

JaCoCo (Java Code Coverage)

JaCoCo – Cobertura de Código Java - é unha biblioteca de código aberto para analizar a **cobertura** de código das **probos unitarias** en contornos Java. Tras usala para executar probas unitarias, indica que partes do código se proban mediante probas unitarias e as que non se están a probar. Podemos obter un **informe** en formato **HTML** onde se visualizan os resultados obtidos tras a compilación.

Para utilizar JaCoCo, imos probar o método estático **ePar(int numero)** da seguinte clase:

```
import java.util.*;
public class NumeroPar {
    public static boolean ePar(int numero){
        boolean resposta=false;
        if (numero%2==0){
            resposta=true;
        }else{
            resposta=false;
        }
        return resposta;
    }
    public static void main(String args[]) {
        Scanner in=new Scanner(System.in);
        System.out.println("Introduce un número: ");
        int num=in.nextInt();
        System.out.println(ePar(num));
    }
}
```

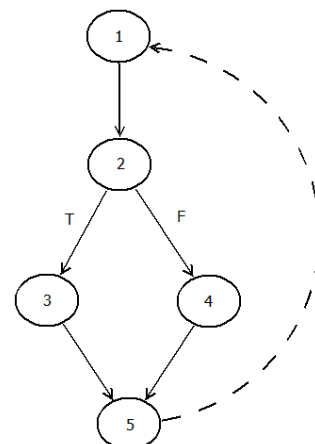
O método **ePar(int numero)** serve para saber se un **número** **entero** é **par** ou non. Se o número é par devolve o valor booleano true e se é falso devolve o valor false.

Para probar a clase **NumeroPar** imos crear un proxecto **Maven** en **NetBeans** de nome **ProxectoNumeroPar**.

Para facer as probas de **cobertura de camiños** imos realizar o **grafo de fluxo** do método **ePar(int numero)**.

```
public static boolean ePar(int numero){
    boolean resposta=false; 1
    if (numero%2==0){ 2
        resposta=true; 3
    }else{
        resposta=false;4
    }
    return resposta; 5
}
```

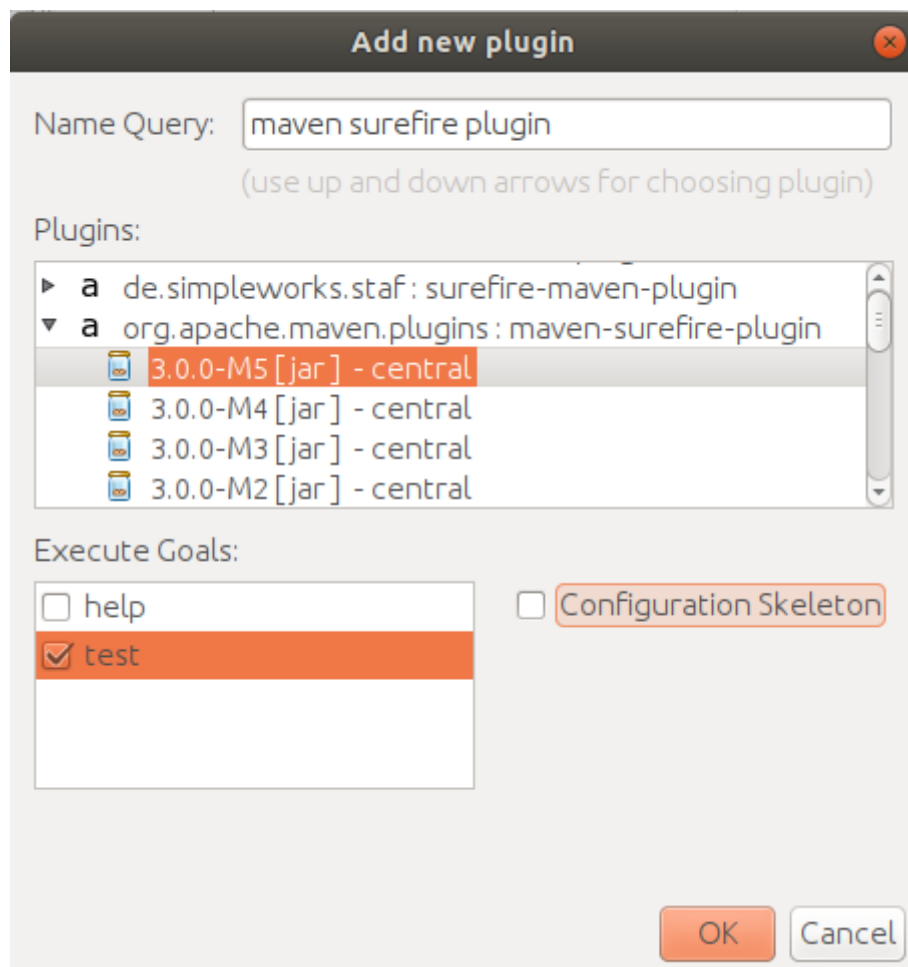
Casos de proba	
Camiños:	numero
1-2-3-5	8
1-2-4-5	7



