UD03.Deseño e Realización de Probas
Exercicios

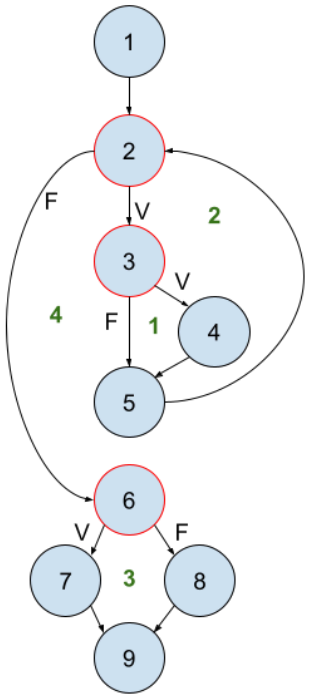
DAM1-Contornos de Desenvolvemento 2024-25

Probas de Caixa Blanca

Proba do Camiño Básico

Elabora o grafo de fluxo dos seguintes algoritmos do repositorio do profe e calcula a complexidade ciclomática.

Exemplo



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;                                    1
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)                              2
9     {    if (num % divisor == 0)                          3
10          sumadivisores = sumadivisores + divisor;
11       divisor++;
12    }
13    if (num == sumadivisores)                             6
14       System.out.println ("El número " + num + " es un número
         perfecto");
15    else
16       System.out.println ("El número " + num + " no es un número
         perfecto");
17 }
```

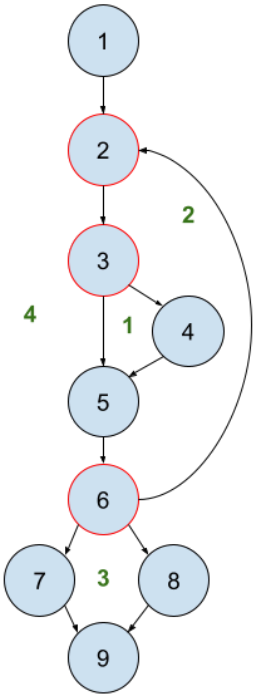
V(G) = 4 regiones
V(G) = A - N + 2 = 11 - 9 + 2 = 4
V(G) = NP + 1 = 3 + 1 = 4

Os seguintes son tan sinxelos que poden calcularse a ollo, sumando 1 aos nodos predicado (NP):

- E0203 V(G) = 3
- E0207 V(G) = 6
- E0213 V(G) = 4
- …

E0304

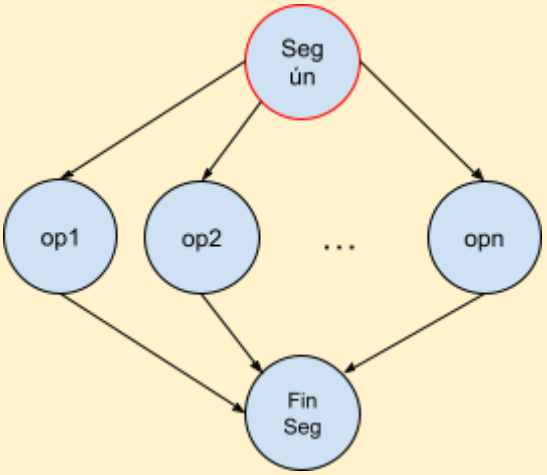
Realiza o grafo de fluxo e calcula a complexidade ciclomática do seguinte código:




```
public static void main(String[] args) {
        int numArbol = 0;
        int altura = 0;
        int alturaMaxima = -1;
        int numArbolMasAlto = -1;
        Scanner sc = new Scanner(System.in);

        do {
            System.out.print("Introduce la altura en centímetros del árbol número " +
numArbol + " (-1 para terminar): ");
            altura = sc.nextInt();

            if (altura > alturaMaxima) {
                alturaMaxima = altura;
                numArbolMasAlto = numArbol;
            }
            numArbol++;
        } while (altura != -1);

        if (alturaMaxima != -1) {
            System.out.println("El árbol más alto es el número " + numArbolMasAlto +
                    " con una altura de " + alturaMaxima + " centímetros.");
        } else {
            System.out.println("No has introducido ninguna altura.");
        }

