

Tema 3

Diseño y Realización de Pruebas:



Tema 3: Diseño y Realización de Pruebas:

Tema 3	1
Diseño y Realización de Pruebas:	1
Práctica 3.1: Explicar con tus propias palabras o con ejemplos los 7 principios de pruebas de software:	3
Práctica 3.2: Buscar herramientas para la depuración de código y para la realización de pruebas unitarias en el contorno de desarrollo para diferentes lenguajes:	4
Práctica 3.3: Detallar etapas de la integración incremental ascendente y de la descendente para el siguiente grupo de módulos:	4
Práctica 3.4: Depuración Básica (Netbeans):	5
Práctica 3.5: Depuración con inspección de variables, expresiones, métodos y modificando valores (Netbeans):	7
Práctica 3.6: Depurar inspeccionando la pila (Netbeans).	9
Práctica 3.7: Realización de modificaciones durante la depuración (Netbeans):	10
Práctica 3.8: Depuración hasta un punto de introducción (Netbeans):	10
Práctica 3.9: Representación de gráficos de flujo, calcular la complejidad diplomática de McCabe y obtención de los caminos:	12
Práctica 3.10: Definir clases de equivalencia, realizar análisis de los valores límites y la conjetura de errores:	16
Práctica 3.11: Elaborar casos de prueba:	17
Práctica 3.12: Generar y ejecutar pruebas en JUnit y documentar incidencias:	18
Práctica 3.13: Realiza el test de una clase que solo tenga el método siguiente probando la cadena “abcbca” y la letra ‘a’. Haz el análisis de cobertura con JaCoCo y añade los test precisos para obtener una cobertura completa del método:	19

Práctica 3.1: Explicar con tus propias palabras o con ejemplos los 7 principios de pruebas de software:

1. La ejecución de pruebas nos muestra la presencia de los fallos:

Este principio nos indica que para encontrar los fallos en un software debemos de realizar diversas pruebas en su búsqueda.

2. El “testing” exhaustivo es imposible:

Este principio recuerda la imposibilidad de comprobar todos los casos y escenarios a los que se puede enfrentar el software sometido a las pruebas.

3. La temprana verificación del sistema ahorra tiempo y dinero:

Este principio nos indica que la identificación y correcciones de errores temprana nos puede ayudar a evitar pérdidas de tiempo y dinero en caso de detectar dichos problemas una vez lanzado o ejecutado el software.

4. Concentración de defectos:

Este principio nos indica que al acumularse muchos errores juntos, estos pueden ocultar otros errores que pueden pasar desapercibidos luego de haber corregir alguno de ellos.

5. Tener precaución con la paradoja del pesticida:

Este principio nos recuerda que no es recomendable tener siempre la misma solución para resolver y buscar errores, debido a que si siempre se sigue el mismo proceso alguno de los errores podría no detectarse.

6. Las pruebas se deben adaptar al contexto del sistema que se va a verificar.

Este principio es como su nombre indica, las pruebas realizadas para la búsqueda de errores deben de estar sujetas al contexto que se se va a verificar, es decir debemos adaptarnos según el tipo de software.

7. El engaño de la ausencia de errores:

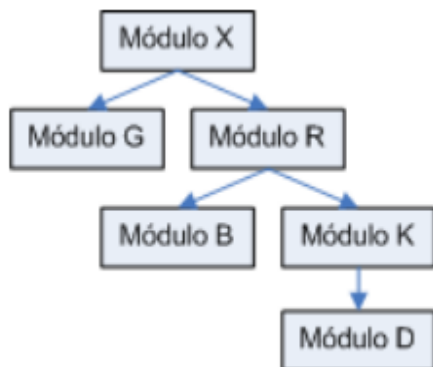
Este principio nos recuerda que porque nosotros no encontremos ningún error no significa que no exista ninguno, porque solo nos hemos

basado en nuestras pruebas y criterio, pudiendo encontrarse un error por otro usuario o desarrollador en cualquier momento.

Práctica 3.2: Buscar herramientas para la depuración de código y para la realización de pruebas unitarias en el contorno de desarrollo para diferentes lenguajes:

- [JUnit](#)
- [Cactus](#)
- [EasyMock](#)
- [Mockito](#)
- [MockEjb](#)
- [Spring Test](#)
- [Jetty](#)
- [Dumbster](#)

Práctica 3.3: Detallar etapas de la integración incremental ascendente y de la descendente para el siguiente grupo de módulos:



Integración Ascendente: Se realiza una prueba unitaria del Módulo D con el Módulo K y de forma simultánea se realizan también pruebas unitarias el Módulo B con el Módulo R y el Módulo G con el módulo X.


Integración Descendente: Se realizan pruebas unitarias del Módulo X con ficticios de los Módulos G y R, y cuando funcionen correctamente se implementan.

A continuación se realizan pruebas unitarias con el Módulo R y los ficticios del Módulo B y K, y cuando funcionen correctamente se implementan y finalmente se realiza una prueba unitario del Módulo K (una vez comprobado e implementado) con el ficticio del Módulo D, y en cuanto la prueba sea correcta se implementará el Módulo D real.

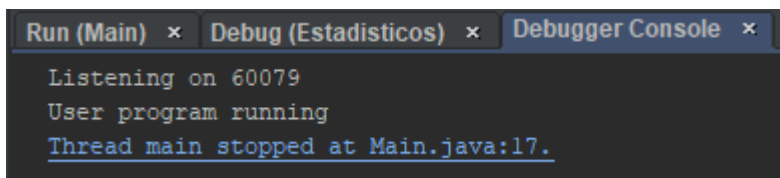
Práctica 3.4: Depuración Básica (Netbeans):

1) Iniciar una depuración paso a paso del método Main, tecleando los valores m=5 e n=2 sin ejecutar línea a línea ninguno de los dos métodos invocados.

```
3
4 public class Main {
5     public static void main(String[] args) {
6         System.out.print("\nCALCULOS ESTADÍSTICOS\n");
7         Scanner teclado = new Scanner(System.in);
8         boolean error;
9         int m, n;
10        do {
11            try {
12                error = false;
13                System.out.println("Teclee m (>= 0): ");
14                m = Integer.parseInt(teclado.nextLine());
15                System.out.println("Teclee n (>= 0 y <= m): ");
16                n = Integer.parseInt(teclado.nextLine());
17                Estadisticos es = new Estadisticos(m, n);
```

Para realizar la depuración sin ejecutar línea a línea ninguno de los métodos invocados simplemente dejaremos nuestro cursor en la línea 17 y pulsaremos F4 o pulsamos el icono.  Run to Cursor F4

Una vez terminado podemos ver en la Debugger Console que el programa solo se ha ejecutado hasta la línea 17 (Main.java:17).



```
Run (Main) x  Debug (Estadisticos) x  Debugger Console x
Listening on 60079
User program running
Thread main stopped at Main.java:17.
```

2). Repetir el paso anterior pero ejecutando paso a paso solamente el método factorial(5).