Tras facer o grafo de fluxo, observamos que a **complexidade ciclomática** é 2 ($V(G)=2$, dúas rexións pechadas no grafo) polo que temos que probar dous camiños que son **1-2-3-5** e **1-2-4-5**. Para probar o primeiro camiño eliximos como **caso de proba** aquel no que **numero** vale **8** e para o segundo aquel no que **numero** vale **7**.

Unha vez determinados os casos de proba, procedemos a codificalos con **JUnit 5**.

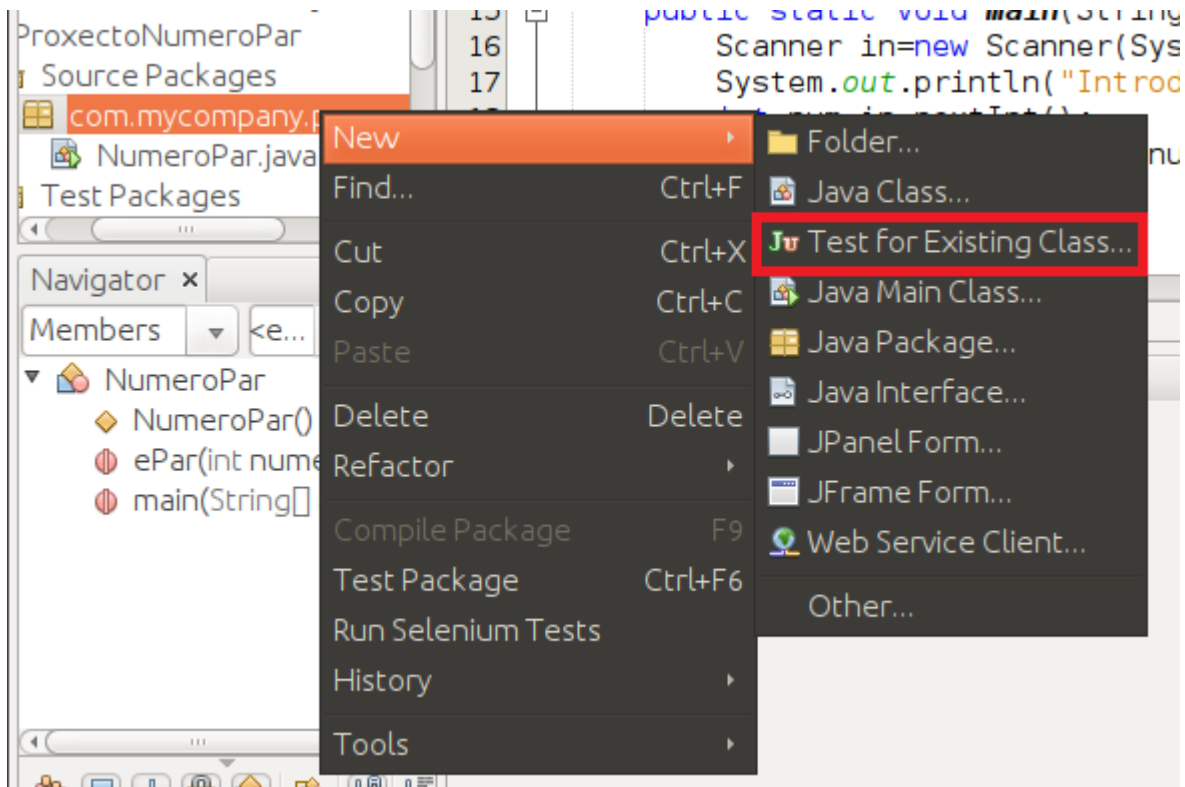
Para utilizar **JUnit 5** en **NetBeans**, necesitamos engadir o **Maven Surefire Plugin** no **pom.xml**. Podemos facelo rapidamente situándonos no código do **pom.xml**, premendo no botón dereito do rato para que apareza o menú contextual **Insert Code...** e tras elixir **Plugin...** indicamos o plugin que buscamos:



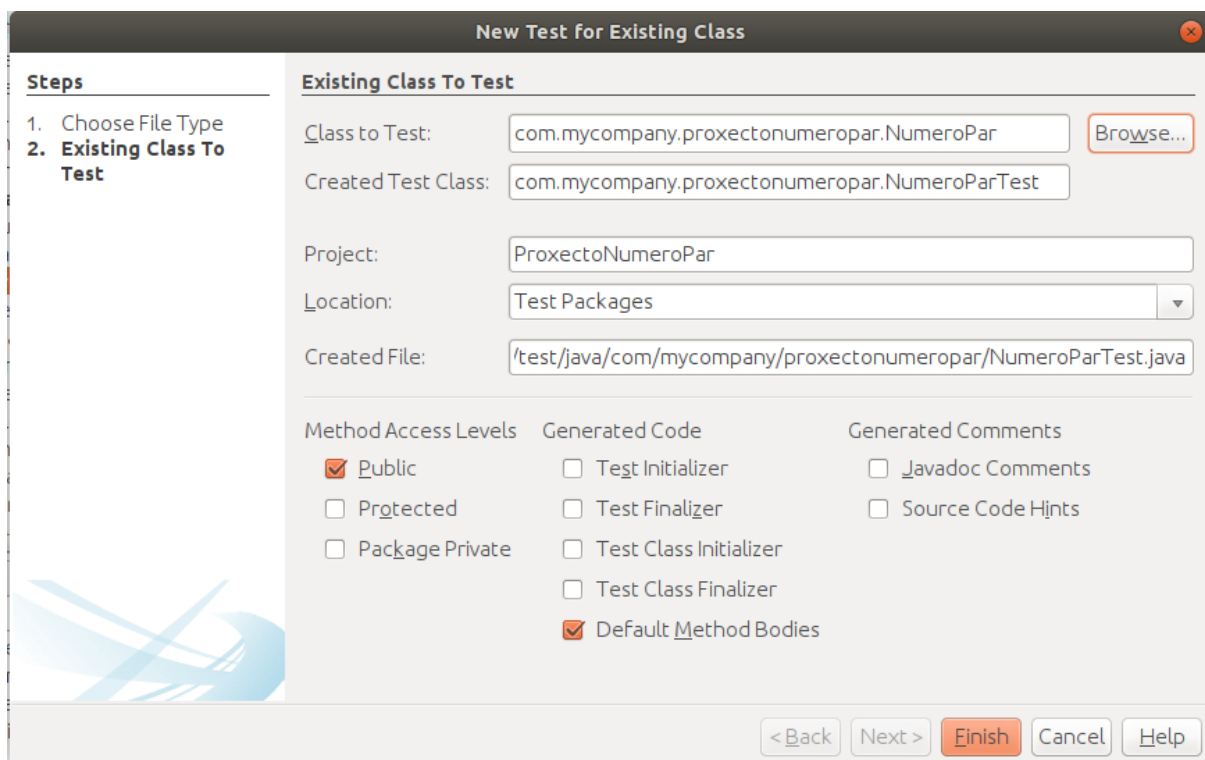
Tamén temos que indicar o **Goal** de execución (test).

