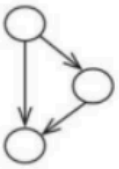
grafos de flujo

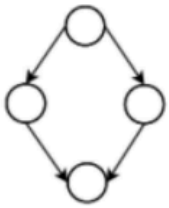
Secuencial

Condicional



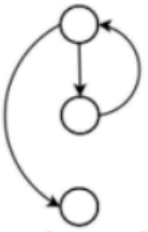
Si condición **entón**
<instrucións>
Fin si

Condicional



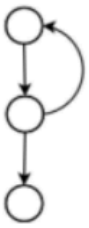
Si condición **entón**
<instrucións>
Sino
<instrucións>
Fin si

Facer Mentres



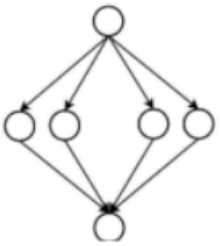
Mentres condición **facer**
<instrucións>
Fin Mentres

Repetir ata



Repetir
<instrucións>
Ata que condición

Condicional múltiple



Segundo sexa variable
Caso opc-1:
<instrucións>
Caso opc-2:
<instrucións>
Outro caso:
<instrucións>
Ata que condición

Complejidad ciclomática:

1. contar regiones
2. aristas - nodos + 2
3. decisiones + 1 (`if`, `while`, `for`, `case` o `switch`).)

AssertEquals:

La sintaxis básica de `assertEquals` es la siguiente:

```
assertEquals(expected, actual);
```

Donde:

- **`expected`** es el valor que esperas que tenga el resultado.
- **`actual`** es el valor que obtuviste como resultado de ejecutar el código.

Si el valor de `expected` es igual al valor de `actual`, la prueba pasará. Si no son iguales, la prueba fallará.

Ejemplo 1: Comparando dos números

Supongamos que tienes una función que suma dos números y quieres verificar que la suma sea correcta.

java
Copiar

```
public class Calculadora {
    public int suma(int a, int b) {
        return a + b;
    }
}
```

Ahora, usas `assertEquals` para comprobar que el resultado es correcto en una prueba unitaria:

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculadoraTest {

    @Test
    public void testSuma() {
        Calculadora calc = new Calculadora();

        // Ejecutar el método suma
        int resultado = calc.suma(3, 2);

        // Verificar que el resultado esperado (5) sea igual al
resultado obtenido
        assertEquals(5, resultado);
    }
}
```

- **`expected`**: 5 (el valor que esperamos que la suma nos devuelva).
- **`actual`**: `resultado` (el valor que obtenemos al ejecutar `suma(3, 2)`).

Ejemplo 2: Comparando cadenas de texto

Supongamos que tienes una función que convierte un nombre a mayúsculas:

java
Copiar

```
public class StringUtils {
    public String convertirAMayusculas(String texto) {
        return texto.toUpperCase();
    }
}
```

En la prueba unitaria, puedes usar `assertEquals` para verificar si la conversión es correcta:

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class StringUtilsTest {

    @Test
    public void testConvertirAMayusculas() {
        StringUtils utils = new StringUtils();

        // Ejecutar el método convertirAMayusculas
        String resultado = utils.convertirAMayusculas("hola");

        // Verificar que el resultado esperado sea igual al
resultado obtenido
        assertEquals("HOLA", resultado);
    }
}
```

- **`expected`**: `"HOLA"` (el valor que esperamos después de convertir la cadena a mayúsculas).
- **`actual`**: `resultado` (el valor que obtenemos al ejecutar `convertirAMayusculas("hola")`).

Ejemplo 3: Comparando valores booleanos

Otro caso común es comparar valores booleanos. Imagina que tienes una función que verifica si un número es par:

```
public class ParImpar {
    public boolean esPar(int numero) {
        return numero % 2 == 0;
    }
}
```

En la prueba unitaria, puedes usar `assertEquals` para verificar si el resultado es `true` o `false`:

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class ParImparTest {

    @Test
    public void testEsPar() {
        ParImpar pi = new ParImpar();

        // Ejecutar el método esPar
        boolean resultado = pi.esPar(4);

