la línea 34 reflejado mediante el círculo pequeño que aparece a la izquierda de dicha línea.

El siguiente paso es ejecutar el programa en modo depuración de forma que se ejecute hasta la instrucción en la que se ha establecido el punto de ruptura. Para ello, se activa la opción de menú *Run* → *Debug*, por lo que se tendrá que introducir por teclado las 15 notas que se solicitan.

Al llegar a la instrucción marcada como punto de ruptura, se interrumpe la ejecución del programa. En ese momento, en el área de depuración (Figura 3.8), se muestra para el hilo en ejecución los números de las líneas en las que está parada la ejecución: en la instrucción n.º 34 del método `calcularNotaMedia()` y en la instrucción n.º 12 del método `main`, en la que se llama al método `calcularNotaMedia()`.



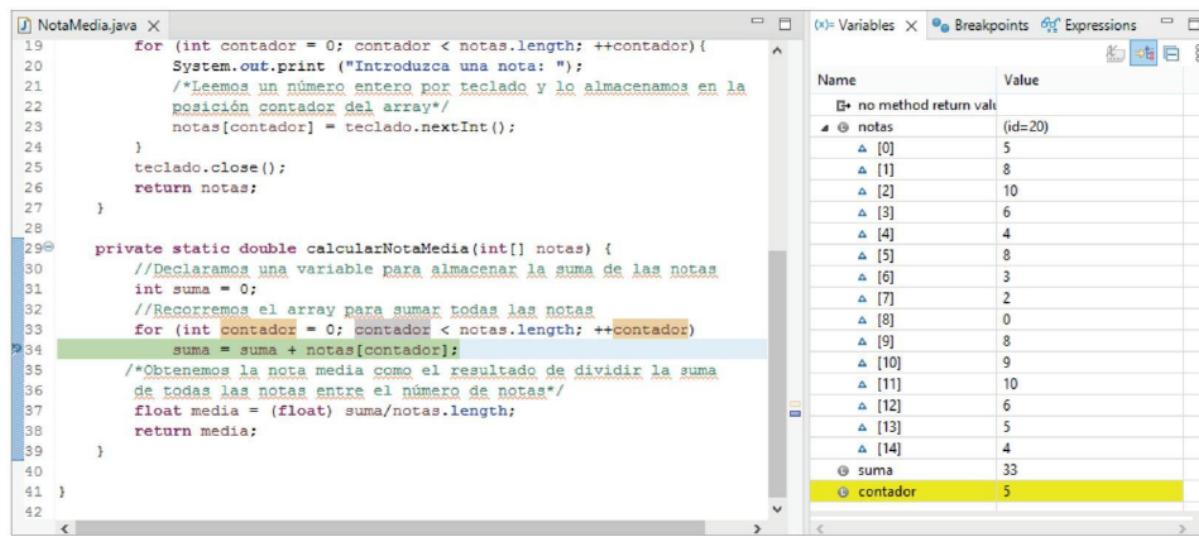
**Figura 3.8.** Área de depuración de la vista de depuración en la que se puede ver el hilo en ejecución y los números de líneas en los que se ha parado la ejecución del programa.

Se puede continuar la ejecución, instrucción a instrucción, de cada método, seleccionando la opción de menú `Run → Step into` o pulsando la tecla F5; o bien ejecutar instrucción a instrucción, pero saltándose los métodos, para lo que habrá que elegir la opción de menú `Run → Step over` o pulsar la tecla F6. Es posible finalizar la ejecución del depurador seleccionando la opción de menú `Run → Terminate` o pulsando las teclas Ctrl + F2.

Se pulsa F5 para ejecutar el programa instrucción a instrucción y la instrucción que se está ejecutando aparece marcada con una flechita en el margen izquierdo.

Si se desea eliminar un punto de ruptura, tan solo hay que hacer doble clic sobre el circulito.

En el área de inspección, en la pestaña `Variables`, se pueden consultar los valores que van tomando las variables en el momento de la ejecución.



**Figura 3.9.** Áreas de edición y de inspección de la vista de depuración en un momento determinado de la ejecución del programa.

En el área de inspección, al ser *notas* un array, se puede hacer clic en el desplegable para ver los valores que toma cada una de sus posiciones. Se puede modificar el valor de cualquier variable colocándose sobre el valor correspondiente y modificándolo.

## 3.6. Pruebas automáticas

Las pruebas de unidad o unitarias, esto es, las pruebas que afectan a una clase y a sus métodos, implican crear algún objeto de la clase que se está probando y realizar llamadas a sus métodos. Esto se puede realizar de manera manual, pero también existen herramientas que permiten realizar pruebas unitarias de manera automatizada, como JUnit, que está integrada en Eclipse.

Para demostrar el funcionamiento de JUnit, en este apartado, se crea una clase *Cuenta* en un proyecto en Eclipse. Esta clase incluye como atributos el número de la cuenta y su saldo, un método constructor con un parámetro por cada uno de los atributos de la clase, métodos de consulta y acceso a cada uno de sus atributos, así como los siguientes métodos:

- *ingresarDinero()*, para incrementar el saldo de la cuenta con el importe recibido como parámetro.
- *extraerDinero()*, para disminuir el saldo de la cuenta con el importe recibido como parámetro.
- *mostrarCuenta()*, que muestra el número de la cuenta y su saldo.