Despois de modificar o **pom.xml** procedemos a crear a **clase de probas**.

Para facelo, podemos situarnos no **paquete** onde temos a clase **NumeroPar** e prememos no botón dereito e eliximos **New > Test for Existing Class...**



Indicamos que queremos probar a clase **NumeroPar** e, tras seleccionar o código que queremos que se xere, prememos no botón de **finalizar** (Finish).



NetBeans crea a clase **NumeroParTest** co seguinte código:

```
package com.mycompany.proxectonumeropar;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class NumeroParTest {

    public NumeroParTest() {
    }
    @Test
    public void testEPar() {
        System.out.println("ePar");
        int numero = 0;
        boolean expResult = false;
        boolean result = NumeroPar.ePar(numero);
        assertEquals(expResult, result);
        fail("The test case is a prototype.");
    }

    @Test
    public void testMain() {
        System.out.println("main");
        String[] args = null;
        NumeroPar.main(args);
        fail("The test case is a prototype.");
    }
}
```

O código xerado é un esqueleto que temos que modificar para facer as probas que nos interesan.

Neste caso borramos a proba do main testMain() xa que só imos probar o método ePar(int numero).

Codificamos os dous casos de proba que obtivemos tras o cálculo da complexidade ciclomática.

A clase de probas sería a seguinte:

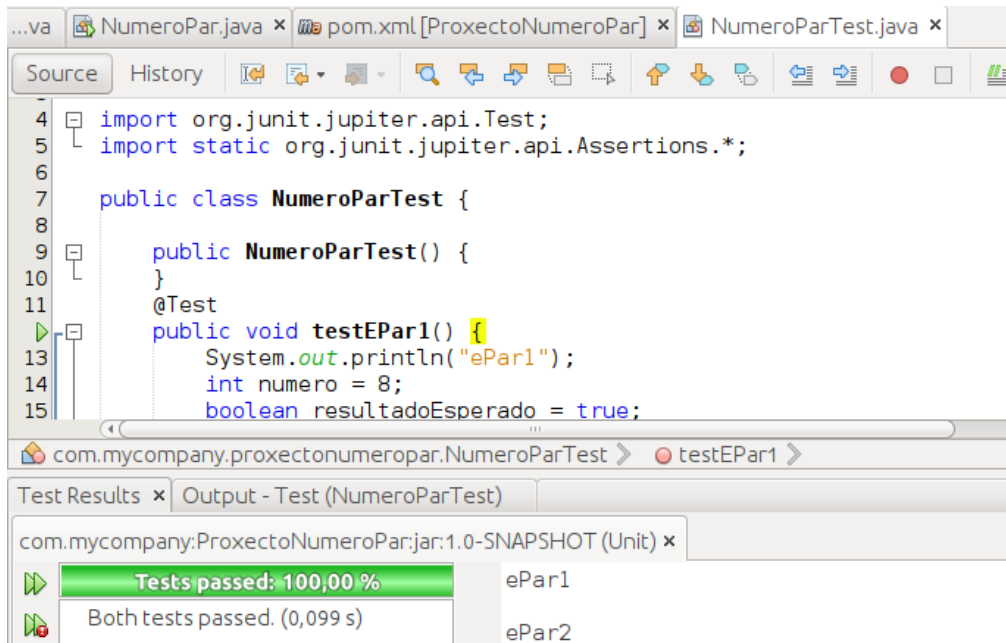
```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class NumeroParTest {
    public NumeroParTest() {
    }
    @Test
    public void testEPar1() {
        System.out.println("ePar1");
        int numero = 8;
        boolean resultadoEsperado = true;
        boolean resultado = NumeroPar.ePar(numero);
        assertEquals(resultadoEsperado, resultado);
    }

    @Test
    public void testEPar2() {
        System.out.println("ePar2");
        int numero = 7;
        boolean resultadoEsperado = false;
        boolean resultado = NumeroPar.ePar(numero);
        assertEquals(resultadoEsperado, resultado);
    }
}
```

Tras **codificar** as probas que queremos facer, **executámolas**.