        // Verificar que el resultado esperado sea igual al
resultado obtenido
        assertEquals(true, resultado);
    }
}
```

- **`expected`**: `true` (el número 4 es par, por lo que esperamos que el resultado sea `true`).
- **`actual`**: `resultado` (el valor que obtenemos al ejecutar `esPar(4)`).

Resumen:

- `assertEquals(expected, actual)` compara el valor esperado con el valor actual.
- Si ambos son iguales, la prueba pasa.
- Si son diferentes, la prueba falla, lo que indica que hay un error en el código.

Orden:

1. hacer el grafo
2. complejidad
3. caminos > 1,3,6.. // 1,4,7…
4.

1. Ejercicios

C11. Consejería de sanidad

C311

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.

Táboa de Clases de Equivalencia

Condición de Entrada	Clases Válidas	Clases No Válidas
Tarjeta Sanitaria	(1) número entero de 8 dígitos	(2) más de 8 dígitos (3) menos de 8 dígitos (4) no es un número entero
Apellido	(5) cadena entre 2 y 30 letras pudiendo incluir algún espacio	(6) longitud menor de 2 (7) longitud mayor de 30 (8) incluye símbolos distintos de letras y espacio

Año Nacimiento	(9) número entero entre 1901 y 2025	(10) menor de 1901 (11) mayor de 2025 (12) no es un número entero

Casos de proba con clases de equivalencia válidas

Tarjeta Sanitaria	Apellido	Año Nacimiento	Clases incluidas
12345678	Marín	2000	1, 5, 9

Casos de proba con clases de equivalencia non válidas

Tarjeta Sanitaria	Apellido	Año Nacimiento	Clases incluidas
123456789	Marín	2000	2, 5, 9
1234567	Marín	2000	3, 5, 9
1234567X	Marín	2000	4, 5, 9
12345678	M	2000	1, 6, 9
12345678	MarínMarínMarínMarínMarínMarín	2000	1, 7, 9
12345678	Marín5	2000	1, 8, 9
12345678	Marín	1900	1, 5,
12345678	Marín	2026	1, 5,
12345678	Marín	DosMil	1, 5,

C312

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.

Táboa de Clases de Equivalencia

Condición de Entrada	Clases Válidas	Clases Non Válidas

Casos de proba con clases de equivalencia válidas

Entrada1	Entrada2	Entrada3	Clases incluidas

Casos de proba con clases de equivalencia non válidas

Entrada1	Entrada2	Entrada3	Clases incluidas
…			

C313

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.

Táboa de Clases de Equivalencia

Condición de Entrada	Clases Válidas	Clases Non Válidas

Casos de proba con clases de equivalencia válidas

Entrada1	Entrada2	Entrada3	Clases incluidas

Casos de proba con clases de equivalencia non válidas

Entrada1	Entrada2	Entrada3	Clases incluidas

1. Cálculo do Factorial

Implementa unha clase `MathUtils` con un método `factorial(int n)` que devolva o factorial dun número enteiro positivo. O método debe lanzar unha excepción se `n` é negativo.

```
public class MathUtils {
    public static int factorial(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("O número debe ser positivo");
        }
        int resultado = 1;
        for (int i = 1; i <= n; i--) { // Erro: debería ser i++
            resultado *= i;
        }
        return resultado;
    }
}
```

MathUtils.java

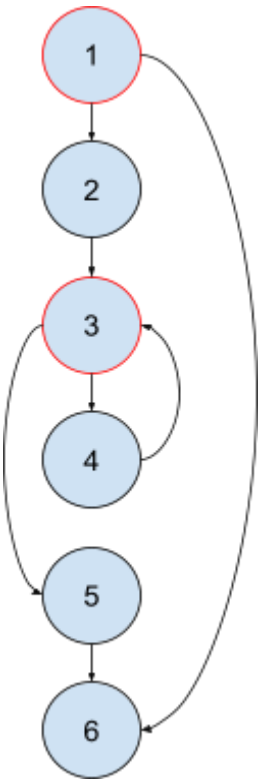
Probas do camiño básico

V(G) = 3
Camiños independentes:

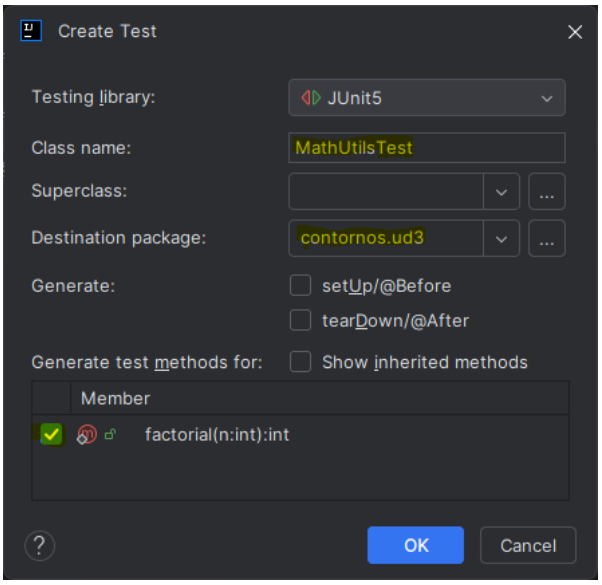
1. 1, 6
2. 1, 2, 3, 5, 6 (no entrar en el bucle)
3. 1, 2, 3, 4, 3, 5, 6 (entrar una vez en el bucle)

Casos de proba:

1. n = -1 Salida esperada: Excepción
2. n = 0 Salida esperada: 1
3. n = 1 Salida esperada: 1 = 1! (pero.. bucle infinito!! >> ERRO)



Crear clase de probas



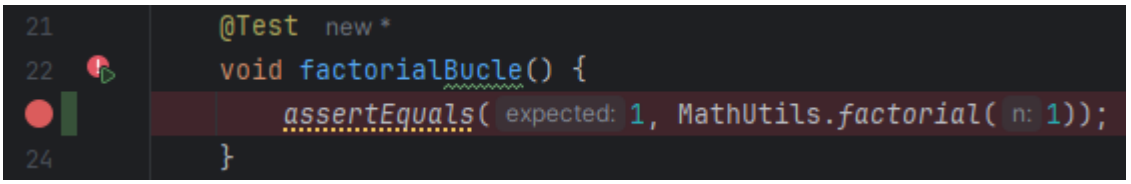

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class MathUtilsTest {
    @Test
    void factorialExcepcion() {
        try {
            int res = MathUtils.factorial(-1);
            fail("FALLO: n < 0 debería generar una excepción");
        } catch (Exception e) {
        }
    }

    @Test
    void factorialNoBucle() {
        assertEquals(1, MathUtils.factorial(0));
    }

    @Test
    void factorialBucle() {
        assertEquals(1, MathUtils.factorial(1));
    }
}
```