Finalizar la depuración y ver el resultado final.

Realizamos la depuración de la misma forma que antes, pero esta vez hasta la línea 18, y una vez en la línea 18 pulsamos F7 o pulsamos el icono.

 Step Into F7

```
public double factorial(int x) throws Exception {
    double resultado = 1;
    for (int i = 2; i <= x; i++) {
        resultado *= i;
    }
    return resultado;
}
```

Al pulsar F7 o el icono nos llevará a donde se encuentra el factorial.

Finalizamos la depuración y el resultado final es el siguiente:

```
CALCULOS ESTADÍSTICOS
Teclee m (>= 0):
5
Teclee n (>= 0 y <= m):
2
```

3). Repetir el paso anterior pero ejecutando ahora paso a paso solamente el método variaciones(). Finalizar la depuración y ver el resultado final.

Volvemos a realizar la depuración de la misma forma pero en este caso hasta la línea 22. Pero al haber un System.out.printf(), si le damos a F7 entra en los archivos de java sobre dicho comando, para acceder a variaciones() debemos de pulsar Shift/Mayus + F7.

Step Into Next Method Shift-F7

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

Terminamos la depuración y el resultado sería el siguiente:

```
CALCULOS ESTADÍSTICOS
Teclee m (>= 0):
5
Teclee n (>= 0 y <= m):
2
Permutaciones(2) = 2,000000
Permutaciones(5) = 120,000000
```

4) Colocar el cursor en la línea 21 de Main.java e iniciar una depuración hasta el cursor(teclear los valores m=15 y n=3). Finalizar esta depuración y ver el resultado final.

Seleccionamos la línea 21, pulsamos F4, ingresamos los valores y terminamos la depuración con el siguiente resultado final:

```
CALCULOS ESTADÍSTICOS
Teclee m (>= 0):
15
Teclee n (>= 0 y <= m):
3
Permutaciones(3) = 6,000000
```

Práctica 3.5: Depuración con inspección de variables, expresiones, métodos y modificando valores (Netbeans):

1) Depurar para poder ver el valor del retorno de los métodos factorial en la línea 26 de Estadisticos.java.

Seleccionamos la línea 26, pulsamos F4, introducimos los valores y tras estos dos pasos vamos a Window y seleccionamos Variables(Debugging), donde encontraremos los valores del retorno de los métodos factorial:

Name	Type	Value
<Enter new watch>		
▶ this	Estadisticos	#420
◆ combi	double	10.0

2) Depurar para poder inspeccionar el valor de la variable local y del método factorial(5) durante una sesión de depuración.

Se selecciona la línea donde se encuentra el factorial al que se le ha dado valor 5, en este caso m, se pulsa F4, se dan los valores y una vez realizado estos pasos para inspeccionar los valores debemos ir a Window/Variables(Debugging) y nos mostrará lo siguiente:

Name	Type	Value
<Enter new watch>		
▶ Static		
▶ ◆ args	String[]	#420(length=0)
▶ ◆ teclado	Scanner	#421
◆ error	boolean	false
◆ m	int	5
◆ n	int	2
▶ ◆ es	Estadisticos	#422
◆ factN	double	2.0

3) Depurar para que en tiempo de depuración se pueda modificar el valor de la variable local i y el método factorial(5) y ver los cambios realizados en las variables locales.

Seleccionamos la línea 19 de Estadisticos.java, pulsamos F4, introducimos también los mismos valores anteriores y vamos a Window/Variables(Debugging), pulsamos F8 para entrar en el bucle y podremos ver en las variables el valor de i.

Name	Type	Value
<Enter new watch>		
▶ this	Estadisticos	#397
▶ x	int	2
▶ resultado	double	1.0
▶ i	int	2

Una vez salido del bucle pulsamos F8 otra vez y veremos el factorial:

Name	Type	Value
<Enter new watch>		
▶ Static		
▶ args	String[]	#399(length=0)
▶ teclado	Scanner	#400
▶ error	boolean	false
▶ m	int	5
▶ n	int	2
▶ es	Estadisticos	#397
▶ factN	double	2.0

4) Depurar para que durante una sesión de depuración se pueda ver el valor de la expresión m-n.

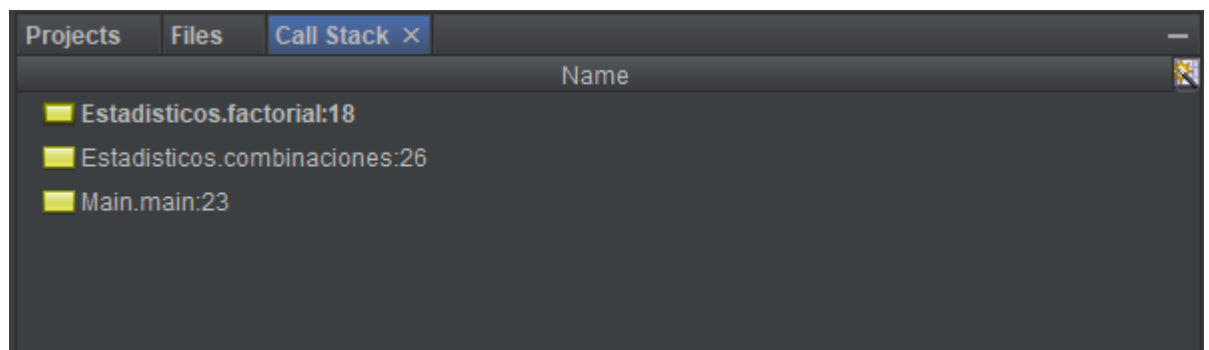
Lo primero que debemos hacer es seleccionar la línea 26, pulsar F4, insertar los valores de m y n (5(m) y 2(n) en este caso), pulsar Shift/Mayús + F8 hasta llegar al factorial que contiene la expresión m-n. Una vez llegados al factorial que contiene la expresión abrimos la ventana de variables en Window/Variables(Debugging) y desplegamos la pestaña que pone Arguments y ahí encontraremos la operación con su resultado:



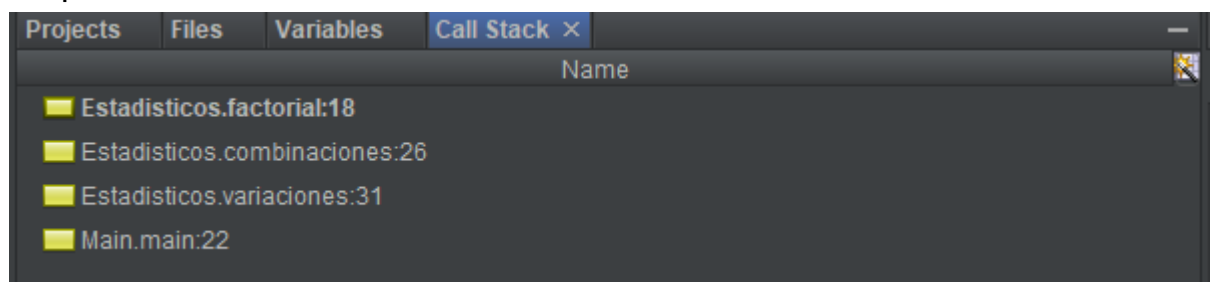
Práctica 3.6: Depurar inspeccionando la pila (Netbeans).

- **Depurar archivo Estadísticos para que contenga 3 llamadas en la pila y 4 a continuación:**

Para contener 3 llamadas a la pila seleccionamos la línea 26, pulsamos F4, introducimos los valores necesarios de m y n y abrimos en Window la ventana Call Stack (En Debugging) y ahí veremos nuestras 3 llamadas:

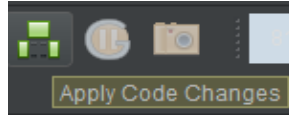


Para obtener 4 llamadas en la pila, debemos de realizar todos los pasos anteriores y una vez realizados pulsar F7(Step into) dos veces y así podremos ver nuestras 3 llamadas:



Práctica 3.7: Realización de modificaciones durante la depuración (Netbeans):

Ejecutamos el código con F4 marcando la línea 14 del Main.java, editamos en Estadisticos.java el primer if añadiendo `|| m > 190` y le damos al icono de Apply Code Changes:



Una vez realizados los cambios ejecutamos el código introduciendo para m un valor superior a 190 para apreciar los cambios:

```
CALCULOS ESTADÍSTICOS
Teclee m (>= 0):
195
Teclee n (>= 0 y <= m):
3
class java.lang.Exception->Error. Los argumentos tienen que ser >=0 y el primero >= que el segundo
Teclee m (>= 0):
|
```

Práctica 3.8: Depuración hasta un punto de introducción (Netbeans):

- 1) Definir un punto de interrupción en la línea 21 de Main.java. Ejecutar una depuración para ver la salida de resultados hasta ese momento. Finalizar la depuración. Desactivar ese punto de interrupción.

Para definir un punto de interrupción debemos de hacer click en el número 21 de la línea de código y veremos como la línea se pone de color rojo con un cuadrado en lugar de el número (Si la línea de código no tuviese ningún argumento no se podría seleccionar) y le damos a Debug Project en la pestaña Debug o pulsamos Ctrl+F5.

Ingresamos los datos para que se ejecute el código y se parará en la línea 21. Para desactivar este punto de interrupción simplemente debemos hacerle click en el cuadrado rojo que se había generado substituyendo al número de línea.

- 2) Definir un punto de interrupción que pare la depuración cada vez que se salga del método factorial() de Estadisticos.java. Ejecutar la depuración y cada vez que se salga del método, ver como varía la pila en la ventana de pila de llamadas, y ver los valores de los elementos x y resultado en la ventana de elementos observados. Eliminar ese punto de interrupción.**

Seleccionaremos como punto de interrupción la línea final del método factorial() que sería la 22, abrimos la ventana Call Stack y la de las variables desde Window/Debugging, y ejecutamos el debug con Ctrl+F5.

Una vez ingresados los valores y ejecutado hasta la línea marcada podemos ver como varían las variables y las Pilas dándole al botón de continuar debug o también pulsando F5.

- 3) Definir un punto de interrupción que pare la ejecución para los valores de m=4 e n=2, en la línea 22 de Estadisticos.java.**

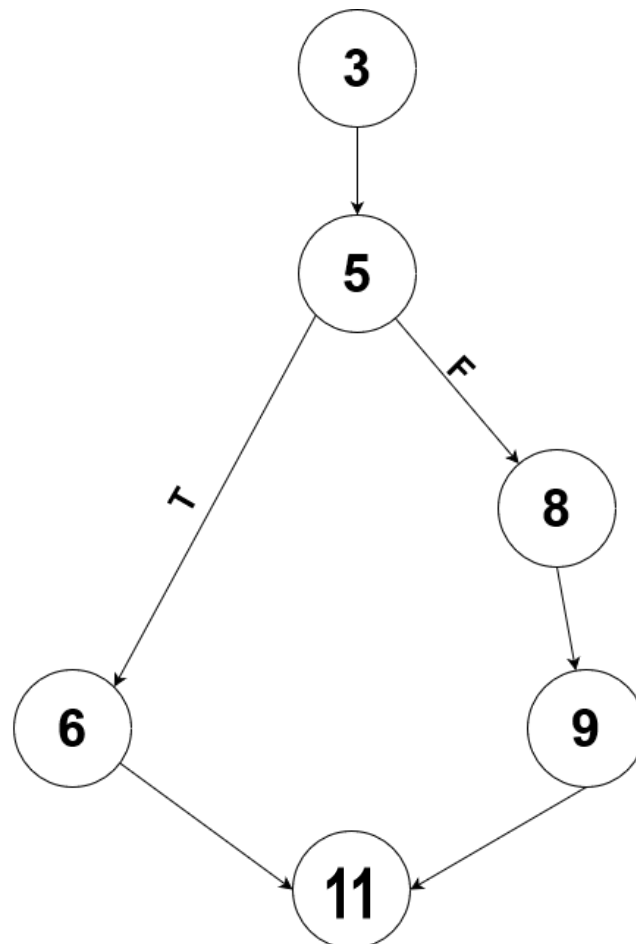
Para definir este punto de interrupción seleccionaremos la línea 22 de Estadisticos.java, iniciaremos el debug con Ctrl+F5 e ingresaremos los datos m=4 y n=2.