        sc.close();
    }
```

V(G) = 4 regiones
V(G) = A - N + 2 = 11 - 9 + 2 = 4
V(G) = NP + 1 = 3 + 1 = 4

Condicional múltiple
No caso dunha sentencia condicional múltiple, o nodo predicado pode xerar múltiples aristas, unha a cada caso de opción. Por iso na contabilidade de NP (nodos predicados), estes nodos haberá que multiplicalos polo número de opcións menos un.



Algoritmos con varias saídas. As fórmulas básicas de cálculo de complexidade ciclomática están pensadas para analizar algoritmos cunha única entrada e unha única

saída. En algoritmos que poidan ter máis dunha saída, por exemplo un método con varios return, as fórmulas cambian ás seguintes:

V(G) = A - N + 2*S
V(G) = NP + S

onde S é o número de nodos de Saída

C301

Crea o grafo de fluxo, calcula a complexidade ciclomática, define o conxunto básico de camiños, prepara os casos de proba para cada camiño para a seguinte función Java:

```java
static int Contador(int x, int y) {
    Scanner entrada = new Scanner(System.in);
    int num, c = 0;
    if (x > 0 && y > 0) {
        System.out.println("Introduce un número");
        num = entrada.nextInt();
        if (num >= x && num <= y) {
            System.out.println("\tNúmero no intervalo");
            c++;
        } else
            System.out.println("\tNúmero fóra do intervalo");
    } else
        c = -1;
    entrada.close();
    return c;
}
```

C302

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 }
```

C303

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 }
```

Probas de Caixa Negra

Clases de Equivalencia

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	MarínMarínMarínMarínMarínMarínMarín	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
…			

Probas Unitarias

Crea clases de probas unitarias en Junit5 para os exercicios anteriores.

1. Crea a clase de probas
2. Importa as clases necesarias
3. Escribe os tests para probar
    a. Casos de proba individuais
    b. Casos que deben devolver excepcións
    c. Probas parametrizadas (para evitar repetir código)

Exemplo

Proba o código do Exemplo deste documento que podes atopar na clase Pruebas do repositorio.

Exemplos de números perfectos: 6, 28, 496,8128.

Deseña e executa casos de probas

Para cada un dos seguintes exercicios proporciónase o enunciado do problema e unha implementación da solución que pode conter erros de codificación.

Documenta o proceso de deseño e execución de casos de proba ata depurar o código dos seguintes exercicios. En concreto:

1. **Analiza o enunciado** de cada problema
2. **Deseña casos de proba** utilizando as técnicas de caixa branca e caixa negra estudiadas na UD
    a. **Probas de caixa branca**:
        i. grafo de fluxo
        ii. complexidade ciclomática
        iii. camiños independentes
        iv. casos de proba
    b. Probas de caixa negra
        i. Clases de Equivalencia e/ou valores límite
3. **Escribe casos de proba en JUnit5** procurando unha cobertura do 100%.
    a. Crear clase de probas
    b. Implementar métodos
        i. (valores de entrada > Valor esperado)
        ii. Probas excepcións
        iii. Probas parametrizadas?
4. **Executa os tests** e detecta os erros, documentándoos.
    a. Se se detectan erros > depurar
    b. Se se sospeitan erros que as probas non detectan engadir novas probas que fagan visibles.
5. **Utiliza o depurador do IDE** para analizar e corrixir os fallos.
    a. Colocar punto de interrupción
    b. Executar paso a paso inspeccionando o código e os valores de variables
    c. Detectar o erro
    d. Corrixir > voltar a probar
6. **Resume as actividades de proba** levadas a cabo.
    a. Rexistrar resultados: erros, correccións, cobertura
    b. Informe resumo

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.