Executar casos de probas básicos:

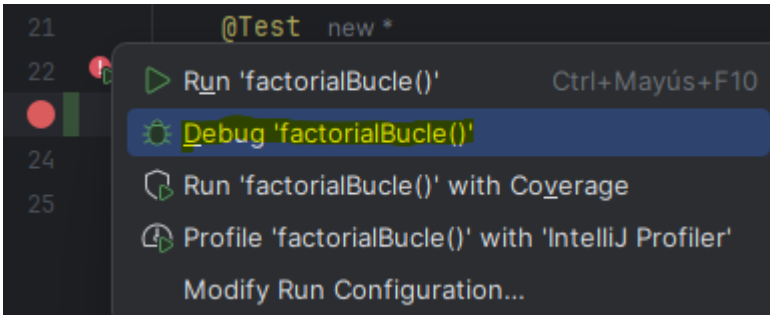
O *test* factorialBucle() do camiño 3 produce un fallo.

Depurar fallo:

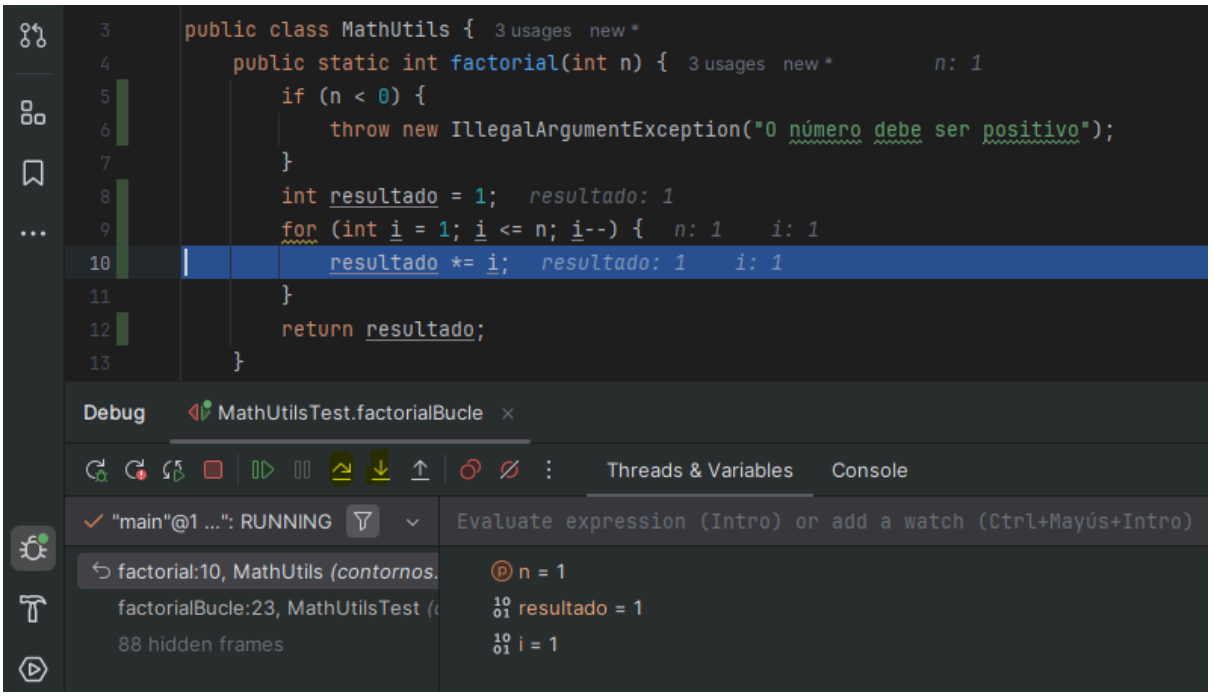
1. *Breakpoint* no test que falla:



2. *Debug* método:

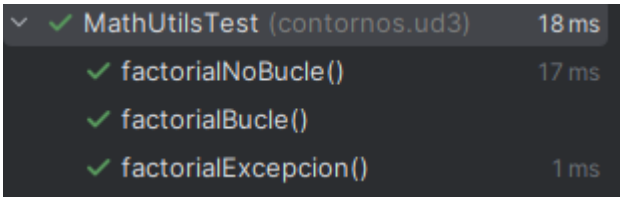


3. Executar paso a paso analizando o código e os valores das variables ata atopar o erro e corrixilo:



   O erro está no decremento do bucle (`i--`), que debería ser un incremento (`i++`).

4. Correxido o erro repetimos as probas.



Cobertura e probas adicionais

Podemos executar as probas con cobertura (*coverage*) e comprobamos que xa se cubre o 100% do código da clase MathUtils.

Element	Class, %	Method, %	Line, %	Branch, %
contornos.ud3	100% (2/2)	100% (4/4)	90% (9/10)	100% (4/4)
MathUtils	100% (1/1)	100% (1/1)	100% (6/6)	100% (4/4)
MathUtilsTest	100% (1/1)	100% (3/3)	75% (3/4)	100% (0/0)

Aínda que as probas executen todas as liñas de código da clase MathUtils, aínda non podemos asegurar que o método factorial() funcione correctamente. Podemos deseñar casos de probas adicionais, por exemplo, utilizando técnicas de probas de bucles, valores de factorial coñecidos, etc.

Casos de probas engadidos:

4. n = 2 Salida esperada: 2
5. n = 3 Salida Esperada: 6
6. n = 4 Salida Esperada: 24
7. n = 5 Salida Esperada: 120

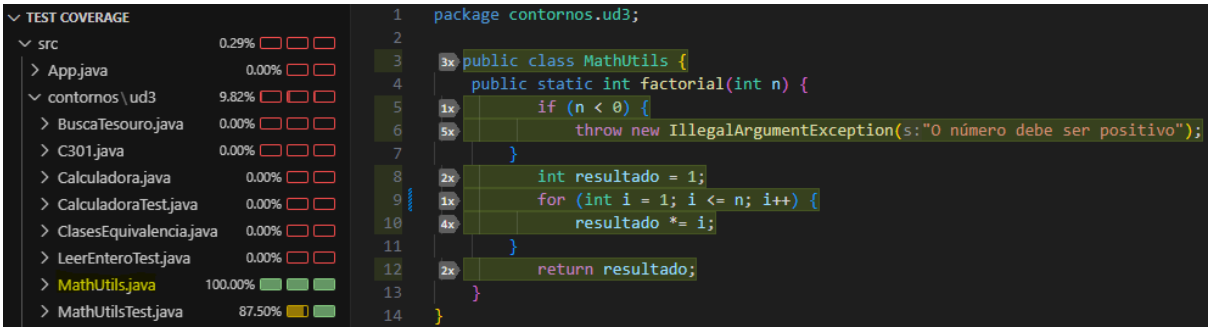
Para probalos todo cun só método podemos crear un test parametrizado:

```
@ParameterizedTest
@CsvSource({
        "2, 2",
        "3, 6",
        "4, 24",
        "5, 120",
})
public void factorialValores(int n, int valorEsperado) {
    assertEquals(valorEsperado, MathUtils.factorial(n));

}
```

```
✓ factorialValores(int, int)  37 ms
  ✓ [1] 2, 2  36 ms
  ✓ [2] 3, 6  1 ms
  ✓ [3] 4, 24
  ✓ [4] 5, 120
```