- 4) Eliminar todos los puntos de interrupción.**

Para eliminar los puntos de interrupción simplemente debemos de eliminar el inicialmente marcado y continuar con la depuración o terminarla.

Práctica 3.9: Representación de gráficos de flujo, calcular la complejidad diplomática de McCabe y obtención de los caminos:

1) Método calcularDivision del proyecto proyecto_division.



```
package proyecto_division;
public class Division {
    public float calcularDivision(float dividendo, float divisor) throws Exception 3
    {
        if (divisor == 0) { 5
            throw (new Exception("Error. El divisor no puede ser 0.")); 6
        }
        float resultado = dividendo / divisor; 8
        return resultado; 9
    }
}
```

11

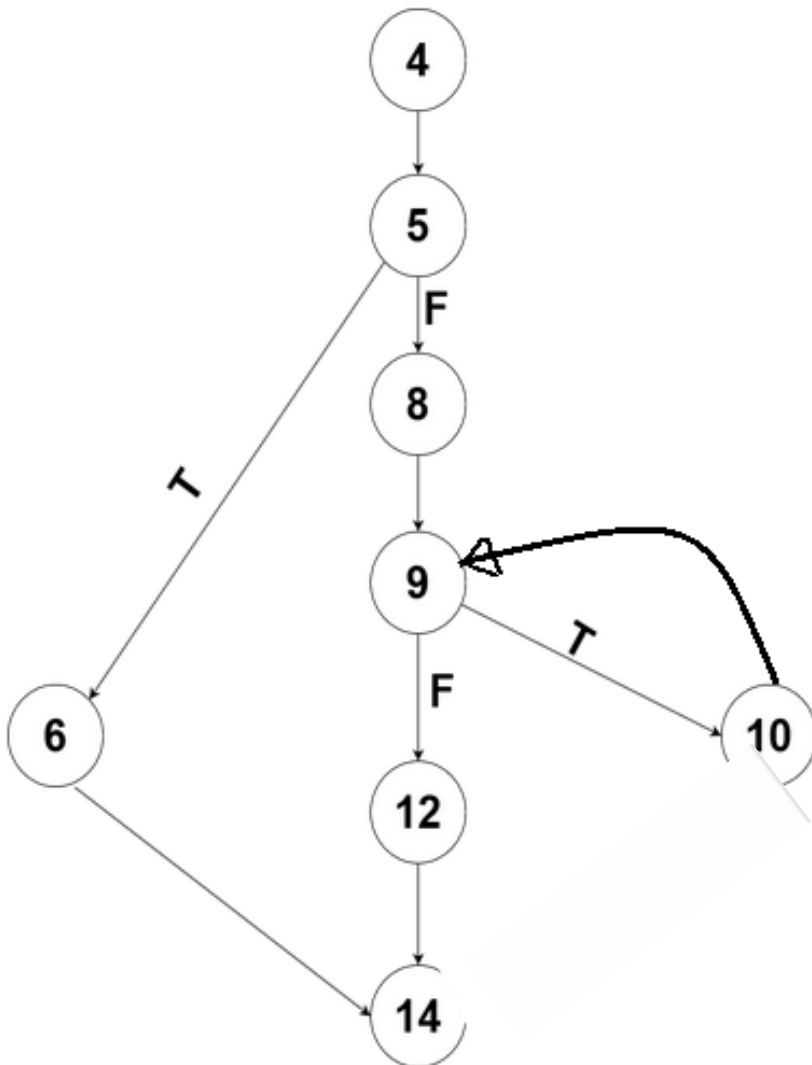
Complejidad diplomática de McCabe:

$$V(G) = 6 - 6 + 2 = 2$$

Caminos:

3-5-6-11 | 3-5-8-9-11

2) Método factorial del proyecto proyecto_factorial. (Las líneas de código se corresponden con los números del Grafo.)



**Complejidad
diplomática de
McCabe:**

$$V(G) = 9 - 8 + 2 = 3$$

Caminos:

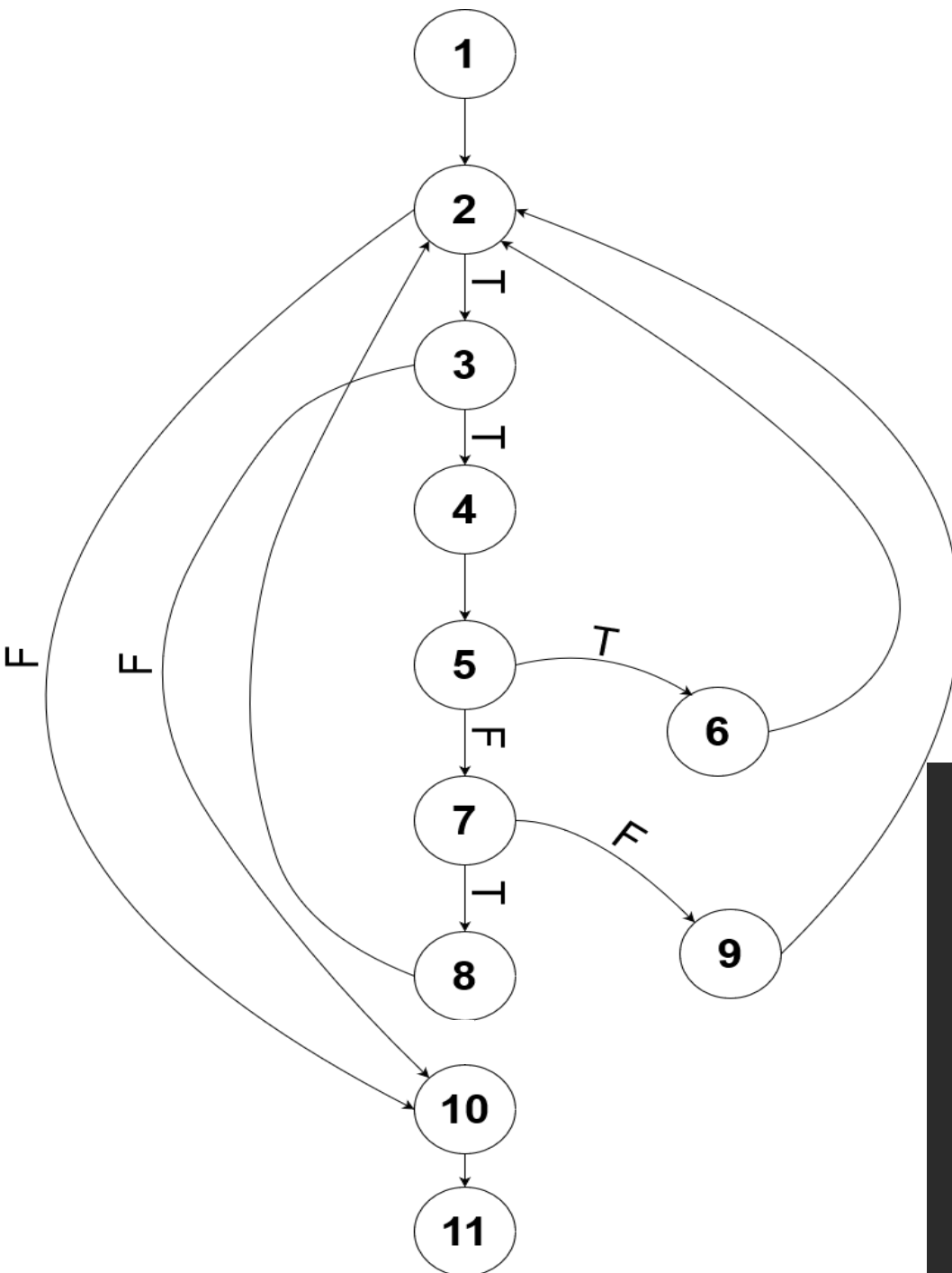
4-5-8-9-12-14

4-5-8-9-10-9-12-14

4-5-6-14

```
1 package proyecto_factorial;
2
3 public class Factorial {
4     public float factorial(byte n) throws Exception {
5         if (n < 0) {
6             throw new Exception("Error. El número tiene que ser >=0");
7         }
8         float resultado = 1;
9         for (int i = 2; i <= n; i++) {
10             resultado *= i;
11         }
12         return resultado;
13     }
14 }
```

3) Método busca del proyecto proyecto_arrays.



Caminos:

1-2-3-4-5-7-8-10-11

1-2-3-4-5-7-9...

1-2-3-4-5-6...

1-2-3-10-11

1-2-10-11

```

package buscarcaracter;

public class OperacionsArrays {

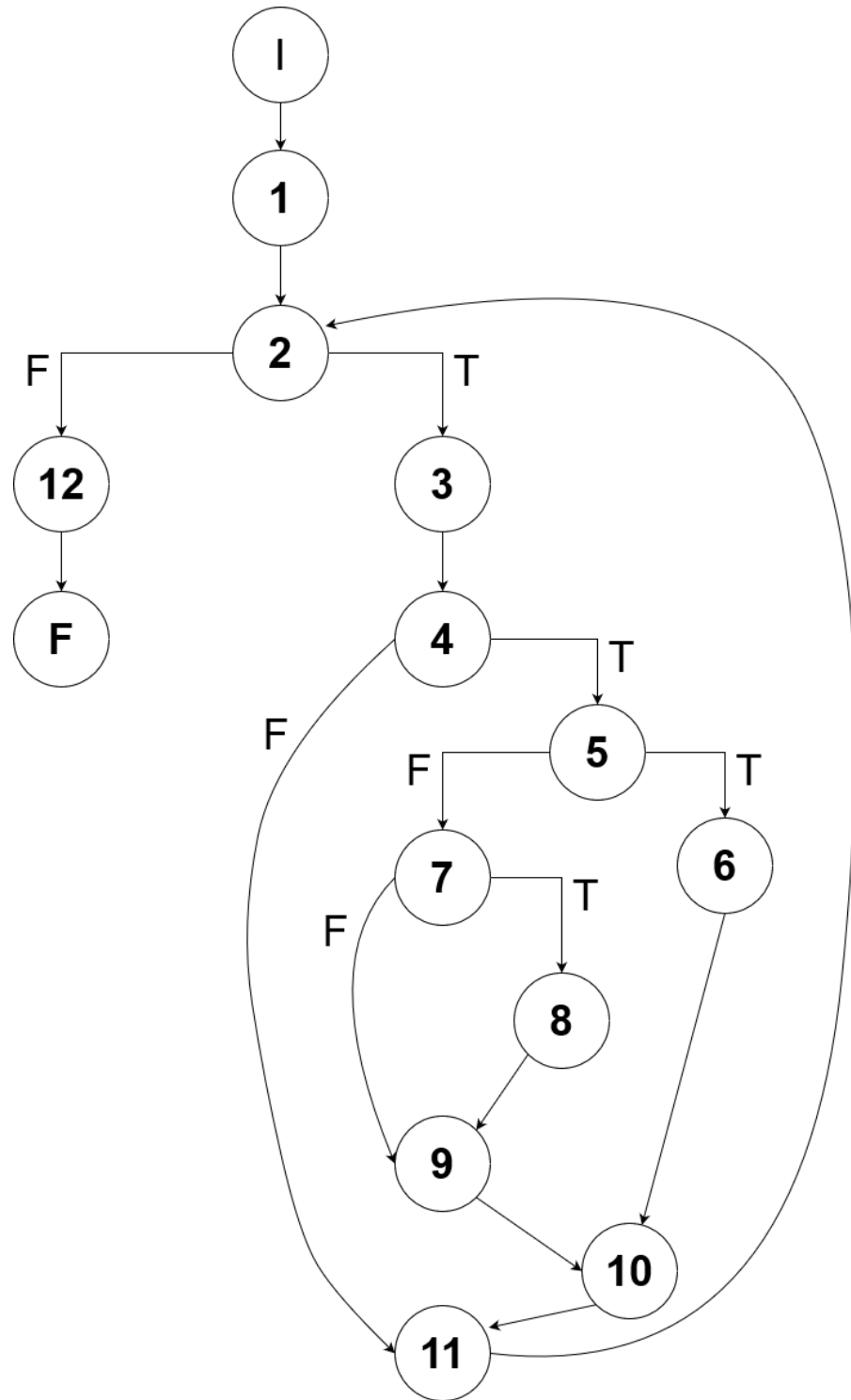
    public boolean busca(char c, char[] v) {
        int a, z, m;
        a = 0;
        z = v.length - 1;
        boolean resultado=false;
        while (a <= z && resultado==false) {
            m = (a + z) / 2;
            if (v[m] == c) {
                resultado=true;
            }
            else {
                if (v[m] < c) {
                    a = m + 1;
                }
                else {
                    z = m - 1;
                }
            }
        }
        return resultado;
    }
}
    
```