Prememos sobre o código da proba no botón dereito do rato e eliximos no menú contextual **Test File**.

O resultado da execución pode verse no seguinte pantallazo:

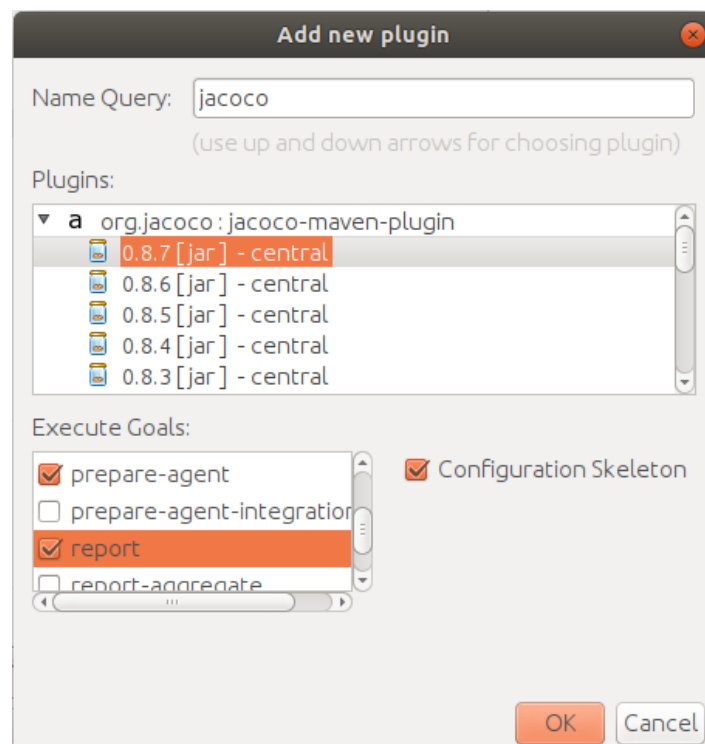


Vemos que a proba **pasou** os **dous test**.

Agora imos proceder a realizar a avaliación da cobertura de código con **JaCoCo**.

Para utilizar JaCoCo imos engadilo no **pom.xml**. Tendo o foco no código do **pom.xml** prememos no botón dereito e eliximos no menú contextual **Insert Code...** para posteriormente elixir **Plugin....**

Buscamos **jacoco** no buscador de plugins:

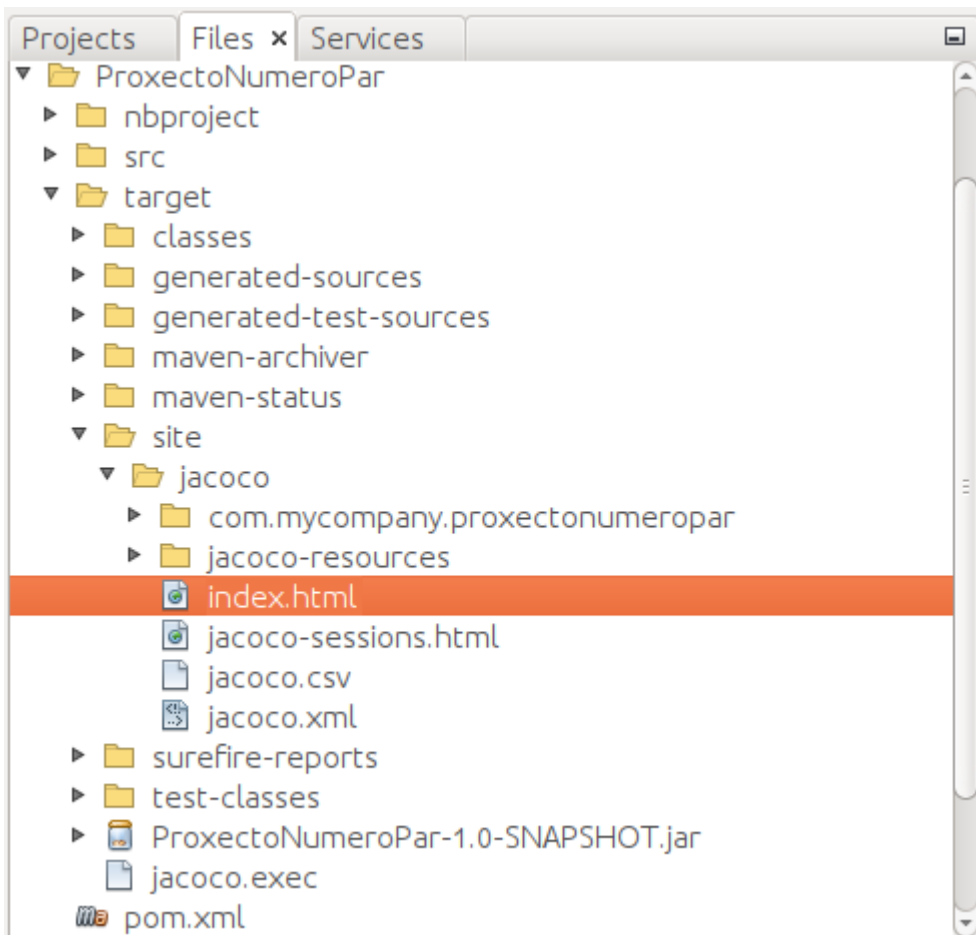


Eliximos os goals **prepare-agent** e **report** e prememos no botón OK.

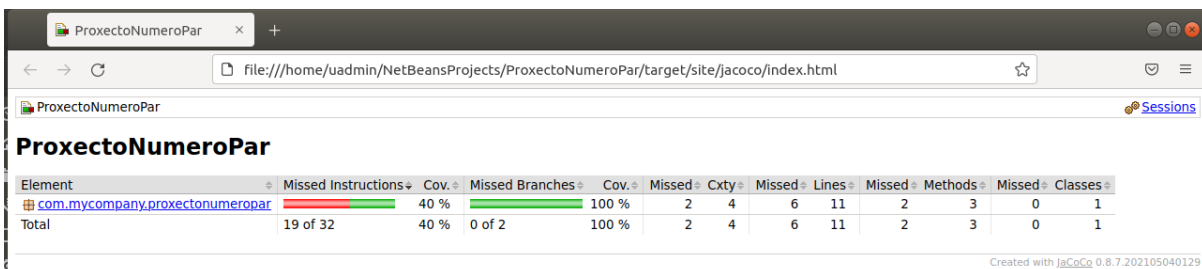
Tras engadir o plugin de JaCoCo, o que imos facer vai ser un **Clean and Build** do proxecto para compilalo.

Prememos na ventá **Projects** no **ProxectoNumeroPar** e tras pulsar no botón dereito do rato, eliximos no menú contextual **Clean and Build**.

Na ventá **Files** podemos ver que no cartafol **target** se creou un cartafol **site**. No cartafol site é onde ten que estar o **informe HTML** sobre a cobertura de código.



O primeiro **informe** que nos aparece é o que informa sobre o paquete **com.mycompany.proxectonumeropar**.