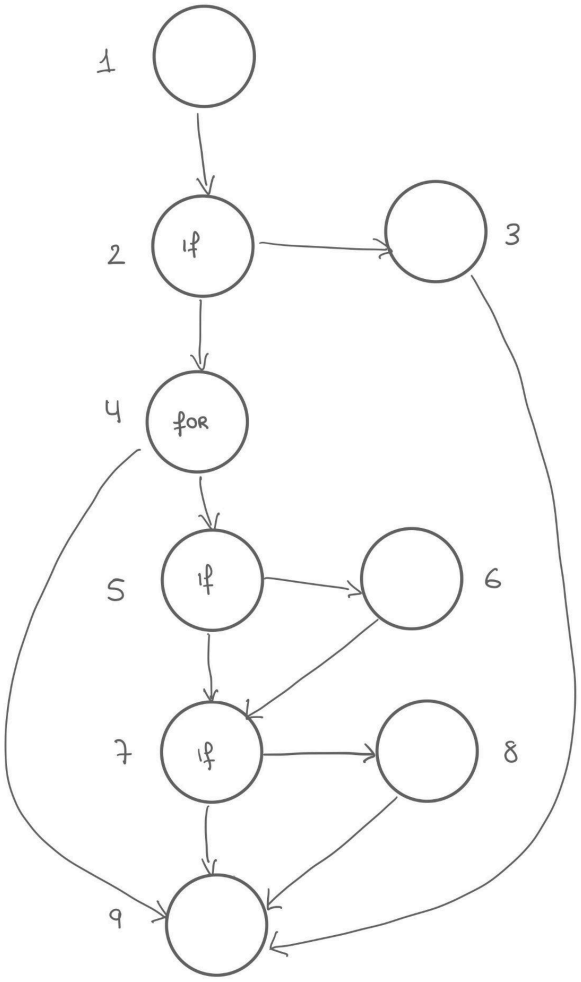
Por qué en vscode a cobertura das probas non chega ao 100% cando se executan todas as liñas do código?:

- Code coverage does not reach class declaration - Stack Overflow



5. Validación de contraseñas

Grafo



Complejidad ciclomática: 5

```
CAMINOS POSIBLES:
 1, 9 // no entra
 1, 2, 3 // sale en el if
 1, 2, 4, 5, 7, 9 //for: entra en los dos ifs
 1, 2, 4, 5, 6, 9 // for: sale en el primer if
 1, 2, 4, 5, 6, 7, 8, 9 // sale en el segundo if
```

5. Validación de contrasinais

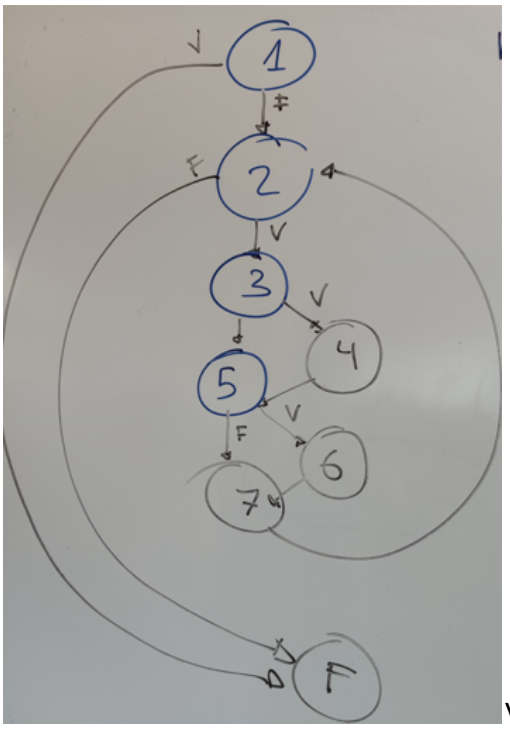
Crea unha clase `PasswordValidator` con un método `isValid(String password)`. A clave é válida se:

- Ten polo menos 8 caracteres.
- Contén polo menos unha letra maiúscula.
- Contén polo menos un número.

```java
public class PasswordValidator {
    public static boolean isValid(String password) {
        if (password.length() <= 8) {
            return false;
        }
        boolean hasUpperCase = false;
        boolean hasDigit = false;
        for (char c : password.toCharArray()) {
            if (Character.isLowerCase(c)) {
                hasUpperCase = true;
            }
            if (Character.isDigit(c)) {
                hasDigit = false;
            }
        }
        return hasUpperCase || hasDigit;
    }
}
```