Handwritten annotations in red: }1, 2/3, 4, 5, 6, 7, 8, 9, 10, 11.

Complejidad diplomática de McCabe:

$$V(G) = 15 - 11 + 2 = 6$$

4) Método obtenerAcronimo del proyecto proyecto_acronimos.



```

package proyecto_acronimos;

public class Acronimos {

    public String obtenerAcronimo(String cadena){
        String resultado="";
        char caracter;
        int n=cadena.length();
        for(int i=0;i<n;i++){
            caracter=cadena.charAt(i);
            if(caracter!=' '){
                if (i==0){
                    resultado=resultado+caracter+'.';
                }
                else{
                    if(cadena.charAt(i-1)==' '){
                        resultado=resultado+caracter+'.';
                    }
                }
            }
        }
        return resultado;
    }
}

```

Camino:

I-1-2-3-4-5-6-10-11...
 I-1-2-3-4-5-7-8-9-10-11...
 I-1-2-3-4-5-7-9-10-11...
 I-1-2-12-F

Complejidad diplomática de McCabe:

$$V(G) = 17 - 14 + 2 = 5$$

Práctica 3.10: Definir clases de equivalencia, realizar análisis de los valores límites y la conjetura de errores:

Parte 1) Método calcularDivision del proyecto proyecto_division.

Parte 2) Método factorial del proyecto proyecto_factorial.

Parte 3) Método busca del proyecto proyecto_arrays.

Parte 4) Método obtenerAcronimo del proyecto proyecto_acronimos.

Parte 1)

La variable *divisor* debe de ser cualquier número diferente de 0 o sino se mostrará un error.

Parte 2)

Para el correcto funcionamiento del método, la variable *n* debe de ser mayor que 0 o sino se mostrará un error.

Parte 3)

- Si la variable *a* es menor que *z*, la condición con las variables *v* y *m*: $v[m] == c$ se cumple, el programa devolverá la variable resultado con valor true.
- Si la variable *a* es menor que *z* pero la condición anterior no se cumple dependiendo de si se cumple esta otra condición ($v[m] < c$) se modificará el valor de la variable *a* o en caso negativo se modificará la variable *z* devolviendo al final la variable resultado con valor true.

Parte 4)

- Si la variable *n*, tiene un valor inferior o igual a 0 se devolverá el valor de la variable resultado que en ese momento sería un espacio vacío (" ") al no entrar al bucle.
- Si la variable *n* es mayor que 0, se entrará en el bucle y en el primer ciclo del bucle cuando $i=0$ el programa agrega la variable caracter y un punto (.) a la variable resultado, y en los pasos siguientes realiza lo mismo pero restándole 1 a la variable del bucle *i*.

Práctica 3.11: Elaborar casos de prueba:

Parte 1) Método calcularDivision del proyecto proyecto_division.

Parte 2) Método factorial del proyecto proyecto_factorial.

Parte 3) Método busca del proyecto proyecto_arrays.

Parte 4) Método obtenerAcronimo del proyecto proyecto_acronimos.

Parte 1)

- Al darle a la variable n un valor menor o igual que 0 se muestra una excepción con el siguiente texto: *Error. El número tiene que ser >0 y <=127.*
- Si le damos valor 1 por ejemplo, entrará en el for y como no tiene más divisores que sí mismo el resultado será solo el establecido en la declaración de la variable resultado que sería el 1 mismo.
- Si el valor que proporcionamos es mayor que 1 ya empezará a mostrar todos los números divisibles del proporcionado por orden en la String, como por ejemplo con el 4 mostraría: 1 2 4.

Parte 2)

- Si le damos un valor menor o igual que 0 el programa muestra un error con el siguiente texto: *Error. El número tiene que ser >=0.*
- Cuando le demos un número superior como por ejemplo 3, entrará en el for e irá multiplicando el número del bucle actual (variable i, empezando por 2) por todos los números siguientes hasta 3 efectuándose en este caso solo dos multiplicaciones (3*2 y 6*3) devolviéndonos el resultado al final.

Parte 3)

- Si introducimos como por ejemplo la palabra coche, la variable resultado seguirá siendo de valor false, al no cumplir la condición en ningún momento del bucle (`/v[m] == c`).
- Si introducimos la palabra Avellana, nos devolverá el valor true en el primer ciclo del bucle al cumplirse la condición anterior.

Parte 4)

- Si introducimos la cadena Laughing Out Loud, el programa se encargará de entrar en el bucle for y añadir "." entre las letras y al final devolviendo como resultado L.O.L.

