

UNIDAD 3

Diseño y realización de pruebas

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

3.1. Filosofía de las pruebas del software

- **Objetivo:** detección de defectos en el software antes de que se entregue al cliente, buscando un equilibrio entre el esfuerzo requerido y la capacidad de detección de defectos.
- **Dos puntos de vista:**
 - Validación: ¿el producto es el que quiere el cliente?
 - Verificación: ¿el producto funciona correctamente?
- **Aspectos estratégicos:**
 - Estrategia de aplicación de las pruebas (orden).
 - Técnicas de diseño de casos de prueba.
- **Caso de prueba:** entradas, condiciones de ejecución y resultados esperados para un objetivo particular. Si la salida obtenida no coincide con la esperada → depuración.

3.2. Estrategia de pruebas del software

Las pruebas comienzan por los componentes más pequeños y se va incrementado el alcance de la prueba.

3.2.1. Prueba de unidad

Se comienza probando cada clase y, por tanto, todos sus métodos no triviales.

3.2.2. Prueba de integración

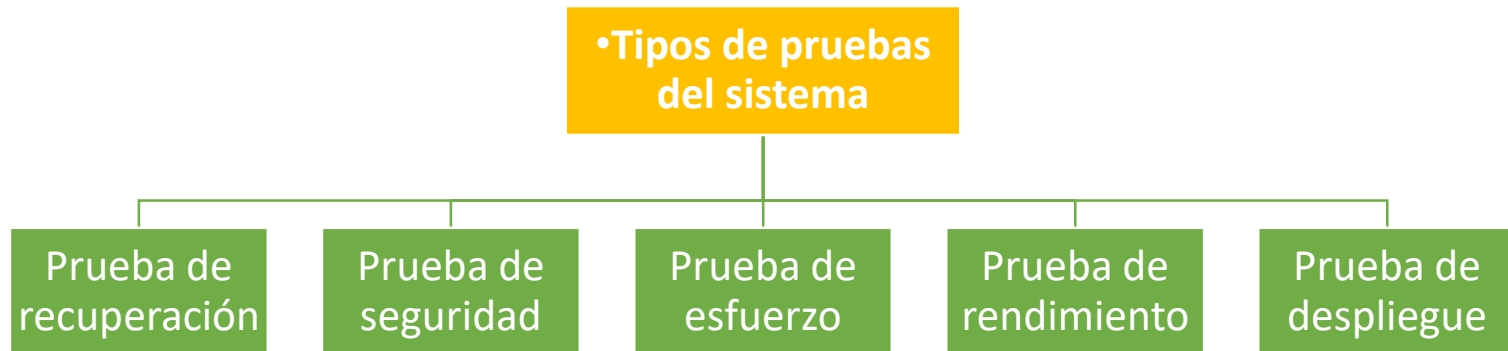
Dos estrategias:

•1. Prueba
basada en hebra

2. Prueba
basada en uso

3.2.3. Prueba del sistema

Prueba de un sistema integrado de hardware y software para comprobar si cumple con todos los requisitos.



3.2.4. Prueba de validación

Consiste en determinar si el software es considerado válido por la persona usuaria y si está preparado para su implantación en el entorno de uso.

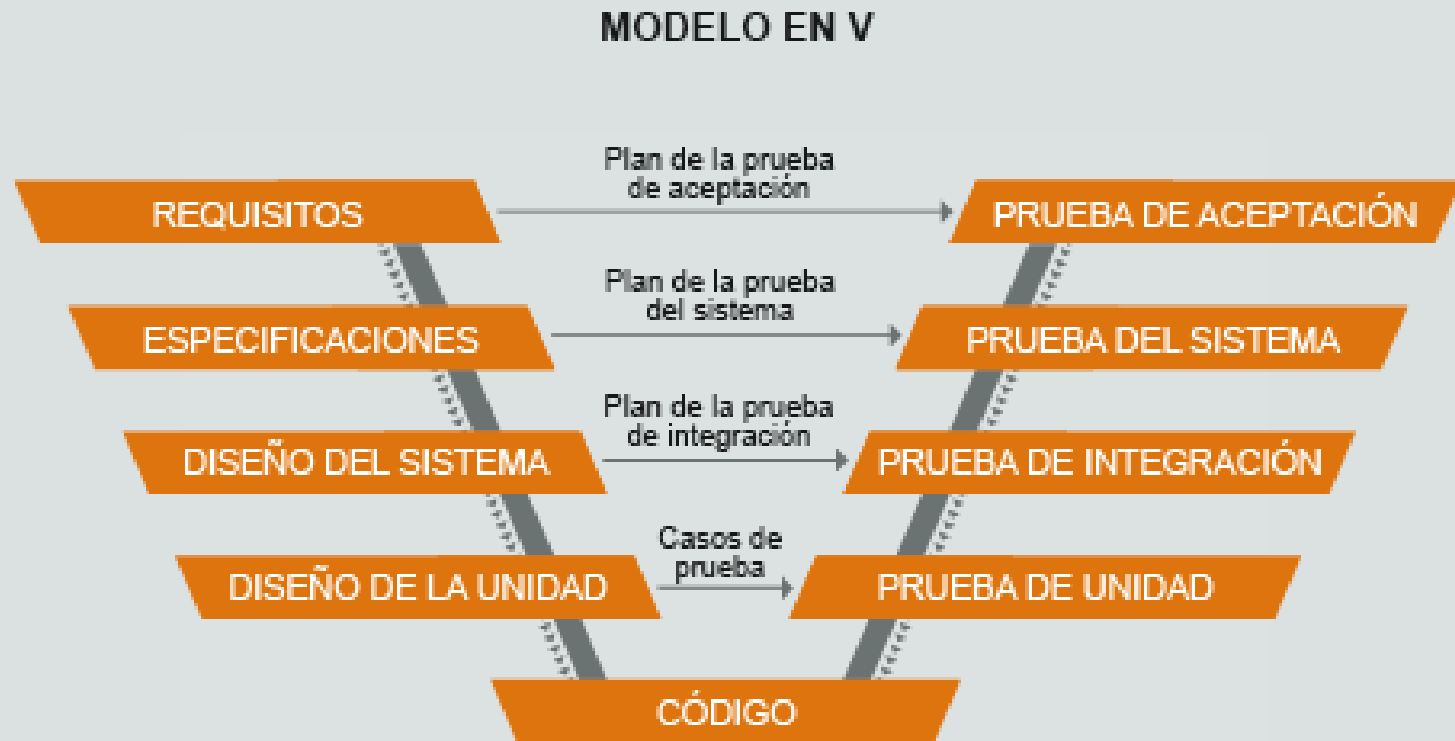
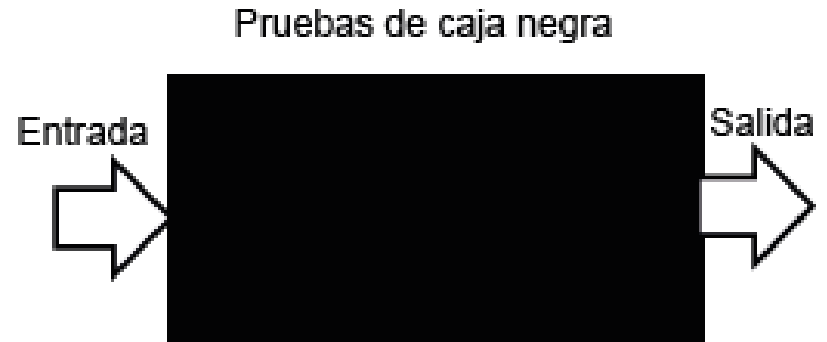
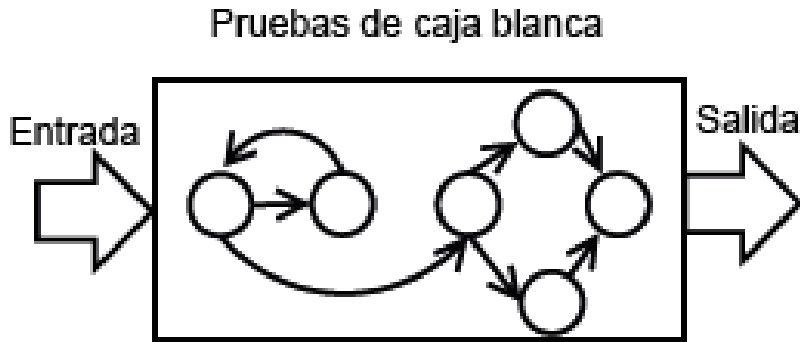