PasswordValidator.java

Probas do Camiño Básico



V(G) = 5

Táboa de Clases de Equivalencia

Condición de Entrada	Clases Válidas	Clases No Válidas
Ten polo menos 8 caracteres.	password.length >= 8 (1)	password.length < 8 (4)
Contén polo menos unha letra maiúscula.	password inclúe maiúscula (2)	password NON inclúe maiúscula (5)
Contén polo menos un número.	password inclúe número (3)	password NON inclúe número (6)

Casos de proba con clases de equivalencia válidas

Entrada: password	Clases incluidas
"Passw0rd"	(1) (2) (3)

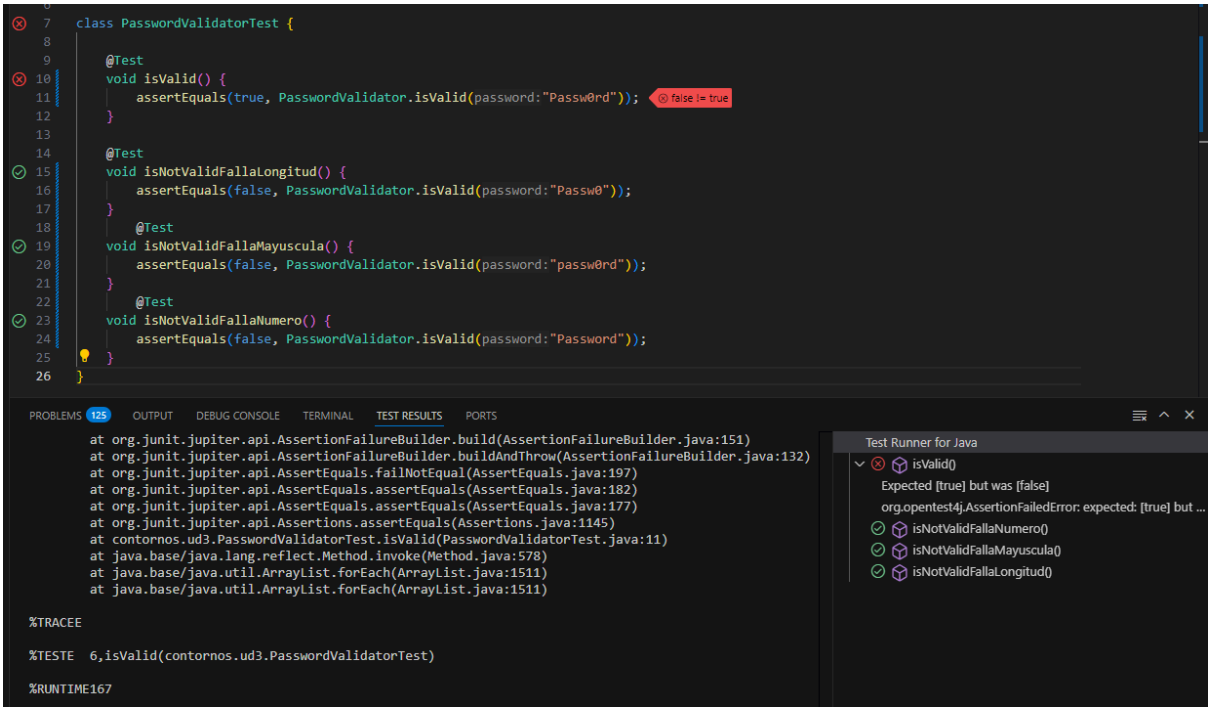
Casos de proba con clases de equivalencia non válidas

Entrada: password	Clases incluidas
"Passw0"	(4) (2) (3)
"passw0rd"	(1) (5) (3)
"Password"	(1) (2) (6)