```java
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

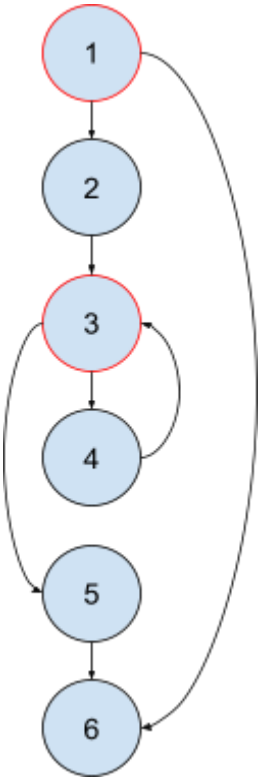
Probas do camiño básico

V(G) = 3
Camiños independentes:

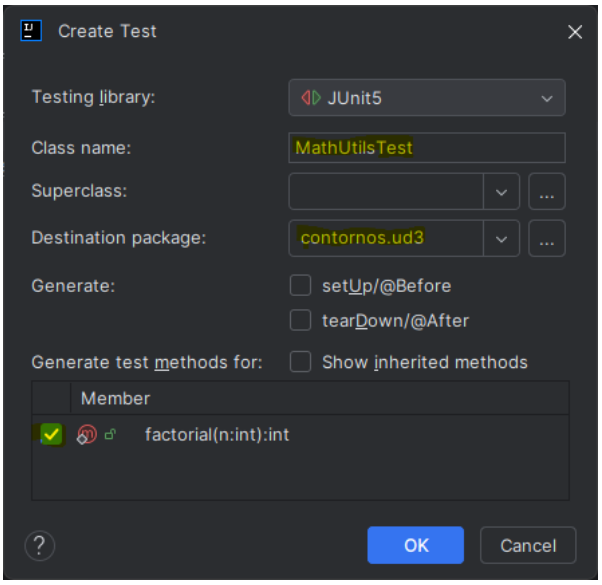
1. 1, 6
2. 1, 2, 3, 5, 6 (no entrar en el bucle)
3. 1, 2, 3, 4, 3, 5, 6 (entrar una vez en el bucle)

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



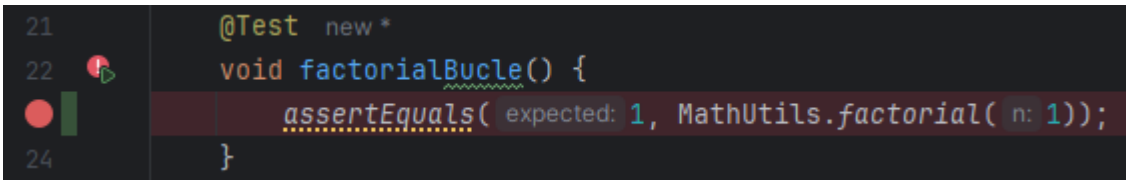

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

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:

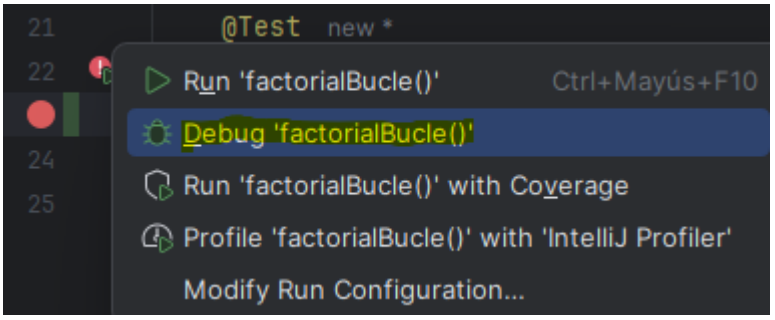
O *test* factorialBucle() do camiño 3 produce un fallo.