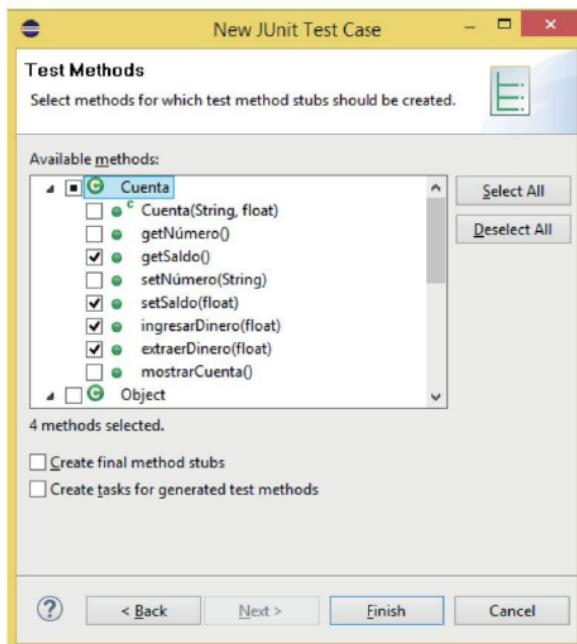
Se muestra a continuación el código de la clase creada:

```
1 package programas;
2 public class Cuenta {
3     private String número;    //Número de la cuenta bancaria
4     private float saldo;      //Saldo de la cuenta bancaria en euros
5     public Cuenta(String numCta, float saldoCta) {
6         número= numCta;
7         saldo = saldoCta;
8     }
9     public String getNúmero(){
10        return número;
11    }
12    public float getSaldo(){
13        return saldo;
14    }
15    public void setNúmero(String numCta){
16        número = numCta;
17    }
18    public void setSaldo(float saldoCta){
19        saldo = saldoCta;
20    }
```

```

20     public void ingresarDinero(float importe){
21         saldo = saldo + importe;
22     }
23     public void extraerDinero(float importe){
24         saldo = saldo - importe;
25     }
26     public void mostrarCuenta(){
27         System.out.println ("Nº cuenta: "+ getNúmero());
28         System.out.println ("Saldo: "+ getSaldo()+" €");
29     }
30 }
```

El siguiente paso es crear una clase de prueba para esta clase. Seleccionada la clase *Cuenta*, se activa con el botón derecho del ratón la opción del menú contextual *New → → JUnit Test Case*. Se dejan las opciones por defecto y se pulsa en el botón *Next*. En la ventana emergente (Figura 3.10), se eligen los métodos que se quieren probar; en este caso, se probarán los métodos *getSaldo*, *setSaldo*, *ingresarDinero* y *extraerDinero*.



**Figura 3.10.** Ventana en la que se seleccionan los métodos que se desean probar, en este caso, para la clase *Cuenta*.

Se creará una clase de prueba llamada *CuentaTest*, cuyo código se muestra a continuación:

```

1 package programas;
2 import static org.junit.jupiter.api.Assertions.*;
3 import org.junit.jupiter.api.Test;
4 class CuentaTest {
5     @Test
```

```
6     void testGetSaldo() {
7         fail("Not yet implemented");
8     }
9     @Test
10    void testSetSaldo() {
11        fail("Not yet implemented");
12    }
13    @Test
14    void testIngresarDinero() {
15        fail("Not yet implemented");
16    }
17    @Test
18    void testExtraerDinero() {
19        fail("Not yet implemented");
20    }
```

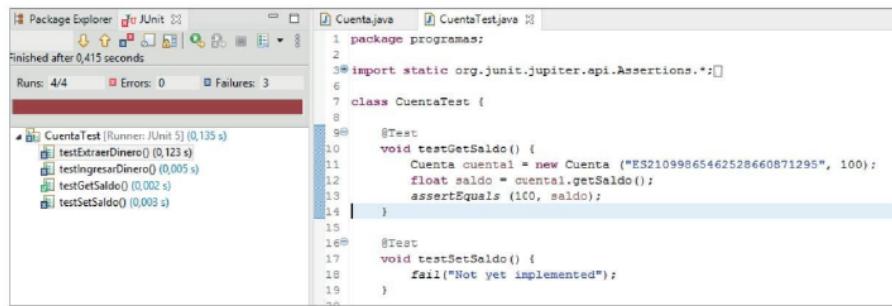
Como se puede observar, se ha creado un método de prueba para cada uno de los métodos que se han seleccionado antes. El nombre de cada uno de estos métodos comienza por *test* y, por cada uno de ellos, aparece la notación *@Test* por ser un método de prueba. A continuación, se deben implementar estos métodos. Para implementar los métodos de prueba, es útil el empleo del método *JUnit assertEquals* (*valorEsperado, valorReal*), que comprueba si el valor esperado indicado coincide con el valor real.

Después, se crea el código para el método de prueba *testGetSaldo()*, que comprueba el funcionamiento del método *getSaldo()*. A tal fin, se crea un objeto de la clase *Cuenta* con un saldo inicial de 100 €. El método se llamará *getSaldo()*, y luego se comprueba con *assertEquals()* si el saldo de la cuenta es de 100 €:

```
1  @Test
2  void testGetSaldo() {
3      Cuenta cuenta1 = new Cuenta ("ES21099865462528660871295", 100);
4      float saldo = cuenta1.getSaldo();
5      assertEquals (100, saldo);
6  }
```

Si se escribe este código para el método y se pulsa el botón *Run* o la opción de menú *Run → Run*, se podrá ver, en la parte izquierda de la pantalla (véase Figura 3.11), que se han ejecutado cuatro casos de prueba en los cuales no se ha producido ningún error, pero que ha habido tres fallos. Al lado de cada método de prueba, hay un símbolo, de manera que:

- Si es de color rojo, indica que se ha producido un error.
- Si es de color azul, indica que se ha producido un fallo.
- Si es de color verde, indica que la prueba ha tenido éxito.



**Figura 3.11.** Pantalla en la que se muestra el resultado de la ejecución de la clase de prueba CuentaTest, obteniendo una prueba exitosa (la del método testGetSaldo()) y tres fallos porque los otros tres métodos aún no han sido implementados.