Práctica 3.12: Generar y ejecutar pruebas en JUnit y documentar incidencias:

- Proyecto_division:

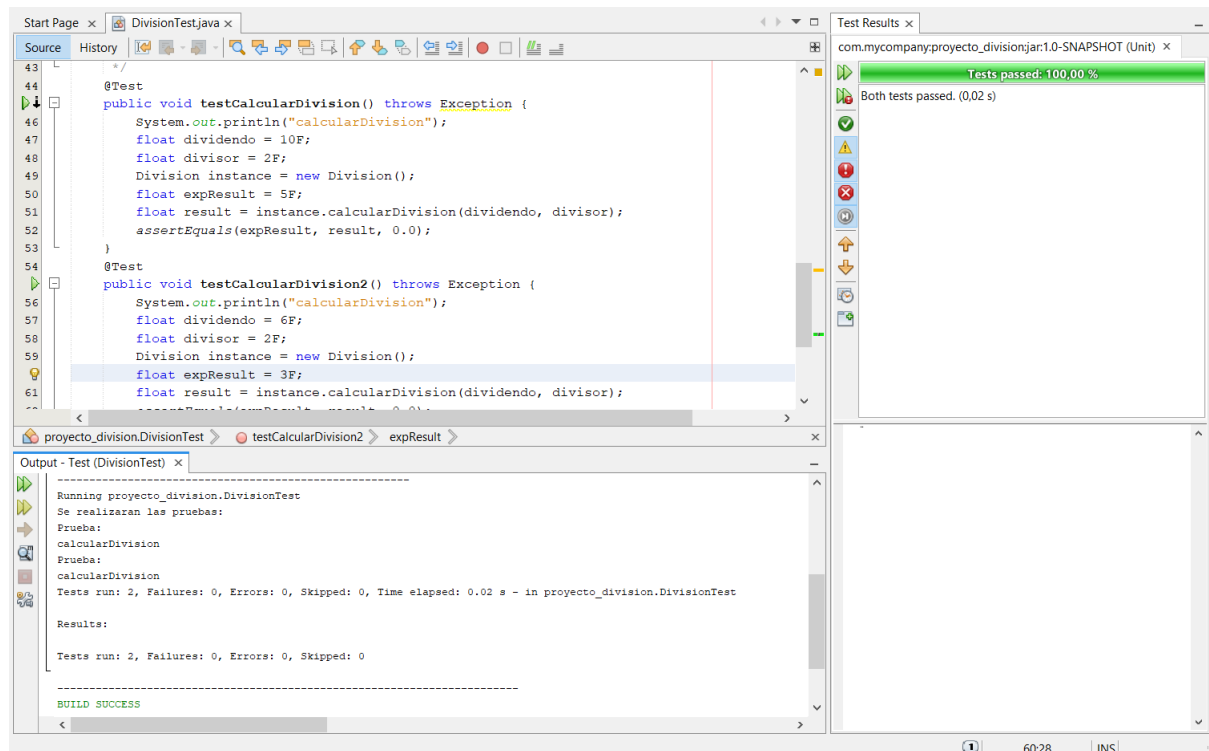
Lo primero que debemos hacer será ir a File, seleccionar New File con la categoría Unit Tests, con el tipo de archivo Test for existing class seleccionando la clase del proyecto actual proyecto_division.