Figura 3.1. Representación del modelo en V, en el que la fase posterior a la programación (las pruebas) se divide en varias subfases que corresponden a las etapas del ciclo de vida en cascada, si bien las etapas de análisis y diseño se dividen cada una de ellas en dos.

3.3. Técnicas de diseño de casos de prueba

- ❑ Diferentes formas en que se puede abordar la prueba del software
- ❑ Tipos de técnicas:
 1. Técnicas de caja blanca o pruebas estructurales.
 2. Técnicas de caja negra o pruebas funcionales.



3.3.1. Pruebas estructurales o de caja blanca

Se deben crear casos de prueba para:

- ☐ Revisar todas las rutas independientes del código.
- ☐ Revisar todas las decisiones lógicas en sus lados verdadero y falso.
- ☐ Ejecutar todos los bucles en sus fronteras y dentro de sus fronteras.
- ☐ Revisar estructuras de datos internas.

Prueba del camino básico

- Se calcula la complejidad ciclomática de McCabe, que indica el número de caminos independientes que hay en el código.
- **Pasos:**
 1. Asignar un número único a cada sentencia y condición → nodo
 2. Construir un grafo de flujo con los nodos y aristas, que representan el flujo de control.

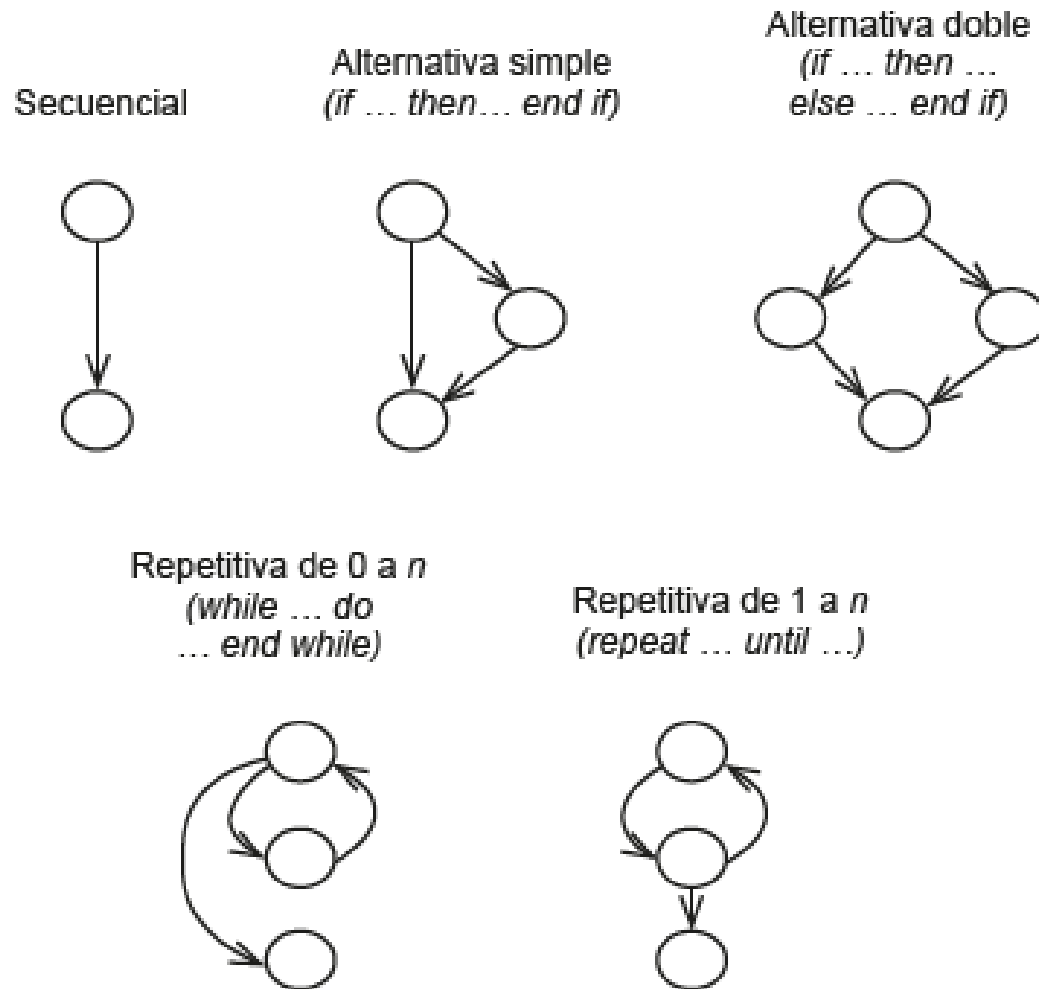


Figura 3.2. Representación de los grafos de flujo para diferentes estructuras de control.

3. Calcular la complejidad ciclomática $V(G)$:

$$\bullet V(G) = \text{nº regiones}$$

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

$$V(G) = \text{nodos predicados} + 1$$

$V(G)$ indica el número de caminos independientes y, por tanto, el número de casos de prueba que hay que crear.