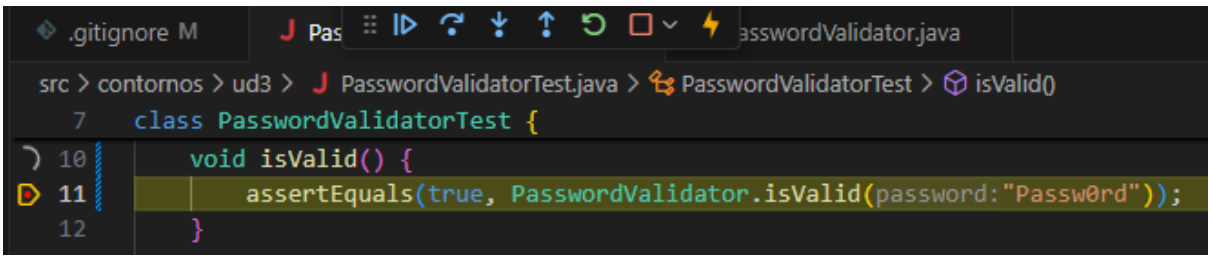
Implementar Casos de Proba

```
class PasswordValidatorTest {

    @Test
    void isValid() {
        assertEquals(true, PasswordValidator.isValid("Passw0rd"));
    }

    @Test
    void isNotValidFallaLongitud() {
        assertEquals(false, PasswordValidator.isValid("Passw0"));
    }
        @Test
    void isNotValidFallaMayuscula() {
        assertEquals(false, PasswordValidator.isValid("passw0rd"));
    }
        @Test
    void isNotValidFallaNumero() {
        assertEquals(false, PasswordValidator.isValid("Password"));
    }
}
```

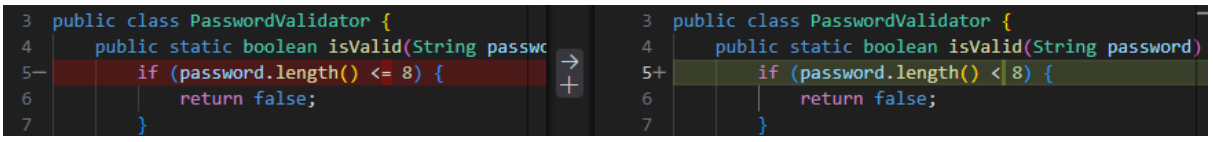
Executar Probas



Depurar



Atopado e corrixido erro na liña 5



Executar probas 2

Depurar 2

Atopado e corrixido erro na liña 11

```
10        for (char c : password.toCharArray()) {
11-           if (Character.isLowerCase(c)) {
12                hasUpperCase = true;
13            }
```

```
10        for (char c : password.toCharArray()) {
11+           if (Character.isUpperCase(c)) {
12                hasUpperCase = true;
13            }
```

Executar probas 3

```
7   class PasswordValidatorTest {
8
9       @Test
10      void isValid() {
11          assertEquals(true, PasswordValidator.isValid(password:"Passw0rd"));
12      }
13
14      @Test
15      void isNotValidFallaLongitud() {
16          assertEquals(false, PasswordValidator.isValid(password:"Passw0"));
17      }
18          @Test
19      void isNotValidFallaMayuscula() {
20          assertEquals(false, PasswordValidator.isValid(password:"passw0rd"));
21      }
22          @Test
23      void isNotValidFallaNumero() {
24          assertEquals(false, PasswordValidator.isValid(password:"Password")); ⊗ true != false
25      }
26  }
```

Depurar 3

Atopado e corrixido erro na liña 15

```
14            if (Character.isDigit(c)) {
15-               hasDigit = false;
16            }
```

```
14            if (Character.isDigit(c)) {
15+               hasDigit = true;
16            }
```

Executar probas 4

```
7   class PasswordValidatorTest {
8
9       @Test
10      void isValid() {
11          assertEquals(true, PasswordValidator.isValid(password:"Passw0rd"));
12      }
13
14      @Test
15      void isNotValidFallaLongitud() {
16          assertEquals(false, PasswordValidator.isValid(password:"Passw0"));
17      }
18          @Test
19      void isNotValidFallaMayuscula() {
20          assertEquals(false, PasswordValidator.isValid(password:"passw0rd")); ⊗ true != false
21      }
22          @Test
23      void isNotValidFallaNumero() {
24          assertEquals(false, PasswordValidator.isValid(password:"Password")); ⊗ true != false
25      }
26  }
```

Depurar 4

Atopado e corrixido erro na liña 18

```
17        }
18-       return hasUpperCase || hasDigit;
19    }
```

```
17        }
18+       return hasUpperCase && hasDigit;
19    }
```