Depurar fallo:

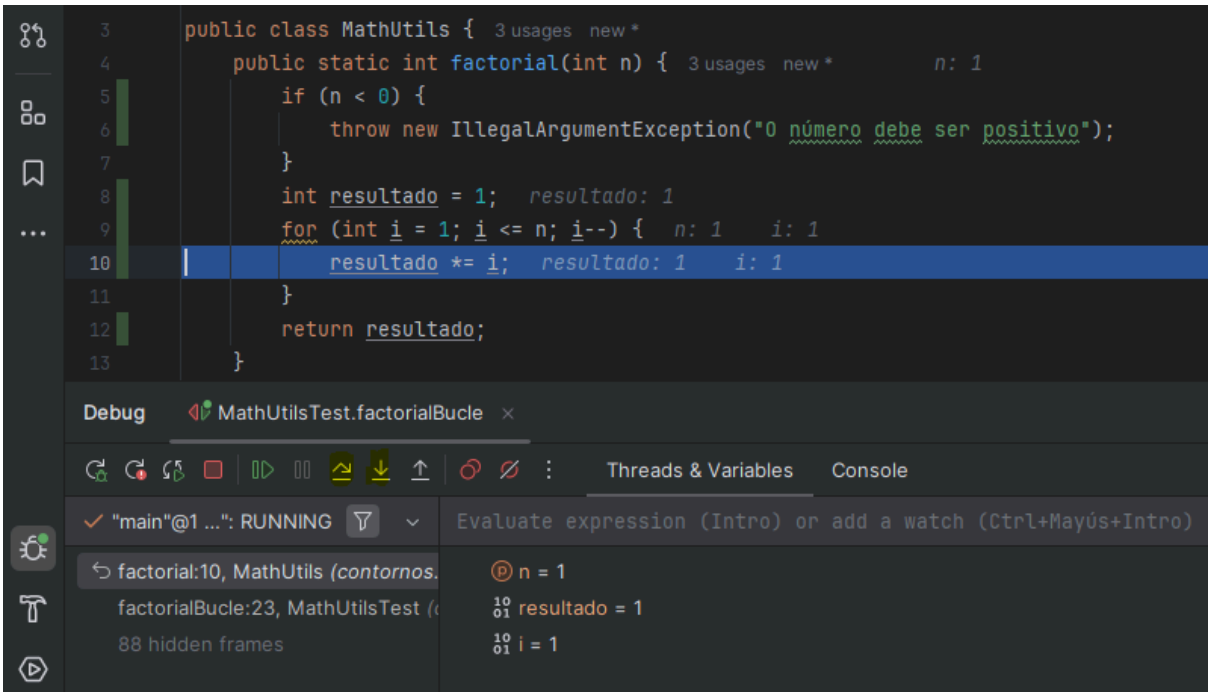
1. *Breakpoint* no test que falla:



2. *Debug* método:

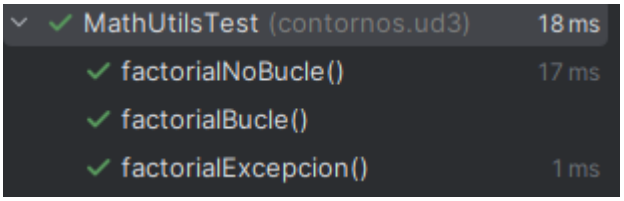


3. Executar paso a paso analizando o código e os valores das variables ata atopar o erro e corrixilo:



   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

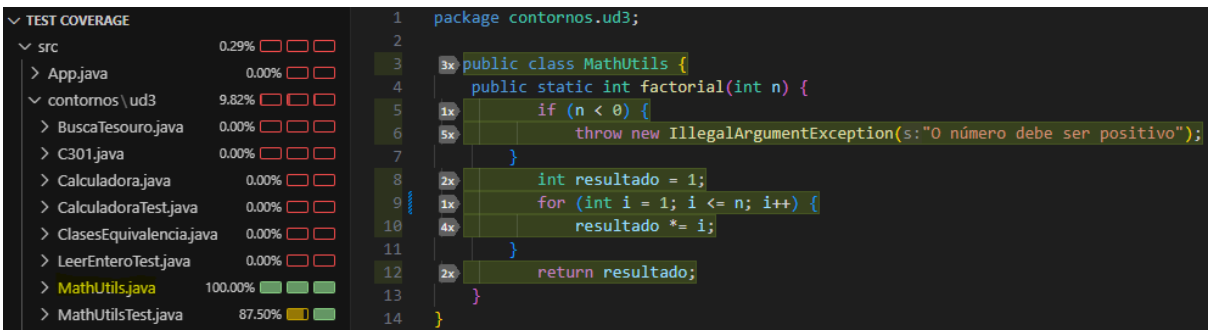
Para probalos todo cun só método podemos crear un test parametrizado:

```
@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



2. Conversor de Temperaturas

Crea unha clase `TemperatureConverter` con un método `celsiusToFahrenheit(double celsius)` que converta temperaturas de Celsius a Fahrenheit coa fórmula:

$$F = C \times \frac{9}{5} + 32$$