4. Generar casos de prueba para cada camino indicando datos de entrada y salida esperada.

```

begin
  1  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();

  2  while (num != 0)
  3    if (num > 0)
  4      cont_pos++;
      suma_pos+=num;
    else
  5      cont_neg++;
      suma_neg+=num;
    endif;
  6    System.out.print("Introduce número: ");
    num=Entrada.entero();
  7  endwhile;
  8  if (cont_pos != 0)
  9    media_pos = (float)suma_pos/cont_pos;
    System.out.println("Media de los positivos: "+ media_pos);
  10 endif;
  11 if (cont_neg !=0)
  12    media_neg= (float)suma_neg/cont_neg;
    System.out.println("Media de los negativos: "+ media_neg);
  13 endif;
end;

```

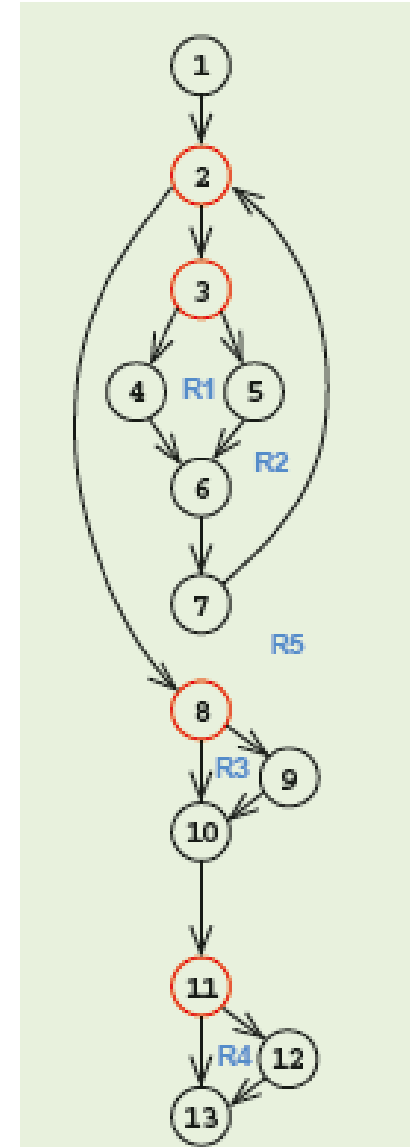
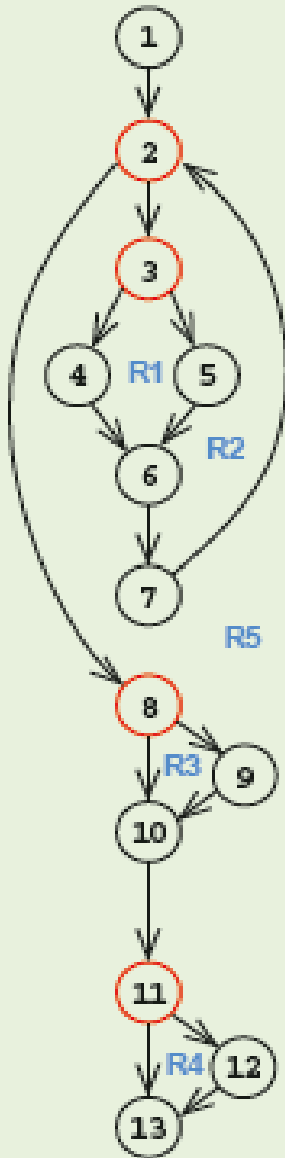


Figura 3.3. Código y grafo de flujo



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

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

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

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

Prueba de bucles

Casos de prueba para cada tipo de bucle:

- ❑ **Bucles simples:** Si n° máximo de iteraciones es n , hay que probar pasar por el bucle 0, 1, 2, m ($m < n$), $n-1$, n y $n+1$ veces.
- ❑ **Bucles anidados:**
 - Comenzar con el bucle más interno, manteniendo el resto en valores mínimos.
 - Aplicar pruebas de bucle simple al bucle más interno, manteniendo los bucles exteriores en sus valores mínimos.
 - Ir de fuera hacia dentro manteniendo los otros bucles exteriores en valores mínimos y los otros interiores en valores típicos.
 - Continuar hasta probar todos los bucles.
- ❑ **Bucles concatenados:**
 - Si son independientes, aplicar estrategia para bucles simples.
 - Si son dependientes, aplicar estrategia para bucles anidados.

Prueba de flujo de datos

1. Asignar un número (nodo) a cada instrucción y condición (grafo de flujo).
2. Calcular los conjuntos DEF y USO para cada instrucción:

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

3. Encontrar las cadenas DU para cada variable: $\text{DU}[X, I, I']$ tal que:
 - X está en $\text{DEF}(I)$
 - X está en $\text{USO}(I')$
 - Entre I e I' no hay ninguna otra definición de X
4. Generar el número mínimo de casos de prueba que incluyan caminos para todas las cadenas DU.
5. Determinar datos de entrada para cada cadena DU y la salida esperada.

begin

```
1  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();
```

```
2  while (num != 0)
```

```
3      if (num > 0)
```

```
4          cont_pos++;
          suma_pos+=num;
```

```
5      else
```

```
        cont_neg++;
        suma_neg+=num;
```

```
        endif;
```

```
6      System.out.print("Introduce número: ");
      num=Entrada.entero();
```

```
7  endwhile;
```

```
8  if (cont_pos != 0)
```

```
9      media_pos = (float)suma_pos/cont_pos;
      System.out.println("Media de los positivos: "+ media_pos);
```

```
10 endif;
```

```
11 if (cont_neg !=0)
```

```
12     media_neg= (float)suma_neg/cont_neg;
     System.out.println("Media de los negativos: "+ media_neg);
```

```
13 endif;
```

```
end;
```

DEF(1) = {num}

USO(2) = {num}

USO(3) = {num}

USO(4) = {num}

USO(5) = {num}

DEF(6) = {num}

DU [num, 1, 2] (1)

DU [num, 1, 3] (2)

DU [num, 1, 4] (3)

DU [num, 1, 5] (4)

DU [num, 6, 2] (5)

DU [num, 6, 3] (6)

DU [num, 6, 4] (7)

DU [num, 6, 5] (8)

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).

3.3.2. Pruebas funcionales o de caja negra

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

Particiones o clases de equivalencia

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

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)

Tabla 3.2. Caso de prueba con clases de equivalencia válidas

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

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

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

Análisis de valores límite

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

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

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

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

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

Conjetura de errores

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

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

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

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

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

Prueba del sistema → pruebas de caja negra.

Prueba de validación → pruebas de caja negra del sistema.