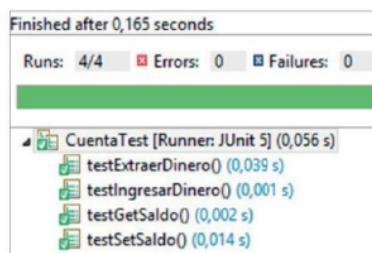
En este caso, en el panel izquierdo, junto al método testGetSaldo(), se observa un símbolo de color verde que indica que el resultado esperado ha coincidido con el obtenido. En el caso de los otros métodos de prueba, se indica que se ha producido un fallo porque aún no han sido implementados. Se implementarán de esta forma:

```

1 @Test
2 void testSetSaldo() {
3     Cuenta cuenta1 = new Cuenta ("ES21099865462528660871295", 0);
4     cuenta1.setSaldo(100);
5     assertEquals (100, cuenta1.getSaldo());
6 }
7 @Test
8 void testIngresarDinero() {
9     Cuenta cuenta1 = new Cuenta ("ES21099865462528660871295", 100);
10    cuenta1.ingresarDinero(400);
11    assertEquals (500, cuenta1.getSaldo());
12 }
13 @Test
14 void testExtraerDinero() {
15     Cuenta cuenta1 = new Cuenta ("ES21099865462528660871295", 100);
16     cuenta1.extraerDinero(20);
17     assertEquals (80, cuenta1.getSaldo());
18 }

```

Una vez implementados, si se hace clic en el botón de ejecución, se indicará que la ejecución de los cuatro casos de prueba ha sido exitosa:



**Figura 3.12.** Parte de la pantalla en la que se muestra el resultado de la ejecución de la clase de prueba CuentaTest, obteniéndose una prueba exitosa para los cuatro métodos de la clase porque los cuatro han podido implementarse y, además, el resultado de la ejecución de cada método coincide con el esperado.



## Recuerda

En relación con lo dicho en el Apartado 3.1, se puede afirmar que un buen caso de prueba es aquel que ofrece una elevada probabilidad de detectar un error, o lo que es lo mismo, aquel que es muy probable que tenga éxito. Recuérdese que el objetivo de las pruebas es detectar errores antes de que el software sea entregado al cliente, por lo que cabe afirmar que las pruebas tienen éxito en la medida en que, mediante su ejecución, se encuentran defectos.

Sin embargo, en el ámbito de la herramienta JUnit, se marcan en color verde aquellos métodos de prueba en los que el resultado obtenido coincide con el esperado. El color verde suele vincularse con el éxito o con lo bueno y, por lo tanto, se puede afirmar que, en JUnit, se considera exitoso aquel método de prueba como consecuencia del cual no se produce un error o divergencia entre el resultado obtenido y el esperado. Esto parece ir en contra de lo afirmado en el párrafo anterior y explicado en apartados anteriores, es decir, en contra de la consideración de que un caso de prueba exitoso es aquel que detecta un fallo.

No obstante, no se trata de una incongruencia en relación con lo que se indicó en el Apartado 3.1, sino, más bien, una forma diferente de interpretar el resultado de la ejecución de un caso de prueba. Hay que seguir siendo conscientes de que la detección de un defecto al ejecutar un caso de prueba es una «buena noticia» en la medida en que permite corregir algo que se ha hecho mal porque, al ser humanos, nos podemos equivocar, y lo positivo de todo ello es que aún se tiene tiempo de enmendar el error sin que el cliente lo detecte.

### 3.6.1. Tratamiento de excepciones

También es posible crear métodos de prueba que comprueben si, en determinados casos, en un método, se ha producido una excepción si esta estaba prevista.

Para exemplificar ese caso, aquí se modificará el método `extraerDinero()` de la clase `Cuenta`, de manera que se produzca una excepción cuando el saldo resultante de la extracción de dinero sea negativo. A tal fin, se lanza una excepción, `ArithmeticeException`, que muestra un mensaje de error indicando lo que ha ocurrido. El método `extraerDinero()` quedaría del siguiente modo:

```
1 public void extraerDinero(float importe) {  
2     if ((saldo - importe) < 0)  
3         throw new java.lang.ArithmeticeException ("Saldo negativo");  
4     else  
5         saldo = saldo - importe;  
6 }
```

Con el fin de comprobar que, efectivamente, se lanza una excepción cuando el saldo de la cuenta es negativo, se modifica el método de prueba `testExtraerDinero()` generando un caso de prueba que dé lugar a esta situación, por ejemplo, creando una cuenta con 100 € y solicitando una extracción de 120 € de esta. Para ello, se debe generar la excepción `ArithmeticeException`, capturando esa excepción y colocando el código que la genera dentro de un bloque `try`. En caso de que no se genere la excepción, como es de esperar, se muestra un mensaje indicando que se ha producido un fallo con `fail`.

```

1  @Test
2  void testExtraerDinero() {
3      try{
4          Cuenta cuenta1 = new Cuenta ("ES21099865462528660871295", 100);
5          cuenta1.extraerDinero(120);
6          fail ("ERROR. Se debería haber lanzado una excepción al
resultar un saldo negativo");
7      }
8      catch (ArithmeticException ae){
9          //Prueba correcta
10     }
11 }
```

Si se ejecuta de nuevo la clase de prueba, se verá que la ejecución de los 4 casos de prueba, `testGetSaldo()`, `testSetSaldo()`, `testIngresardinero()` y `testExtraerDinero()`, es correcta porque en este caso se ha generado una excepción al resultar el saldo de la cuenta negativo.

## 3.6.2. Anotaciones

Muy frecuentemente, al crear un caso de prueba, es necesario repetir ciertas instrucciones en cada uno de los métodos de prueba. Así, en todos los métodos de la clase `CuentaTest` se puso como primera instrucción la de creación de una cuenta con un saldo de 100 €:

```
Cuenta cuenta1 = new Cuenta ("ES21099865462528660871295", 100);
```

No obstante, es posible escribir la anotación `@BeforeEach` delante de un método para colocar en dicho método el código que se debe ejecutar por cada método de prueba y antes de cualquiera de estos métodos.

De forma análoga, existe la anotación `@AfterEach`, para indicar que todas las instrucciones del método para el que se especifica esta anotación se deben ejecutar después de la ejecución de cualquier método de prueba.