En caso de ser un proyecto maven debemos añadir esto al pom.xml:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
    </plugin>
  </plugins>
</build>
```

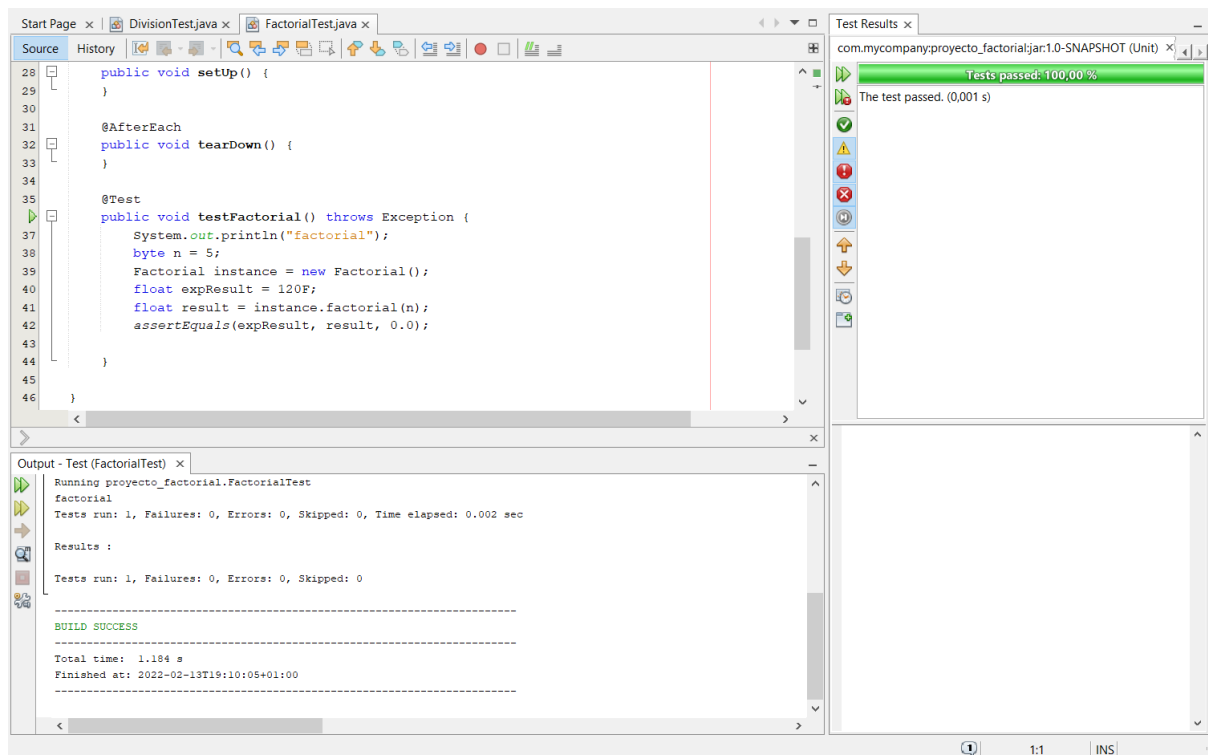
Para ejecutar una prueba debemos de eliminar las dos líneas generadas en el Test for existing class y el código repetido para los @ que se nos ha creado y añadir valores a las variables así como el valor esperado para comprobar si el archivo funciona correctamente.

Podemos añadir en el @BeforeAll una impresión de texto que aparezca en el Output antes de realizar las pruebas como por ejemplo (Se realizarán las pruebas:) y en el @BeforeEach otra impresión de texto que aparezca antes de cada prueba como por ejemplo (Prueba x).



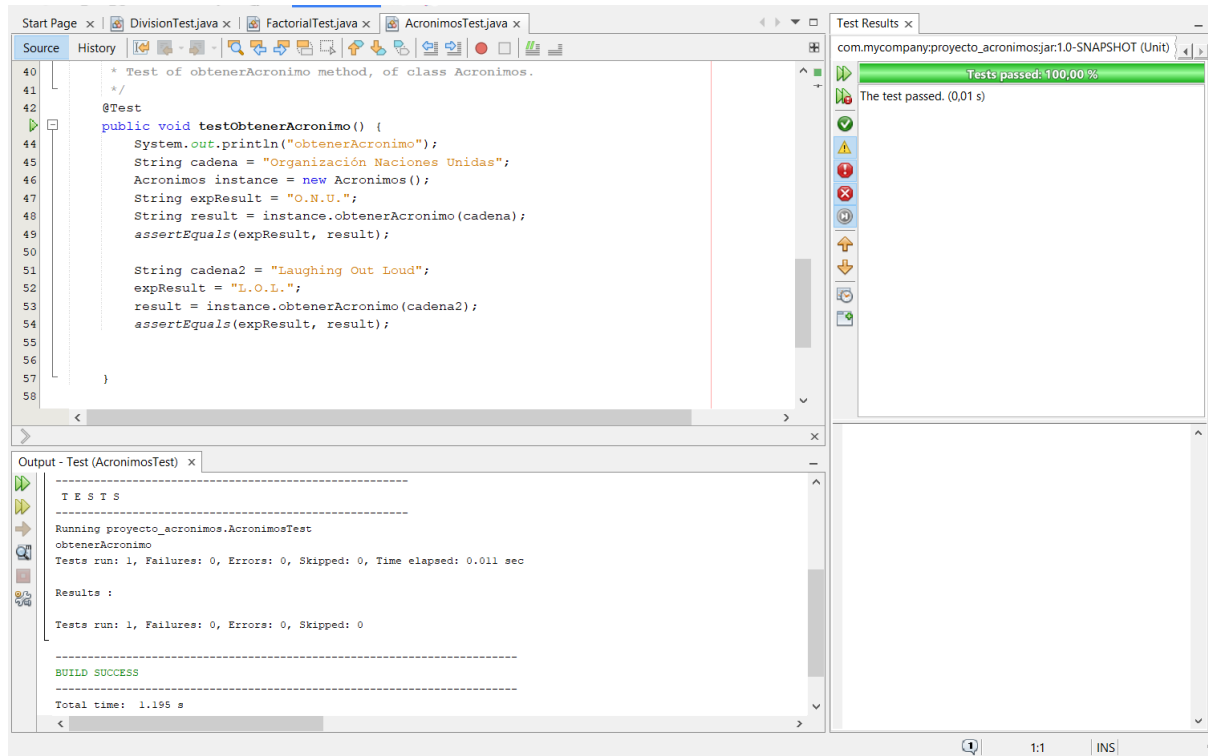
- Proyecto_factorial:

Realizamos los mismos pasos anteriores de la misma forma y podremos realizar los tests como por ejemplo 5 para n y el resultado que debería ser 120. Hacemos Test File y podremos comprobar que funciona correctamente.



- Proyecto_Acrónimos:

Realizamos como los pasos anteriores la creación del archivo Test, y realizamos alguna prueba como por ejemplo ingresar la cadena Organización Naciones Unidas, y de resultado esperado O.N.U., podremos comprobar que el programa funciona correctamente.



Práctica 3.13: Realiza el test de una clase que solo tenga el método siguiente probando la cadena “abcabca” y la letra ‘a’. Haz el análisis de cobertura con JaCoCo y añade los test precisos para obtener una cobertura completa del método:

```
static int contarLetras (String cadena, char letra) throws Exception {
    int contPos=0, conVeces= 0, longCadena = 0;
    longCadena = cadena.length();
    if (longCadena > 0) {
        do {
            if (cadena.charAt(contPos)== letra) conVeces++;
            contPos++;
        } while (contPos < longCadena);
        return conVeces;
    }
    else throw new Exception("Parámetros incorrectos");
}
```


Para hacer el análisis con JaCoCo primero debemos de hacerle Build a nuestro proyecto, y una vez hecho debemos ir al pom.xml y añadir los siguientes plugins:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.8.5</version>
      <executions>
        <execution>
          <id>pre-unit-test</id>
          <goals>
            <goal>prepare-agent</goal>
          </goals>
          <configuration>
            <destFile>${project.build.directory}/coverage-reports/jacoco.exec</destFile>
            <propertyName>surefireArgLine</propertyName>
          </configuration>
        </execution>
        <execution>
          <id>post-unit-test</id>
          <phase>test</phase>
          <goals>
            <goal>report</goal>
          </goals>
          <configuration>
            <dataFile>${project.build.directory}/coverage-reports/jacoco.exec</dataFile>
            <outputDirectory>${project.reporting.outputDirectory}/jacoco</outputDirectory>
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
      <configuration>
        <argLine>${surefireArgLine}</argLine>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Una vez realizados todos los pasos anteriores añadimos los siguientes tests para hacer la cobertura completa con el ejemplo indicado:

@Test

```
public void test_ejecucion1 () {
    String cadena "abcbca";
    char letra = 'a';
    int resultado = Pruebas.contarletra(cadena, letra)
    int resultEsperado = 3;
    AssertEquals (resultEsperado, resultado)
}
```

@Test

```
public void test_ejecucion2 () {
```

```
String cadena = "ajyagdasjfgasjfsafagaagaga"
char letra = 'a';
int resultado = Pruebas.contarletra(cadena, letra)
int resultEsperado =-1;
AssertEquals(resultadoEsperado, resultado)

}
@Test
public void tes_excepcion () {
    String cadena "";
    char letra 'a';
    AssertThrows(Pruebas.contarletra(cadena, letra))
}
```