3.4. Documentación de las pruebas

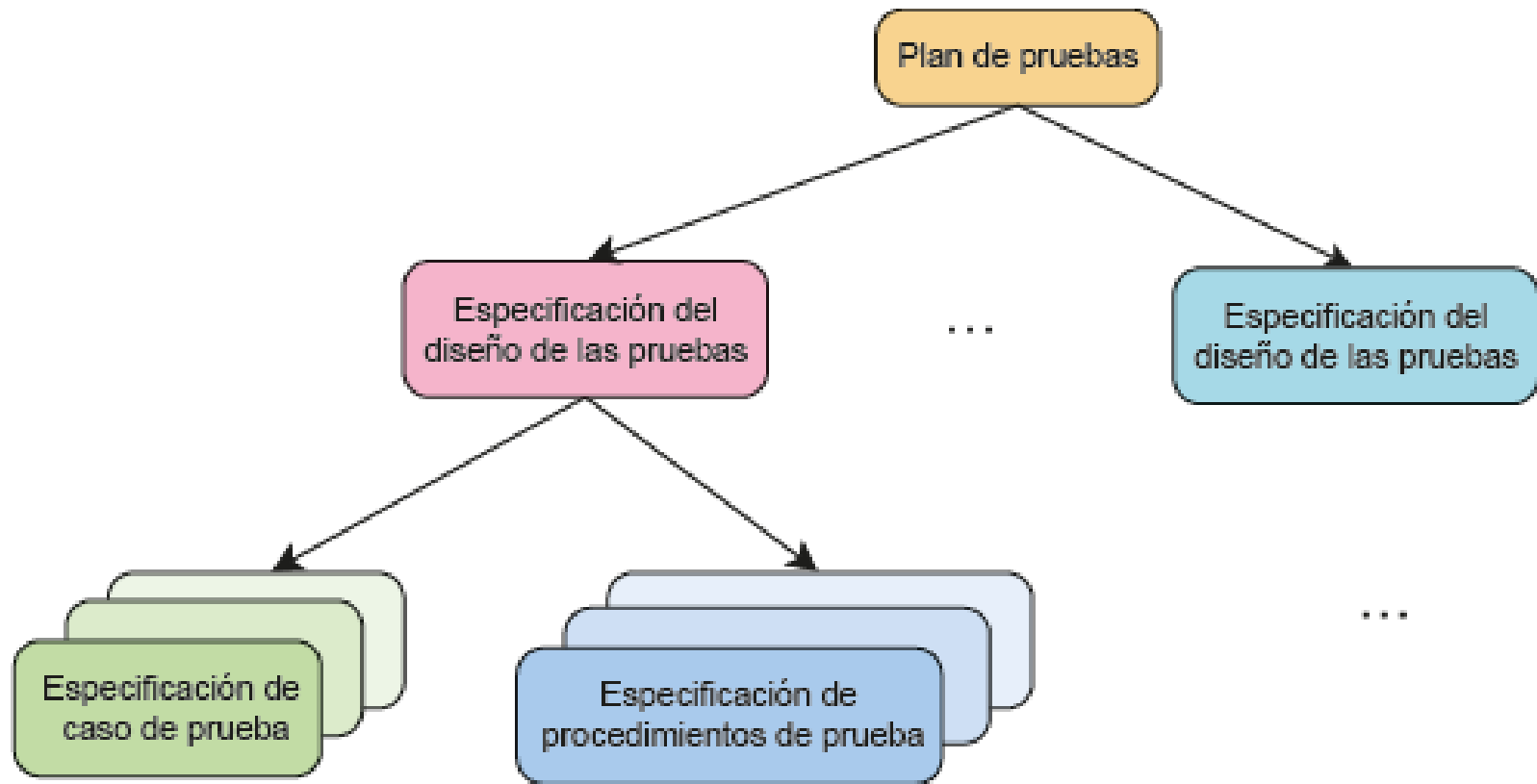


Figura 3.4. El plan de pruebas es un documento único para una aplicación con indicaciones generales acerca de las pruebas. Este se desglosa, por cada elemento que se va a probar, en varias especificaciones del diseño de las pruebas. Cada uno de estos documentos contiene además varias especificaciones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

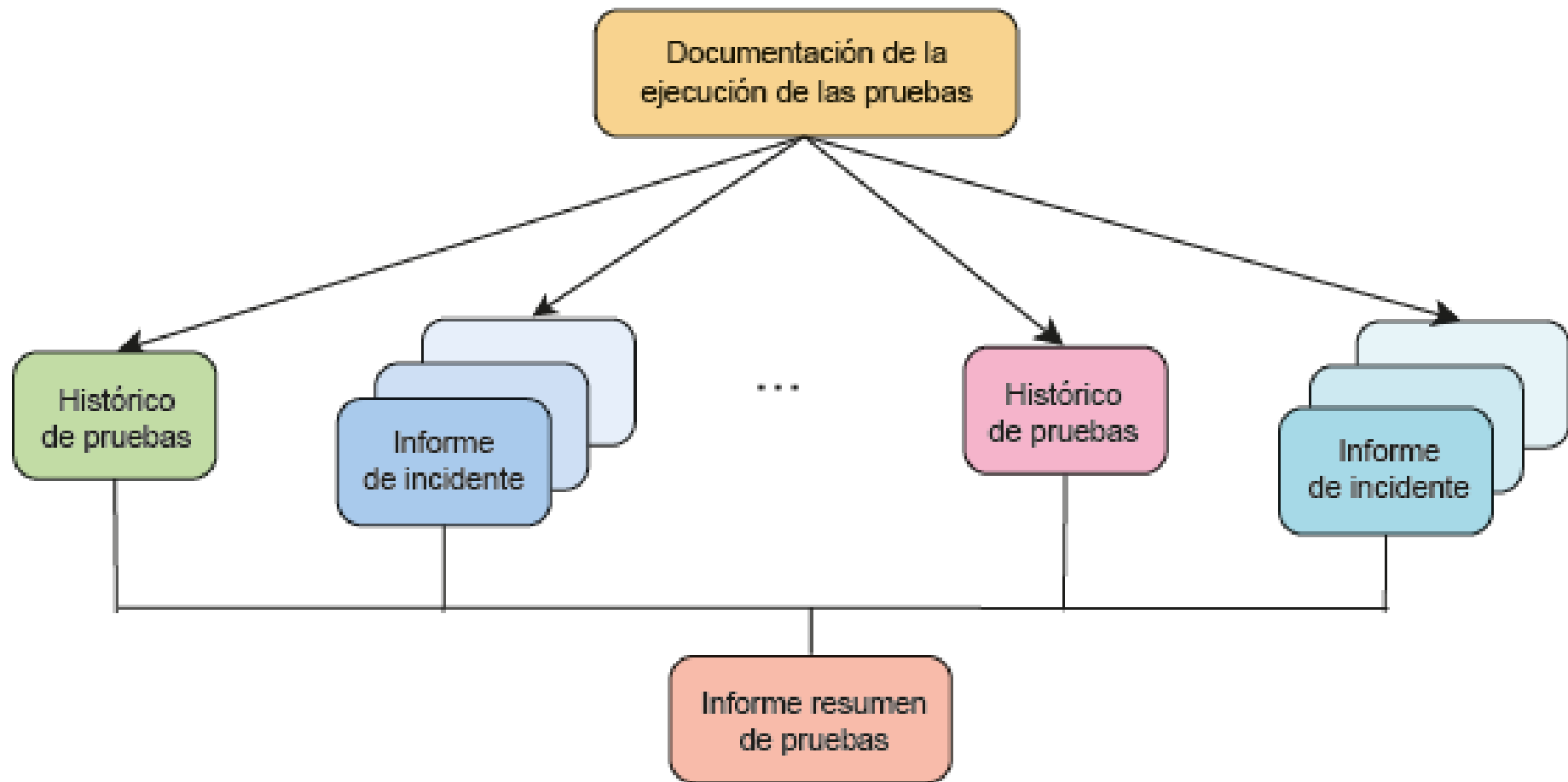


Figura 3.5. Se parte de una especificación del diseño de las pruebas y, por cada ejecución de estas, a partir de una serie de casos de pruebas y sus procedimientos, se genera un histórico de pruebas con todos los hechos relevantes ocurridos durante esas pruebas y un informe por cada incidente ocurrido.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