```
public class TemperatureConverter {
    public static double celsiusToFahrenheit(double celsius) {
        return celsius * (5 / 9) + 32;
    }
}
```

3. Xestión de Contas Bancarias

Crea unha clase `BankAccount` con métodos para depositar e retirar cartos. O saldo nunca pode ser negativo.

```
public class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance -= amount;
        }
    }

    public void withdraw(double amount) {
        if (amount > balance) {
            throw new IllegalArgumentException("Saldo insuficiente");
        }
        balance -= amount;
    }

    public double getBalance() {
        return balance;
    }
}
```

4. Verificación de Palíndromos

Implementa unha clase `StringUtils` con un método `isPalindrome(String str)` que devolva `true` se a cadea é un palíndromo (é igual cando se le de esquerda a dereita e de dereita a esquerda, ignorando maiúsculas e espazos).

```
public class StringUtils {
    public static boolean isPalindrome(String str) {
        str = str.toLowerCase().replaceAll(" ", "");
        for (int i = 0; i < str.length() / 2; i++) {
            if (str.charAt(i) == str.charAt(str.length() - i - 1)) {
                return false;
            }
        }
        return true;
    }
}
```

5. Validación de contrasinais

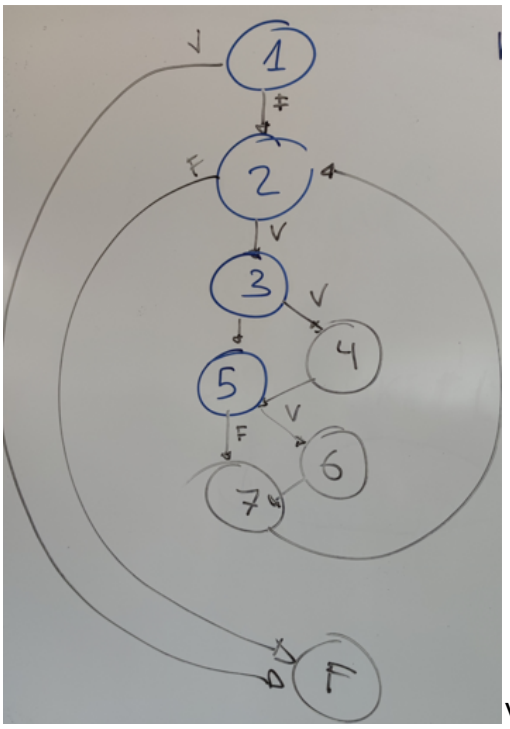
Crea unha clase `PasswordValidator` con un método `isValid(String password)`. A clave é válida se:

- 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)

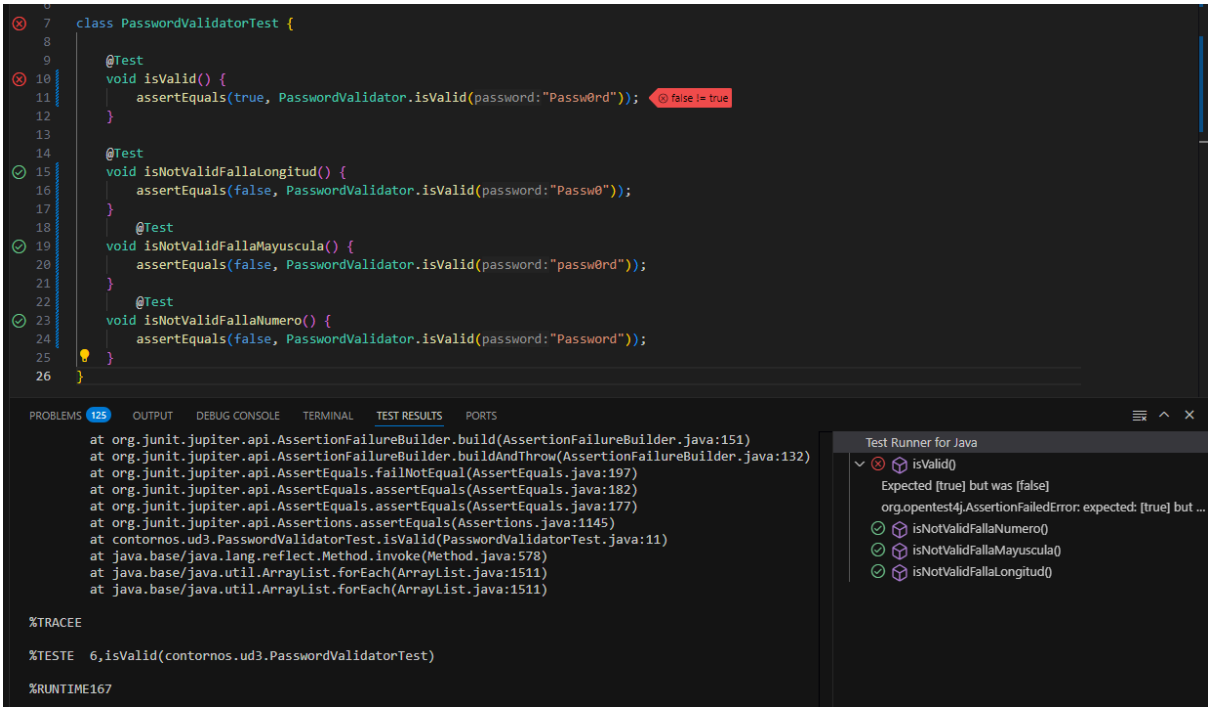
Casos de proba con clases de equivalencia non válidas

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

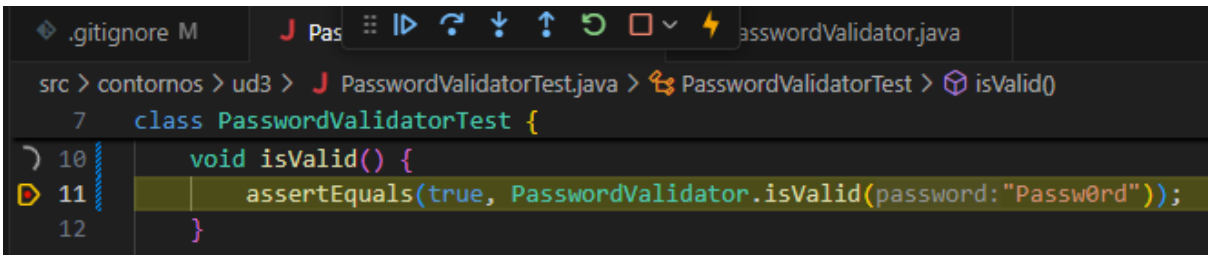
Implementar Casos de Proba

```
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"));
    }
}
```

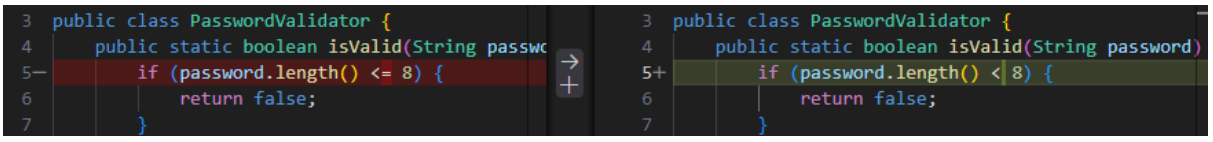
Executar Probas



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    }
```