A modo de ilustración, aquí se modificará la clase `CuentaTest` para incluir dicha instrucción en un método `nuevaCuenta()`. La clase quedaría así:

```

1 package programas;
2 import static org.junit.jupiter.api.Assertions.*;
3 import org.junit.jupiter.api.Test;
4 import org.junit.jupiter.api.BeforeEach;
5 class CuentaTest {
6     private Cuenta cuenta;
7     @BeforeEach
8     public void nuevaCuenta(){
```

```
9         cuenta = new Cuenta ("ES21099865462528660871295", 100);
10    }
11    @Test
12    void testGetSaldo() {
13        float saldo = cuenta.getSaldo();
14        assertEquals (100, saldo);
15    }
16    @Test
17    void testSetSaldo() {
18        cuenta.setSaldo(100);
19        assertEquals (100, cuenta.getSaldo());
20    }
21    @Test
22    void testIngresarDinero() {
23        cuenta.ingresarDinero(400);
24        assertEquals (500, cuenta.getSaldo());
25    }
26    @Test
27    void testExtraerDinero() {
28        try{
29            cuenta.extraerDinero(120);
30            fail ("ERROR. Se debería haber lanzado una excepción
31            al resultar un saldo negativo");
32        } catch (ArithmetricException ae){
33            //Prueba correcta
34        }
35    }
36 }
```

Asimismo, es muy habitual tener que realizar ciertas acciones una sola vez al inicio de las pruebas, es decir, antes de la ejecución de cualquier método de prueba. Para ello, se debe emplear la anotación `@BeforeAll`. De manera similar, existe la notación `@AfterAll`, cuyas instrucciones se deben ejecutar después de la ejecución de todos los métodos de prueba.

Los métodos con las anotaciones `@BeforeAll` y `@AfterAll`, así como los atributos a los que acceden, deben tener asignada la propiedad `static` porque para el uso de estos métodos y de los atributos a los que acceden, no es necesario crear ningún objeto de la clase a la que pertenecen.

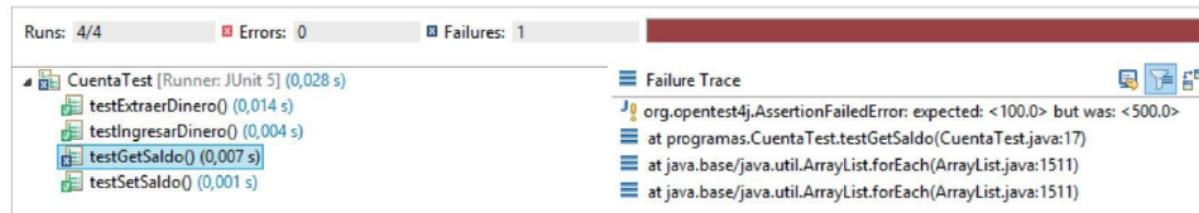
En el caso de la clase de prueba `CuentaTest`, se puede emplear la notación `@BeforeAll` para el método `nuevaCuenta()`, pues la instrucción de creación de la cuenta es suficiente con que se ejecute una sola vez antes de la ejecución de todos los métodos de prueba. Con el objeto de emplear la notación `@BeforeAll`, se modificará la clase `CuentaTest`:

```

1 package programas;
2 import static org.junit.jupiter.api.Assertions.*;
3 import org.junit.jupiter.api.Test;
4 import org.junit.jupiter.api.BeforeAll;
5 class CuentaTest {
6     private Cuenta cuenta;
7     @BeforeAll
8     public void nuevaCuenta(){
9         cuenta = new Cuenta ("ES21099865462528660871295", 100);
10    }
11    @Test
12    void testGetSaldo() {
13        float saldo = cuenta.getSaldo();
14        assertEquals (100, saldo);
15    }
16    @Test
17    void testSetSaldo() {
18        cuenta.setSaldo(100);
19        assertEquals (100, cuenta.getSaldo());
20    }
21    @Test
22    void testIngresarDinero() {
23        cuenta.ingresarDinero(400);
24        assertEquals (500, cuenta.getSaldo());
25    }
26    @Test
27    void testExtraerDinero() {
28        try{
29            cuenta.extraerDinero(120);
30            fail ("ERROR. Se debería haber lanzado una excepción
31                 al resultar un saldo negativo");
32        } catch (ArithmetricException ae){
33            //Prueba correcta
34        }
35    }
36 }

```

Al ejecutar la clase de prueba, se obtiene el resultado que se muestra en la Figura 3.13.



**Figura 3.13.** Resultado de la ejecución de la clase de prueba CuentaTest. En este caso, la prueba ha originado un fallo para el método testGetSaldo().

Como se puede observar, se señala un fallo para el método `testGetSaldo()` porque se esperaba el valor 100, pero el valor del saldo es 500. Esto es debido a que el orden de ejecución de los métodos de prueba no es el orden en el que se han escrito en la clase `CuentaTest`, sino el orden en el que aparecen en el resultado de la ejecución, que se muestra en la Figura 3.13, que es en orden alfabético por nombre de los métodos. Dado que, en el método `testIngresarDinero()`, se ha incrementado el saldo en 400 €, tras haberle asignado a la cuenta un saldo de 100 € al inicio, en el método `nuevaCuenta()`, que tiene asignada la notación `@BeforeAll`, el saldo de la cuenta debe ser 500 €, y no 100 €, como debía ser en la anterior versión de la clase `CuentaTest`. El motivo es que, en ese caso, antes de la ejecución de cada método de prueba, se ejecutaba de nuevo el método `nuevaCuenta()`, por tener este asignado la anotación `@BeforeEach`. Se puede observar, por tanto, como el empleo de una notación u otra (`@BeforeAll` o `@BeforeEach`) es muy relevante. Por tanto, para que el método `testGetSaldo()` se ejecute correctamente, se debe cambiar el valor esperado del saldo a 500 €, quedando el método `testGetSaldo()` así:

```
1 @Test
2 void testGetSaldo() {
3     float saldo = cuenta.getSaldo();
4     assertEquals (500, saldo);
5 }
```

### Recuerda



El criterio de cobertura de las pruebas es relevante porque es el que se usa para determinar en qué momento se considera que se han realizado tantas pruebas con una alta probabilidad de detección de errores como para considerar que ya no es necesario continuar realizando pruebas sobre el elemento del software considerado. Un criterio de cobertura ampliamente empleado es el recorrido de todas las instrucciones del código fuente del elemento que se está probando.