3.5. Herramientas de depuración

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

❑ **Ayudas de un depurador:**

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

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

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

Opción de menú Window → Perspective → Open Perspective → Debug

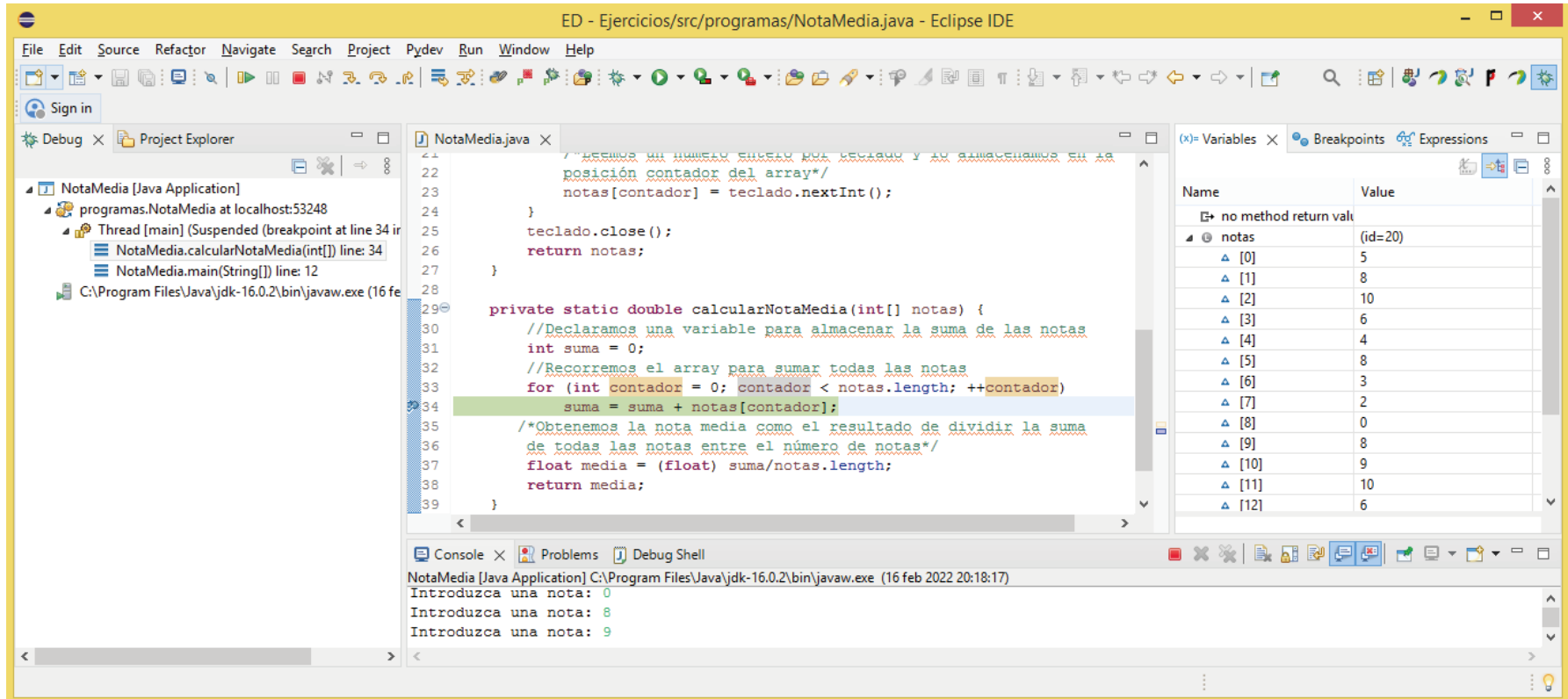
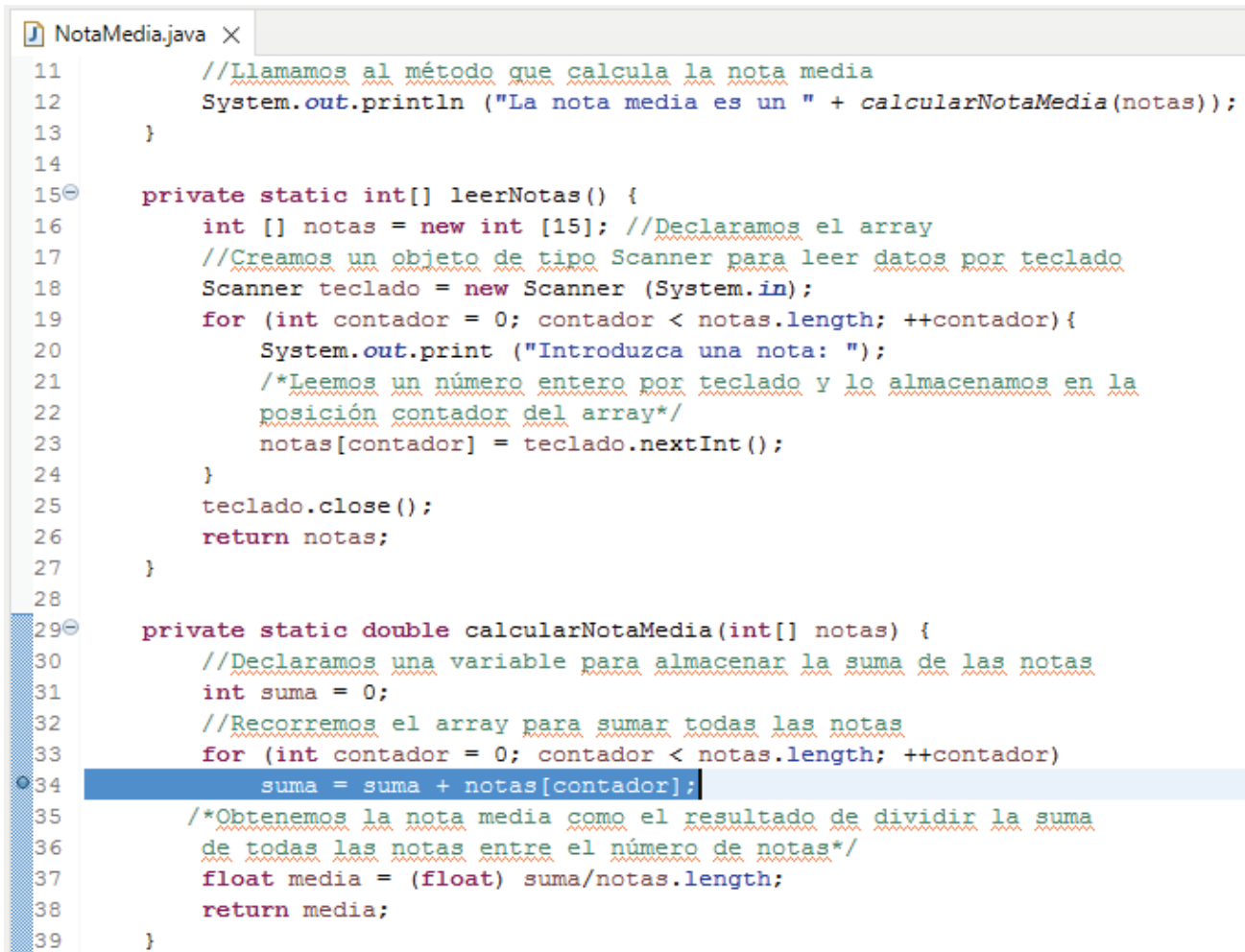