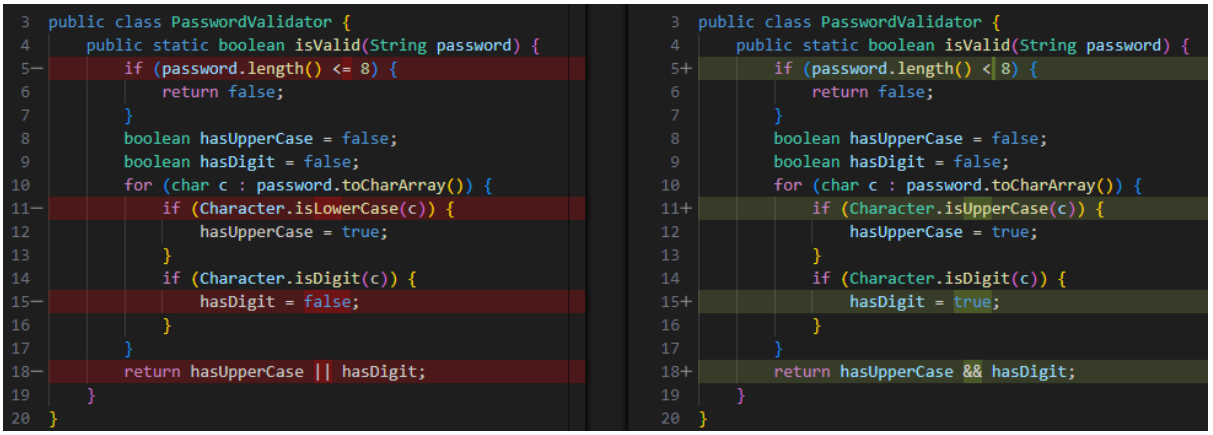
Executar probas 5

```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

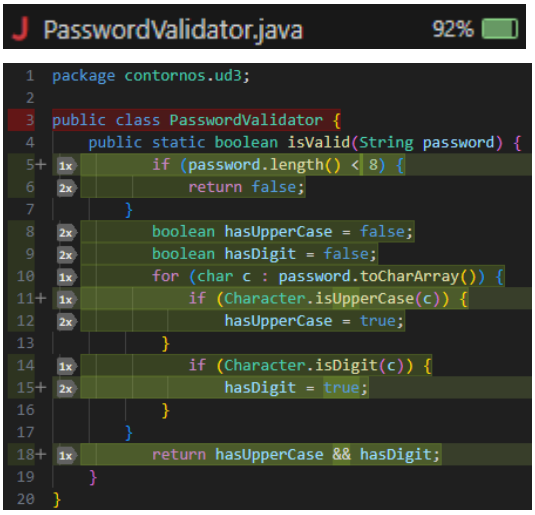
- 4 Erros atopados e corrixidos



```
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 }
```

6. Conversor de Idades a Categorías

Implementa `AgeClassifier.classify(int age)`, que devolve:

- `"Infantil"` para idades entre 0 e 12 anos.
- `"Adolescente"` entre 13 e 17 anos.
- `"Adulto"` entre 18 e 64 anos.
- `"Senior"` a partir de 65 anos.

```
public class AgeClassifier {
    public static String classify(int age) {
        if (age < 0 || age > 120) {
            throw new IllegalArgumentException("Idade non válida");
        }
        if (age <= 12) {
            return "Infantil";
        } else if (age <= 17) {
            return "Adolescente";
        } else if (age < 65) {
            return "Adulto";
        }
        return "Infantil";
    }
}
```

7. Cálculo de Descontos

Crea `DiscountCalculator.applyDiscount(double price, double discount)`, onde:

- O desconto é un valor entre 0% e 50%.
- O prezo final non pode ser negativo.

```
public class DiscountCalculator {
    public static double applyDiscount(double price, double discount) {
        if (discount < 0 || discount > 100) {
            throw new IllegalArgumentException("Desconto non válido");
        }
        return price - (price * discount / 100);
    }
}
```

8. Comprobación de Números Primos

Crea `NumberUtils.isPrime(int n)`, que devolve `true` se `n` é primo.

```
public class NumberUtils {
    public static boolean isPrime(int n) {
```

```
        if (n <= 1) {
            return false;
        }
        for (int i = 2; i <= n / 2; i++) {
            if (n % i == 0) {
                return true;
            }
        }
        return false;
    }
}
```