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
com.mycompany.proxectonumeropar	19 of 32	40 %	0 of 2	100 %	2 4	6 11	2 3	0 1
Total	19 of 32	40 %	0 of 2	100 %	2 4	6 11	2 3	0 1

O primeiro contador de cobertura ten que ver coa **cobertura de sentenzas**, é dicir, a porcentaxe de sentenzas do código que se teñen executado con respecto ao total de instrucións que ten o código. Para o proxecto a cobertura de sentenzas é do 40%.

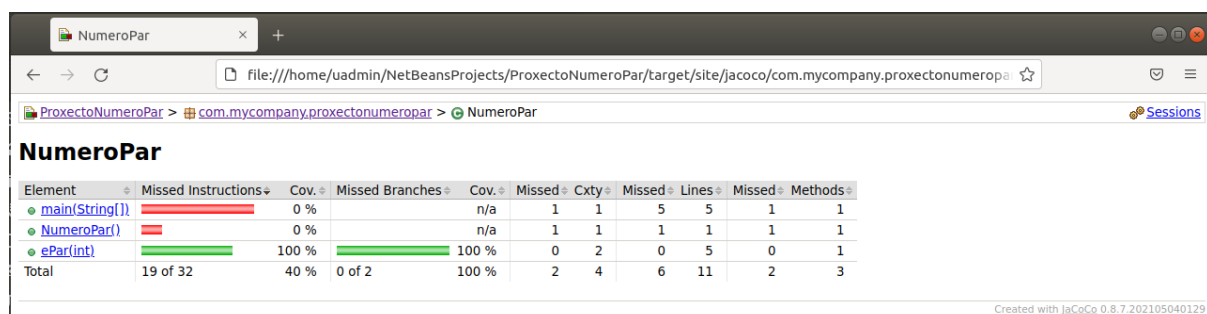
O segundo contador úsase para indicar a **cobertura de decisións**, é dicir, o número de ramas do código con sentenzas if ou switch que se executaron con respecto ao total. Esta cobertura é do 100% para o proxecto.

O último contador é o que calcula a **complexidade ciclomática (Cxyt)**. Para o proxecto a **complexidade ciclomática** é de 4.

Na seguinte imaxe vemos que o método main ten unha cobertura de sentenzas do 0%. O método ePar ten unha cobertura de sentenzas do 100%.

O método main non ten sentenzas if nen switch e por iso a cobertura indica n/a (non aplica).

A **complexidade ciclomática** do método **ePar(int)** vemos que é 2, tal e como calculamos.



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxyt	Cxyt	Missed Lines	Lines	Missed Methods	Methods
main(String[])	19 of 32	0 %	n/a	n/a	1	1	5	5	1	1
NumeroPar()	0 %	0 %	n/a	n/a	1	1	1	1	1	1
ePar(int)	100 %	100 %	100 %	100 %	0	2	0	5	0	1
Total	19 of 32	40 %	0 of 2	100 %	2	4	6	11	2	3

Se nos fixamos no **informe** inferior para o método **ePar(int numero)**, vemos que todo o código deste método ten unha cor de fondo de cor verde e incluso un rombo verde na liña 8.

A cor de **fondo verde** axúdanos a saber que todo o código con esa cor está cuberto coas probas.