Figura 3.6. Vista Depuración mientras se está ejecutando una aplicación en modo de depuración.



```
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 }
```

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.

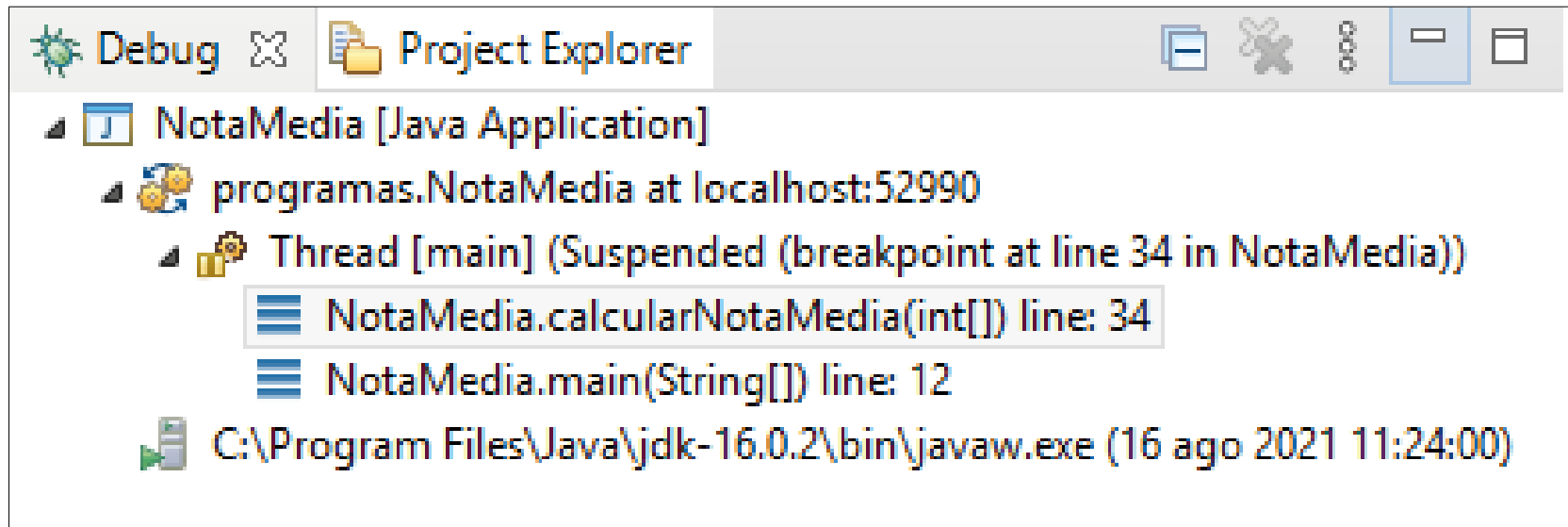


Figura 3.8. Área de depuración de la vista de depuración en la que se puede ver el hilo en ejecución y los números de líneas en los que se ha parado la ejecución del programa.

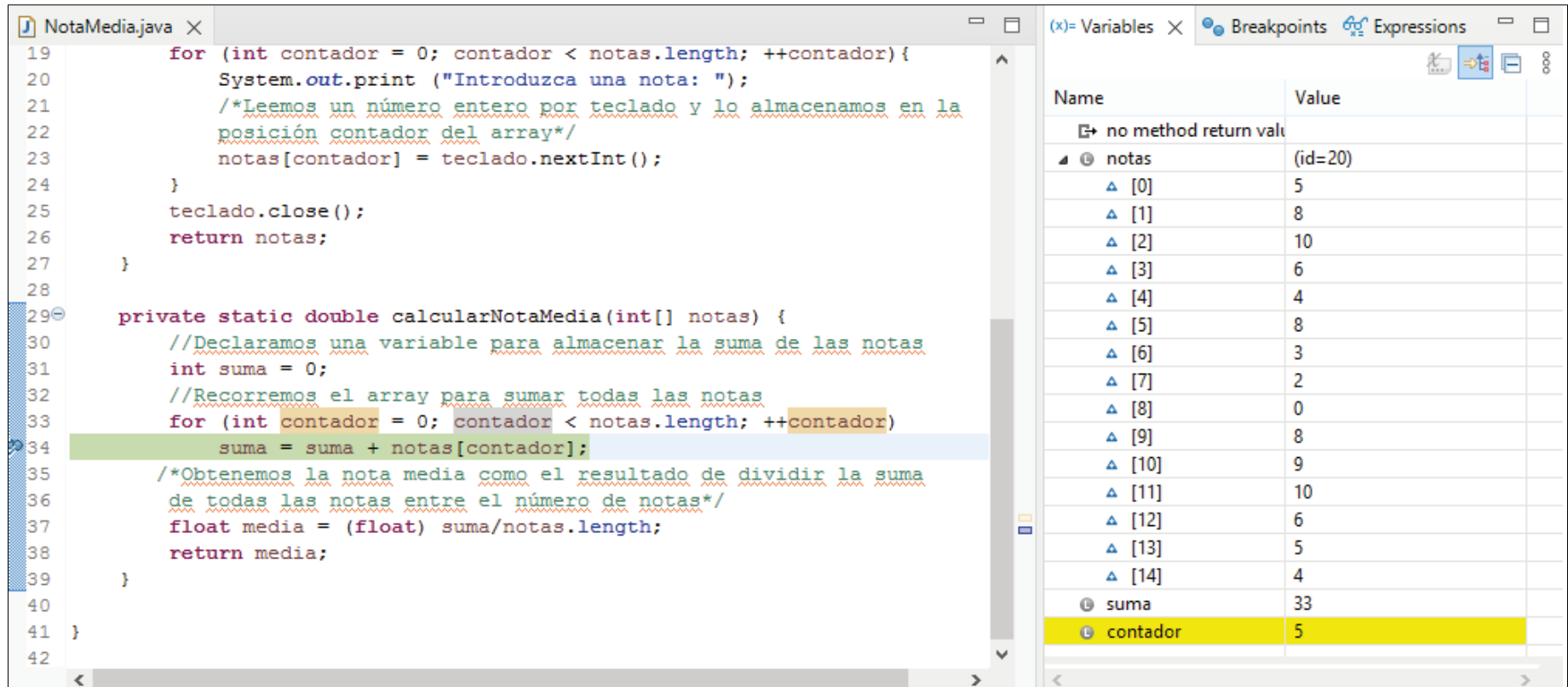


Figura 3.9. Áreas de edición y de inspección de la vista de depuración en un momento determinado de la ejecución del programa.