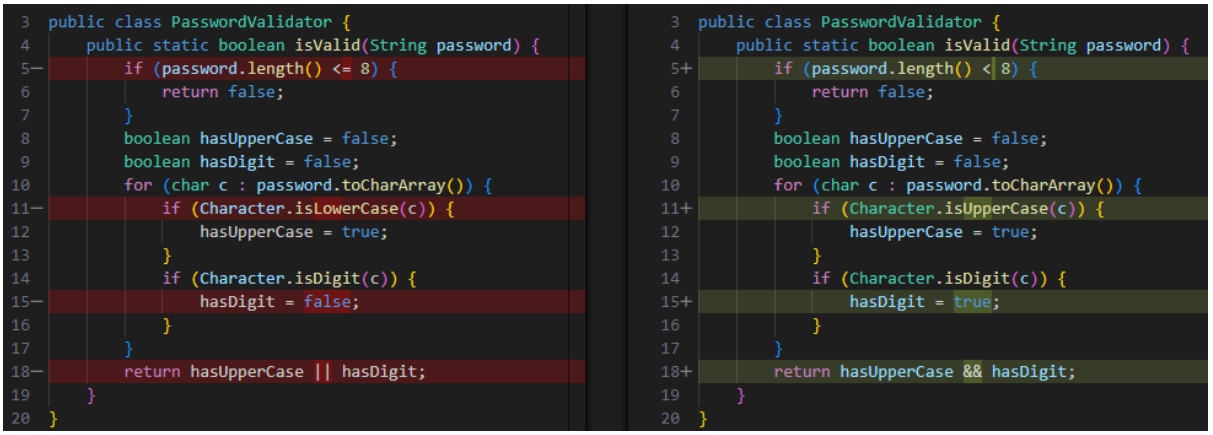
Executar probas 5

```java
class PasswordValidatorTest {

    @Test
    void isValid() {
        assertEquals(true, PasswordValidator.isValid(password:"Passw0rd"));
    }

    @Test
    void isNotValidFallaLongitud() {
        assertEquals(false, PasswordValidator.isValid(password:"Passw0"));
    }
        @Test
    void isNotValidFallaMayuscula() {
        assertEquals(false, PasswordValidator.isValid(password:"passw0rd"));
    }
        @Test
    void isNotValidFallaNumero() {
        assertEquals(false, PasswordValidator.isValid(password:"Password"));
    }
}
```

Resumo das actividades de proba

- Probas executadas 5 veces

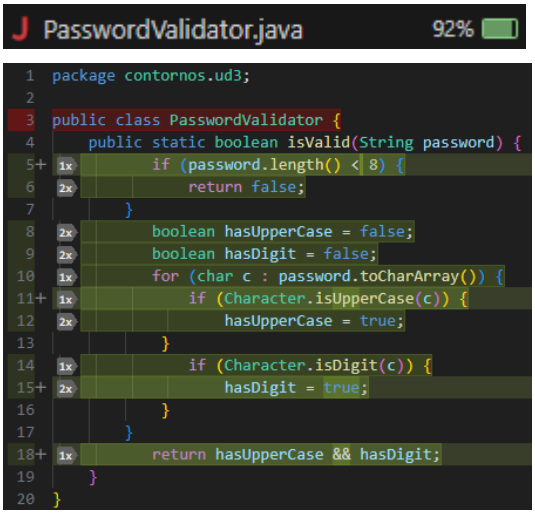
- 4 Erros atopados e corrixidos



```
3  public class PasswordValidator {
4      public static boolean isValid(String password) {
5-         if (password.length() <= 8) {
6              return false;
7          }
8          boolean hasUpperCase = false;
9          boolean hasDigit = false;
10         for (char c : password.toCharArray()) {
11-            if (Character.isLowerCase(c)) {
12                 hasUpperCase = true;
13             }
14             if (Character.isDigit(c)) {
15-                hasDigit = false;
16             }
17         }
18-        return hasUpperCase || hasDigit;
19     }
20 }
```

```
3  public class PasswordValidator {
4      public static boolean isValid(String password) {
5+         if (password.length() < 8) {
6              return false;
7          }
8          boolean hasUpperCase = false;
9          boolean hasDigit = false;
10         for (char c : password.toCharArray()) {
11+            if (Character.isUpperCase(c)) {
12                 hasUpperCase = true;
13             }
14             if (Character.isDigit(c)) {
15+                hasDigit = true;
16             }
17         }
18+        return hasUpperCase && hasDigit;
19     }
20 }
```

- Cobertura completa (agás instanciación da clase)



PasswordValidator.java 92%

```
1  package contornos.ud3;
2
3  public class PasswordValidator {
4      public static boolean isValid(String password) {
5+ 1x         if (password.length() < 8) {
6  2x             return false;
7          }
8  2x          boolean hasUpperCase = false;
9  2x          boolean hasDigit = false;
10 1x          for (char c : password.toCharArray()) {
11+1x              if (Character.isUpperCase(c)) {
12 2x                  hasUpperCase = true;
13             }
14 1x              if (Character.isDigit(c)) {
15+2x                  hasDigit = true;
16             }
17         }
18+1x      return hasUpperCase && hasDigit;
19     }
20 }
```