### 3.6.3. Análisis de la cobertura de las pruebas

Cuando se llevan a cabo pruebas, se debe tener un determinado criterio de cobertura que dé por finalizadas las mismas, pues, como se sabe, la prueba exhaustiva del software es impracticable, esto es, no es posible probar todas las posibles entradas con el fin de determinar si en todos esos casos la salida es la esperada.

Eclipse nos da la opción de analizar la cobertura del código que se ha alcanzado con una determinada clase de prueba. De esta manera, nos indica qué parte del código ha sido probada y cuál no.

Si se toman como referencia la clase `Cuenta` y la clase de prueba `CuentaTest`, se puede analizar la cobertura de código eligiendo la opción de menú `Run → Coverage` con la clase de prueba visible en el explorador de proyectos, o bien seleccionar del menú contextual de la clase `CuentaTest` la opción `Coverage As → Junit Test`. En cualquiera de los dos casos, nos aparecerá en el área de consola (parte inferior de la pantalla) una nueva pestaña llamada `Coverage` con la indicación de la cobertura alcanzada, como se muestra en la Figura 3.14.



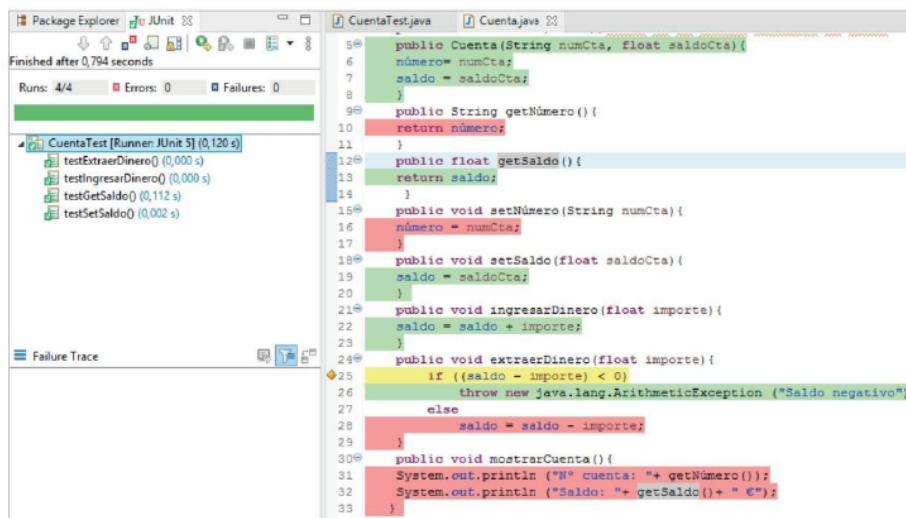
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Ejercicios	6,5 %	70	1.008	1.078
src	6,5 %	70	1.008	1.078
programas	0,0 %	0	964	964
CuentaPrueba	61,4 %	70	44	114
Cuenta.java	48,6 %	35	37	72
Cuenta	48,6 %	35	37	72
mostrarCuenta()	0,0 %	0	23	23
extraerDinero(float)	63,2 %	12	7	19
setNúmero(String)	0,0 %	0	4	4
getNúmero()	0,0 %	0	3	3
Cuenta(String, float)	100,0 %	9	0	9
getSaldo()	100,0 %	3	0	3
ingresarDinero(float)	100,0 %	7	0	7
setSaldo(float)	100,0 %	4	0	4
CuentaTest.java	83,3 %	35	7	42

**Figura 3.14.** Cobertura de prueba alcanzada para la clase Cuenta y para el paquete y el proyecto en el que está incluida esta clase.

Se indica, para cada elemento, el número de instrucciones cubiertas por la prueba (*Covered Instructions*), el número de instrucciones no cubiertas (*Missed Instructions*) y el número total de instrucciones (*Total Instructions*). A partir de estos datos, se calcula la cobertura (*Coverage*), que es el porcentaje que resulta de relacionar el número de instrucciones cubiertas con el número total de instrucciones.

Así, para el caso de la clase *Cuenta*, la cobertura es del 48,6 %. Esto se debe a que hay tres métodos que no han sido probados en absoluto (*setNúmero()*, *getNúmero()* y *mostrarCuenta()*) y otro que solo ha sido probado parcialmente, que es *extraerDinero()*. Si se selecciona en el explorador de proyectos la clase *Cuenta*, es posible observar en el área de edición sus instrucciones en diferentes colores (Figura 3.15):

- El color verde indica que el método ha sido probado por completo.
- El color rojo indica que el método no se ha probado.
- El color amarillo indica que el método se ha probado parcialmente. Por ejemplo, en el método *extraerDinero()*, se puede observar que se ha probado la rama del *if* correspondiente a la excepción (en color verde) pero no la otra rama (en color rojo).



The screenshot shows the Eclipse IDE interface with the JUnit view open. The results pane indicates "Runs: 4/4", "Errors: 0", and "Failures: 0". Below it, the "CuentaTest" test class is expanded, showing four test methods: "testExtraeDinero()", "testIngresarDinero()", "testGetSaldo()", and "testSetSaldo()". The source code for the "Cuenta" class is displayed in the main editor window. Colored annotations highlight specific parts of the code based on test coverage:

- Line 5: "public Cuenta(String numCta, float saldoCta);"
- Line 6: "    numero= numCta;" (green)
- Line 7: "    saldo = saldoCta;" (green)
- Line 9: "public String getNúmero() {"
- Line 10: "    return numero;" (red)
- Line 12: "public float getSaldo() {"
- Line 13: "    return saldo;" (green)
- Line 15: "public void setNúmero(String numCta) {"
- Line 16: "    numero = numCta;" (red)
- Line 18: "public void setSaldo(float saldoCta){"
- Line 19: "    saldo = saldoCta;" (green)
- Line 21: "public void ingresarDinero(float importe){"
- Line 22: "    saldo = saldo + importe;" (green)
- Line 24: "public void extraerDinero(float importe){"
- Line 25: "    if ((saldo - importe) < 0)" (yellow)
- Line 26: "        throw new java.lang.ArithmaticException ("Saldo negativo");" (green)
- Line 27: "    else" (red)
- Line 28: "        saldo = saldo - importe;" (red)
- Line 30: "public void mostrarCuenta(){"
- Line 31: "    System.out.println ("Nº cuenta: "+ getNúmero());" (green)
- Line 32: "    System.out.println ("Saldo: "+ getSaldo()+" €");" (green)
- Line 33: "}" (red)

**Figura 3.15.** Instrucciones del código de la clase Cuenta que se han probado, en diferentes colores, que indican si el método correspondiente se ha probado o no.