3.6. Pruebas automáticas

- En Eclipse se puede emplear JUnit para hacer pruebas unitarias.
- A modo de ejemplo, se va a partir de la siguiente clase *Cuenta*:

```
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;  
10    }  
11    public float getSaldo(){  
12        return saldo;  
13    }  
14    public void setNúmero(String numCta){  
15        número = numCta;  
16    }  
17    public void setSaldo(float saldoCta){  
18        saldo = saldoCta;  
19    }  
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 }
```

Se crea una clase de prueba seleccionando del menú contextual de la clase *Cuenta* la opción New → JUnit Test Case

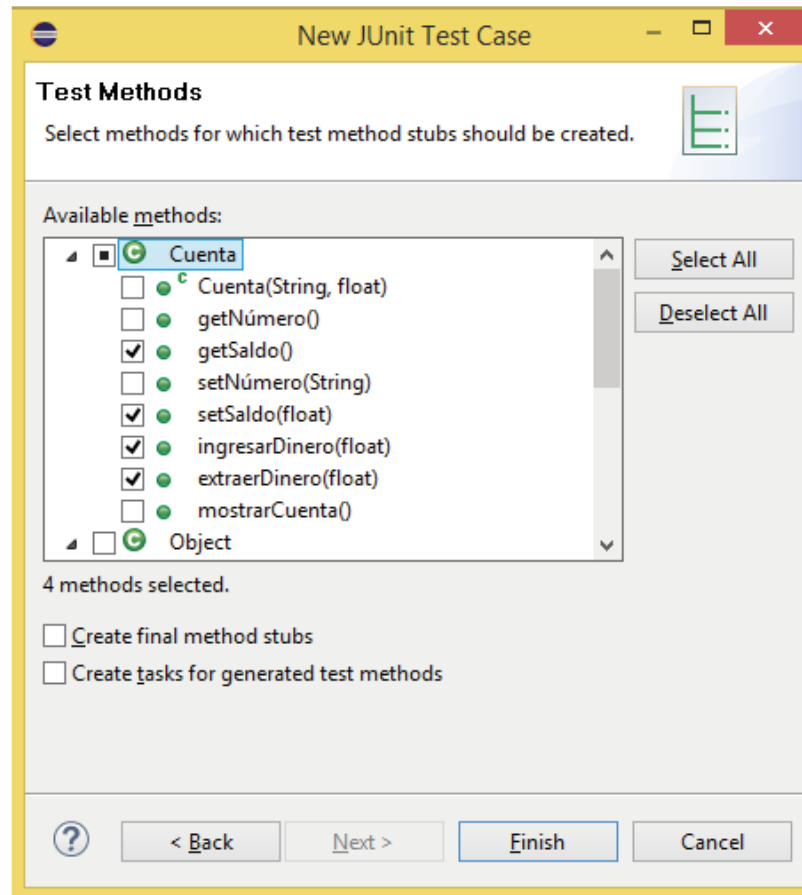


Figura 3.10. Ventana en la que se seleccionan los métodos que se desean probar, en este caso, para la clase *Cuenta*.

Tras implementar el método *testGetSaldo*, se obtiene el siguiente resultado:

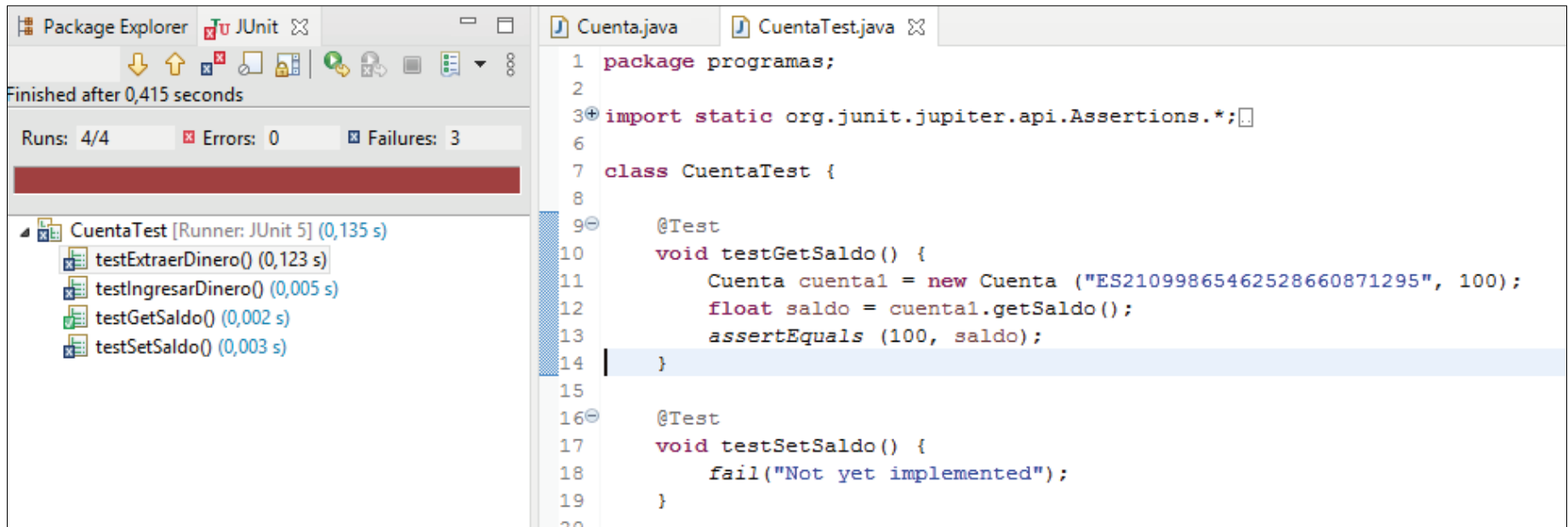


Figura 3.11. Pantalla en la que se muestra el resultado de la ejecución de la clase de prueba *CuentaTest*, obteniendo una prueba exitosa (la del método *testGetSaldo*) y tres fallos porque los otros tres métodos aún no han sido implementados.