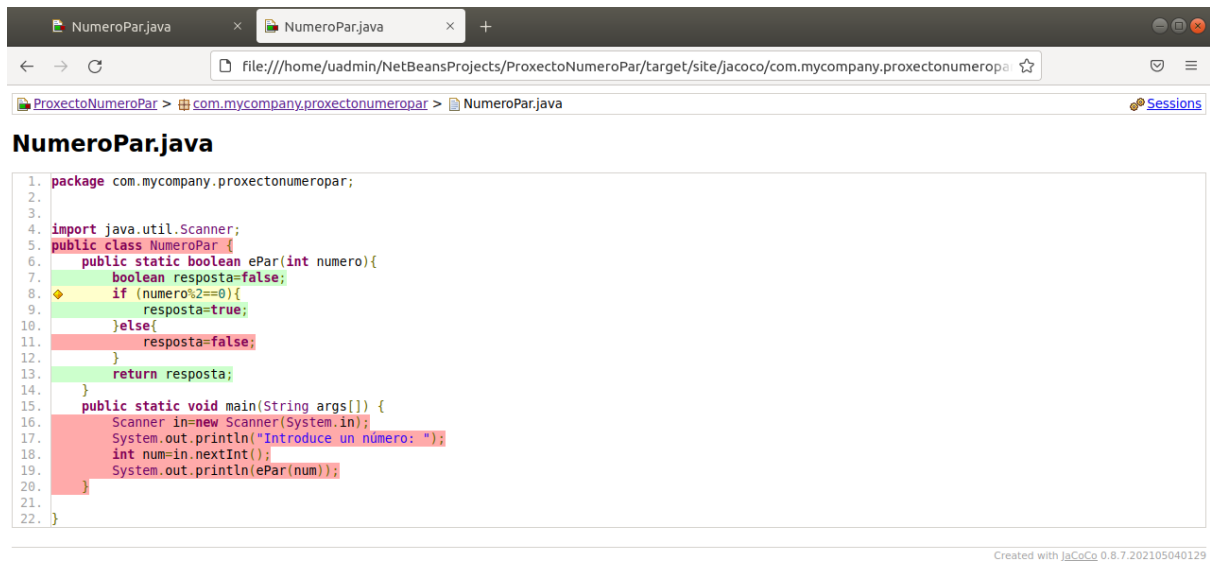
No caso de que o **fondo** fose **amarelo**, saberíamos que a cobertura é só parcial e, se o **fondo** fose de cor **vermella**, entón, ningunha instrución desa liña ten sido executada polas probas.



```
1. package com.mycompany.proxectonumeropa;
2.
3.
4. import java.util.Scanner;
5. public class NumeroPar {
6.     public static boolean ePar(int numero){
7.         boolean resposta=false;
8.         if (numero%2==0){
9.             resposta=true;
10.        }else{
11.            resposta=false;
12.        }
13.        return resposta;
14.    }
15.    public static void main(String args[]) {
16.        Scanner in=new Scanner(System.in);
17.        System.out.println("Introduce un número: ");
18.        int num=in.nextInt();
19.        System.out.println(ePar(num));
20.    }
21.
22. }
```

O **rombo** utilízase para a **cobertura de decisións**. Un rombo verde indica que todas as decisións relacionadas con esa liña se teñen executado.

Se deshabilitamos coa anotación `@Disabled` o caso de proba 2 e volvemos compilar podemos ver que en vez dun rombo verde aparece un rombo amarelo e observamos que o código `resposta=false;` non se executou nas probas.



The screenshot shows a web browser window displaying the execution of `NumeroPar.java`. The code is as follows:

```
1. package com.mycompany.proxectonumeropar;
2.
3.
4. import java.util.Scanner;
5. public class NumeroPar {
6.     public static boolean ePar(int numero){
7.         boolean resposta=false;
8.         if (numero%2==0){
9.             resposta=true;
10.        }else{
11.            resposta=false;
12.        }
13.        return resposta;
14.    }
15.    public static void main(String args[]) {
16.        Scanner in=new Scanner(System.in);
17.        System.out.println("Introduce un número: ");
18.        int num=in.nextInt();
19.        System.out.println(ePar(num));
20.    }
21. }
22. }
```

The execution results show a yellow diamond icon next to the `if (numero%2==0){` line, indicating that the test case was skipped. The output shows the program running successfully.

Se deixásemos sen executar ningún dos casos de proba, o rombo sería vermello.



The screenshot shows the same code as the previous image, but the execution results show a red diamond icon next to the `if (numero%2==0){` line, indicating that the test case failed. The output shows the program running successfully.

JaCoCo, por tanto, xera informes de cobertura de código para proxectos Java.

Non obstante, temos que ter en conta que unha cobertura de código do 100% non reflexa necesariamente unha proba eficaz.