De esta forma, se ha mostrado la cobertura de instrucciones del código, pero se pueden mostrar otros tipos de cobertura haciendo clic en el botón ☰ situado en la parte derecha de la pestaña *Coverage* del área de consola. Así, si se selecciona la cobertura de métodos (opción *Method Counters*), el resultado es el que se observa en la Figura 3.16.

Element	Coverage	Covered Methods	Missed Methods	Total Methods
↳ Cuenta.java	62,5 %	5	3	8
↳ Cuenta	62,5 %	5	3	8
● getNúmero()	0,0 %	0	1	1
● mostrarCuenta()	0,0 %	0	1	1
● setNúmero(String)	0,0 %	0	1	1
↳ Cuenta(String, float)	100,0 %	1	0	1
● extraerDinero(float)	100,0 %	1	0	1
● getSaldo()	100,0 %	1	0	1
● ingresarDinero(float)	100,0 %	1	0	1
● setSaldo(float)	100,0 %	1	0	1

Figura 3.16. Cobertura de métodos para la clase Cuenta.

En este caso, la cobertura es del 62,5 % porque de los 8 métodos de la clase *Cuenta* han sido probados 5.

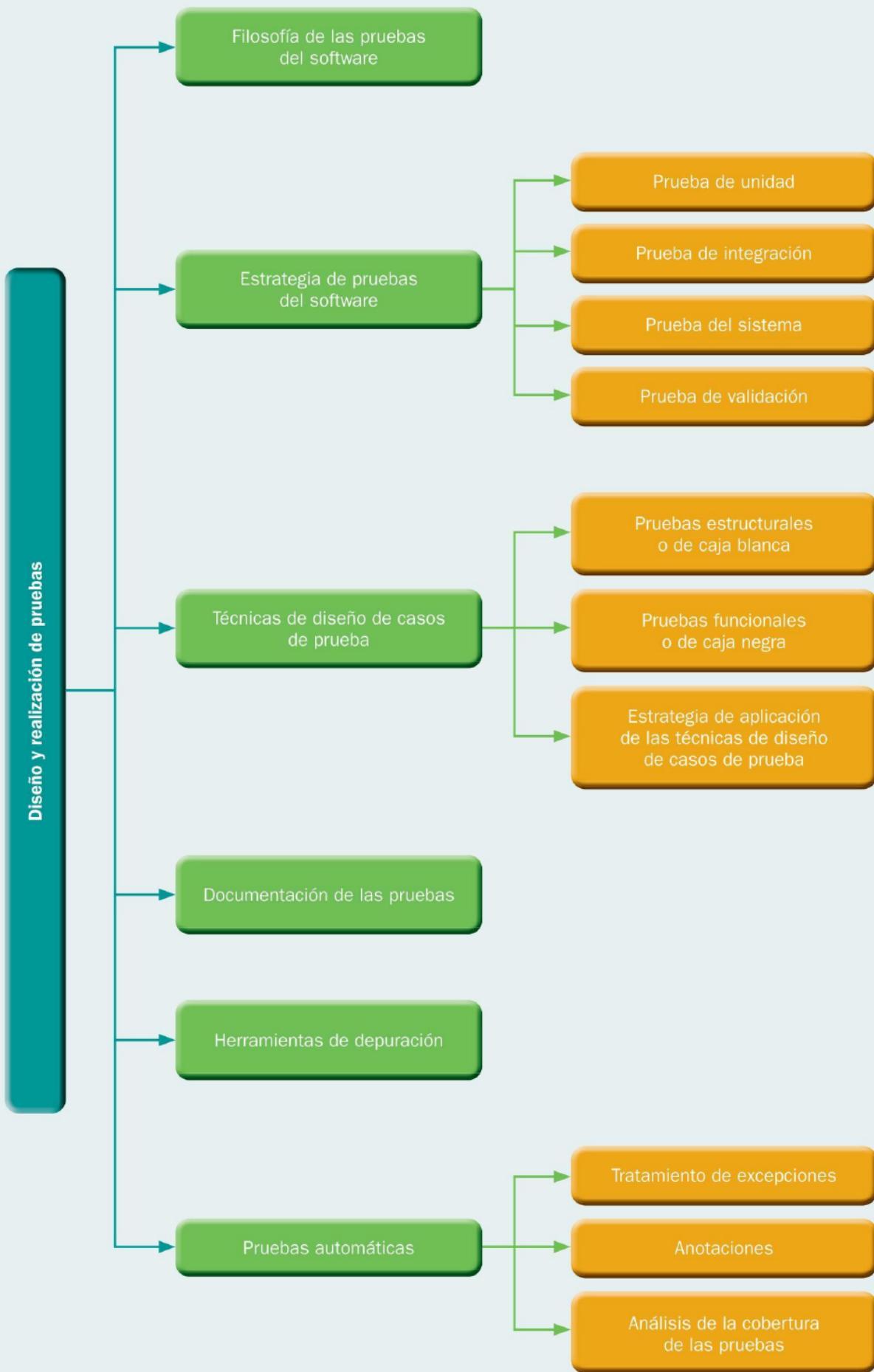
Si se selecciona la opción *Branch Counters*, se señalarán, para los métodos con varias ramas, la cobertura de ramas que se ha conseguido con la clase de prueba. Como en este caso, en la clase *Cuenta*, solo hay un método con dos ramas, el método *extraerDinero()*, el resultado para este método es que la cobertura es del 50 %.