Tras implementar correctamente los otros 3 métodos, se obtiene el siguiente resultado:

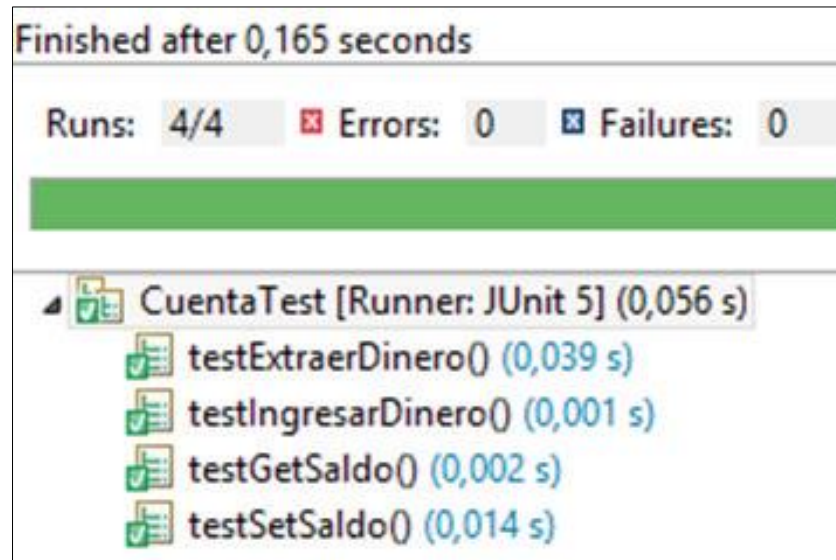


Figura 3.12. Parte de la pantalla en la que se muestra el resultado de la ejecución de la clase de prueba *CuentaTest*, obteniéndose una prueba exitosa para los cuatro métodos de la clase porque los cuatro han podido implementarse y, además, el resultado de la ejecución de cada método coincide con el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

3.6.1. Tratamiento de excepciones

- Se modifica el método *extraerDinero* para que si el saldo resultado es negativo, se lance la excepción *ArithmeticException*.

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

- Se codifica el método de prueba con un caso en el que se debe lanzar la excepción:

```
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 }
```

3.6.2. Anotaciones

- *@BeforeEach* antepuesto a un método → Código que se debe ejecutar antes de cada método de prueba.
- *@AfterEach* antepuesto a un método → Código que se debe ejecutar después de cada método de prueba.
- *@BeforeAll* antepuesto a un método → Código que se debe ejecutar una sola vez antes de todos los métodos de prueba.
- *@AfterAll* antepuesto a un método → Código que se debe ejecutar una sola vez después de todos los métodos de prueba.

3.6.3. Análisis de la cobertura de las pruebas

- Criterio de cobertura para dar por finalizadas las pruebas.
- Con Eclipse es posible determinar la cobertura del código para una clase. Nos indica qué parte del código ha sido probada y cuál no.
- Se elige del menú contextual para la clase *CuentaTest* la opción Coverage As → JUnit Test.

Problems @ Javadoc Declaration Coverage				
Element	Coverage	Covered Instructio...	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.

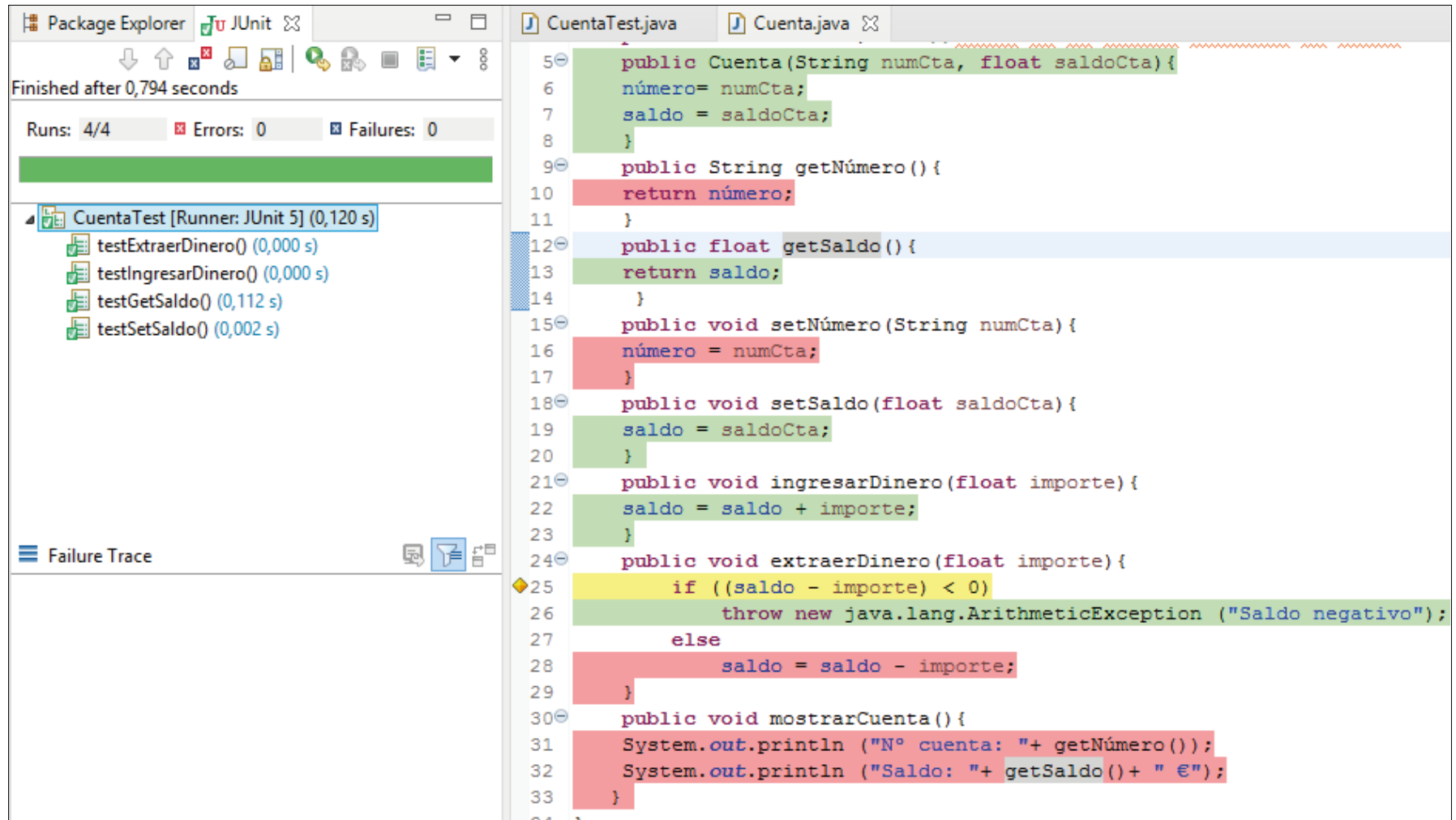


Figura 3.15. Instrucciones del código de la clase *Cuenta* que se han probado, en diferentes colores, que indican si el método correspondiente se ha probado o no.

Problems @ Javadoc Declaration Coverage

CuentaTest (1) (25 ago 2021 16:56:06)

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*.

Problems @ Javadoc Declaration Coverage

CuentaTest (1) (25 ago 2021 16:56:06)

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.