Element	Coverage	Covered Branches	Missed Branches	Total Branches
↳ Cuenta.java	50,0 %	1	1	2
↳ Cuenta	50,0 %	1	1	2
● extraerDinero(float)	50,0 %	1	1	2
↳ Cuenta(String, float)		0	0	0
● getNúmero()		0	0	0
● getSaldo()		0	0	0
● ingresarDinero(float)		0	0	0
● mostrarCuenta()		0	0	0
● setNúmero(String)		0	0	0
● setSaldo(float)		0	0	0

Figura 3.17. Cobertura de ramas para los métodos de la clase Cuenta que tienen ramas. Solo hay un método con dos ramas, de las cuales, se ha probado una de ellas.

# MAPA CONCEPTUAL

## 3. DISEÑO Y REALIZACIÓN DE PRUEBAS



## Actividades de comprobación

- 3.1. Señala la afirmación correcta en relación con las pruebas del software:**
- a) Es posible probar exhaustivamente el software para garantizar que este está exento de errores.
  - b) La prueba del software es por naturaleza una tarea destructiva frente a las restantes etapas del ciclo de vida.
  - c) Mediante las pruebas es posible validar y verificar el software.
  - d) El orden en el que se deben aplicar las pruebas sobre el software viene determinado por las técnicas de diseño de casos de prueba.
- 3.2. Las pruebas cuyo objetivo es garantizar que los elementos que componen el software se comunican adecuadamente y cooperan entre ellos para realizar las tareas encomendadas son las pruebas:**
- a) Del sistema.
  - b) De validación.
  - c) De unidad.
  - d) De integración.
- 3.3. Las técnicas de diseño de casos de prueba en las que se examinan los detalles de cada componente del software para generar los casos de prueba son:**
- a) Las pruebas de caja negra.
  - b) Las pruebas estructurales.
  - c) Las pruebas de caja blanca.
  - d) Las respuestas b y c son correctas.
- 3.4. El gráfico que hay que dibujar a partir del código fuente para poder aplicar la prueba del camino básico se llama:**
- a) Grafo de flujo.
  - b) Organigrama.
  - c) Ordinograma.
  - d) Grafo de programa.
- 3.5. En la prueba del camino básico es posible asignar el mismo nodo a:**
- a) La instrucción que indica el fin de una estructura repetitiva y la siguiente instrucción en secuencia.
  - b) La instrucción que indica el fin de una instrucción condicional y la siguiente instrucción en secuencia.
  - c) Las respuestas a y b son correctas.
  - d) Ninguna de las respuestas anteriores es correcta.
- 3.6. En la técnica de diseño de casos de prueba de flujo de datos, una cadena DU  $[X, I, I']$  es aquella en la que  $X$  está en  $DEF(I)$ ,  $X$  está en  $USO(I')$  y:**
- a) Hay algún camino entre  $I$  e  $I'$  que incluye algún otro uso de la variable  $X$ .
  - b) Hay algún camino entre  $I$  e  $I'$  que incluye alguna otra definición de la variable  $X$ .
  - c) Hay algún camino entre  $I$  e  $I'$  que no incluye ningún otro uso de la variable  $X$ .
  - d) Hay algún camino entre  $I$  e  $I'$  que no incluye ninguna otra definición de la variable  $X$ .

- 3.7.** Indica la regla válida para la generación de casos de prueba en la técnica de clases de equivalencia:
- a) Si una condición de entrada consiste en una condición lógica, se deben identificar una clase válida y dos no válidas.
  - b) Si una condición de entrada consiste en un rango de valores, se deben identificar una clase válida y dos no válidas.
  - c) Si una condición de entrada consiste en un valor específico, se deben identificar una clase válida y una no válida.
  - d) Ninguna de las respuestas anteriores es correcta.
- 3.8.** Las pruebas consistentes en volver a ejecutar casos de prueba tras modificaciones en el software con el fin de asegurarse de que los cambios no han originado nuevos defectos se llaman:
- a) Pruebas de regresión.
  - b) Pruebas de repetición.
  - c) Pruebas de integración.
  - d) Pruebas de validación.
- 3.9.** ¿En qué área de la vista de depuración de Eclipse se pueden ver las variables del programa y los valores que van tomando estas a lo largo de la ejecución del programa?
- a) En el área de depuración.
  - b) En la consola.
  - c) En el área de inspección.
  - d) En el área de edición.
- 3.10.** En el ámbito de las clases de prueba que se pueden crear con JUnit, ¿cuándo se dice que se ha producido un error?
- a) Cuando el resultado de la ejecución de un caso de prueba coincide con el esperado.
  - b) Cuando el resultado de la ejecución de un caso de prueba no coincide con el esperado.
  - c) Cuando un método de prueba no se ha implementado todavía.
  - d) Ninguna de las respuestas anteriores es correcta.

## Actividades de aplicación

- 3.11.** ¿Qué es un caso de prueba?
- 3.12.** ¿Qué diferencia hay entre las pruebas de caja blanca y las de caja negra?
- 3.13.** El siguiente programa escrito en Java pide números por teclado hasta que se introduzca un 0 y calcula, por un lado, la suma de los números pares y, por otro, la suma de los impares. Para este programa, construye el grafo de flujo correspondiente, calcula la complejidad ciclomática e indica un conjunto de caminos independientes. Además, señala, para cada camino, el caso de prueba correspondiente, incluyendo la entrada proporcionada y la salida esperada.

```

1 public static void main(String[] args) {
2     int num;
3     int sumapares = 0;
4     int sumaimpares = 0;
5     Scanner entrada = new Scanner(System.in);
6     System.out.print ("Introduzca un número (0 para
terminar): ");
7     num = entrada.nextInt();
8     while (num != 0)
9     {   if (num % 2 == 0)
10         sumapares = sumapares + num;
11     else
12         sumaimpares = sumaimpares + num;
13     System.out.print ("Introduzca un número (0 para
terminar): ");
14     num = entrada.nextInt();
15 }
16 System.out.println("La suma de los números pares es " +
sumapares);
17 System.out.println("La suma de los números impares es " +
sumaimpares);
18 }
```

- 3.14.** El siguiente programa escrito en Java solicita por teclado un número entero e indica si es primo o no. Para este programa, construye el grafo de flujo correspondiente, calcula la complejidad ciclomática e indica un conjunto de caminos independientes. Además, señala, para cada camino, el caso de prueba correspondiente, incluyendo la entrada proporcionada y la salida esperada.

```

1 public static void main(String[] args){
2     int num, i=2;
3     boolean esPrimo = true;
4     Scanner entrada = new Scanner(System.in);
5     System.out.print ("Introduzca un número entero: ");
6     num = teclado.nextInt();
7     while ( i <= num/2 && esPrimo ){
8         if (num%i == 0){
9             esPrimo = false;
10        }
11        i++;
12    }
13    if (esPrimo)
14        System.out.println ("El número " + num + " es
primo.");
15    else
16        System.out.println ("El número " + num + " no es
primo.");
17 }
```

- 3.15.** Tomando el programa de la Actividad 3.13, obtén las cadenas DU para la variable *sumapares* y encuentra el número mínimo de caminos de prueba que deben recorrer todas estas cadenas.
- 3.16.** Obtén las cadenas DU para la variable *i* del programa de la Actividad 3.14 y encuentra el número mínimo de caminos de prueba que deben recorrer todas estas cadenas.
- 3.17.** Un establecimiento vende sus productos a través de internet y, en la aplicación correspondiente, se solicita al cliente introducir varios datos. Algunos de los datos que se deben introducir y para los que se requieren validaciones son los siguientes:
- a) NIF: debe ser una cadena de 9 caracteres de los cuales los 8 primeros deben ser dígitos, mientras que el último debe ser una letra. La letra debe corresponder a los 8 números de acuerdo con el algoritmo correspondiente.
  - b) El número de la tarjeta de crédito con la que se va a pagar: debe ser un número de 16 cifras.
  - c) La marca de la tarjeta de crédito: solo puede ser Visa, Mastercard o Maestro. Los tratamientos que se deben realizar en cada caso son diferentes.

Crea una tabla de clases de equivalencia y genera los casos de prueba correspondientes, usando la técnica de particiones de equivalencia, e indica, en cada caso, las clases cubiertas.

- 3.18.** Un taller de reparación de vehículos permite reservar cita previa vía internet. En su aplicación, se solicita al cliente introducir varios datos y, para algunos de ellos, se desea realizar validaciones:
- a) La matrícula del vehículo, que debe constar de cuatro números y tres letras.
  - b) El número de puertas del vehículo, que debe ser 3, 4 o 5.
  - c) La potencia del vehículo en caballos, que debe ser un número entero entre 40 y 300.

Genera una tabla de clases de equivalencia y los casos de prueba correspondientes, usando la técnica de particiones de equivalencia, e indica, en cada caso, las clases cubiertas.

- 3.19.** ¿En qué consiste la técnica de prueba de conjetura de errores?
- 3.20.** ¿De qué manera se pueden complementar las técnicas de diseño de casos de prueba de clases de equivalencia y el análisis de valores límite?
- 3.21.** Se proporciona el código de una clase llamada *Punto* con dos atributos para sus coordenadas en dos dimensiones y una serie de métodos que permiten realizar operaciones con puntos:

```
1 public class Punto {  
2     private double x;  
3     private double y;  
4     public Punto(double x, double y){  
5         this.x = x;
```

```
6      this.y = y;
7  }
8  public double getX(){
9      return x;
10 }
11 public double getY(){
12     return y;
13 }
14 public void setX(double x){
15     this.x = x;
16 }
17 public void setY(double y){
18     this.y = y;
19 }
20 public double distancia(Punto p){
21     return Math.sqrt (Math.pow(p.x-this.x, 2) + Math.pow(p.y-
this.y, 2));
22 }
23 public boolean compara(Punto p){
24     if (p.x == x && p.y == y)
25         return true;
26     else
27         return false;
28 }
29 }
```

Crea una clase de prueba para probar el correcto funcionamiento de los métodos *distancia* y *compara* que, respectivamente, calculan la distancia entre dos puntos y los comparan, indicando si son iguales o no. Emplea la anotación *@BeforeEach* para crear dos puntos con los cuales se realicen las dos operaciones.

- 3.22. ¿Para qué se utiliza la notación *@AfterEach* en un método dentro de una clase de prueba?
- 3.23. Calcula la cobertura de métodos y de instrucciones para la clase *Punto* de la Actividad 3.21 y analiza los resultados.

## Actividades de ampliación

- 3.24. ¿Son válidas las estrategias de pruebas de integración expuestas en el Apartado 3.2.2 para software convencional? Si no es así, ¿qué otras estrategias se emplean para este tipo de software?
- 3.25. Busca en la red información sobre las pruebas alfa y beta. ¿De qué tipo de pruebas se trata en cuanto a la estrategia de aplicación de las pruebas? ¿En qué se diferencian las pruebas alfa de las pruebas beta?

- 3.26.** Como se indicó en el Apartado 3.5, las pruebas de regresión consisten en repetir casos de prueba que se ejecutaron antes de realizar ciertas modificaciones para cerciorarse de que los cambios no hayan originado nuevos defectos en el software. Busca información sobre este tipo de pruebas para dar respuesta a las siguientes preguntas: ¿cuándo se deben llevar a cabo este tipo de pruebas?, ¿qué técnicas de diseño de casos de prueba se suelen emplear?, ¿es posible automatizarlas?
- 3.27.** Con ayuda de internet, averigua qué son las pruebas de usabilidad y por qué son tan útiles para las aplicaciones web.
- 3.28.** Investiga en internet en qué consisten las pruebas de humo y cuándo se suelen llevar a cabo.
- 3.29.** Busca en internet información sobre las pruebas de portabilidad. ¿En qué consisten este tipo de pruebas y para qué tipo de software se suelen llevar a cabo?

## Enlaces web de interés

-  **TutorialCup** - <https://www.tutorialcup.com/es/testing>  
(Tutorial sobre temas relacionados con el desarrollo de software)
-  **javiergarzas.com** - <https://www.javiergarzas.com/>  
(Blog de Javier Garzas sobre temas relacionados con el desarrollo de software y, especialmente, con las metodologías ágiles)
-  **Tutorials point** - <https://www.tutorialspoint.com/>  
(Portal web que ofrece tutoriales gratuitos y cursos sobre programación y otras tecnologías)