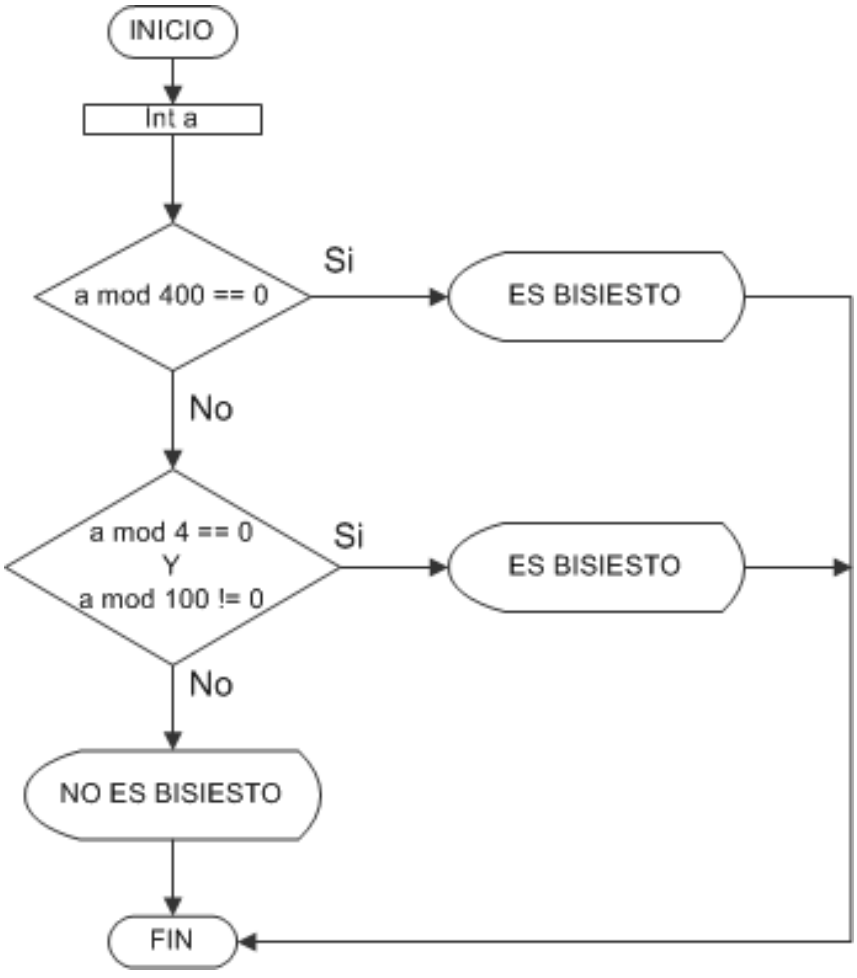
Complejidad ciclomática ¿Nuestro código es fácil de mantener y probar? (I)

DEFINICIÓN

La complejidad ciclomática es una métrica de calidad software basada en el cálculo del número de caminos independientes que tiene nuestro código.

Estos caminos se identifican a partir de las estructuras de control (condicionales, bucles, …) incluidas en la mayoría de los lenguajes de programación, y más concretamente, a partir del diagrama de flujo de cada uno de nuestros métodos.



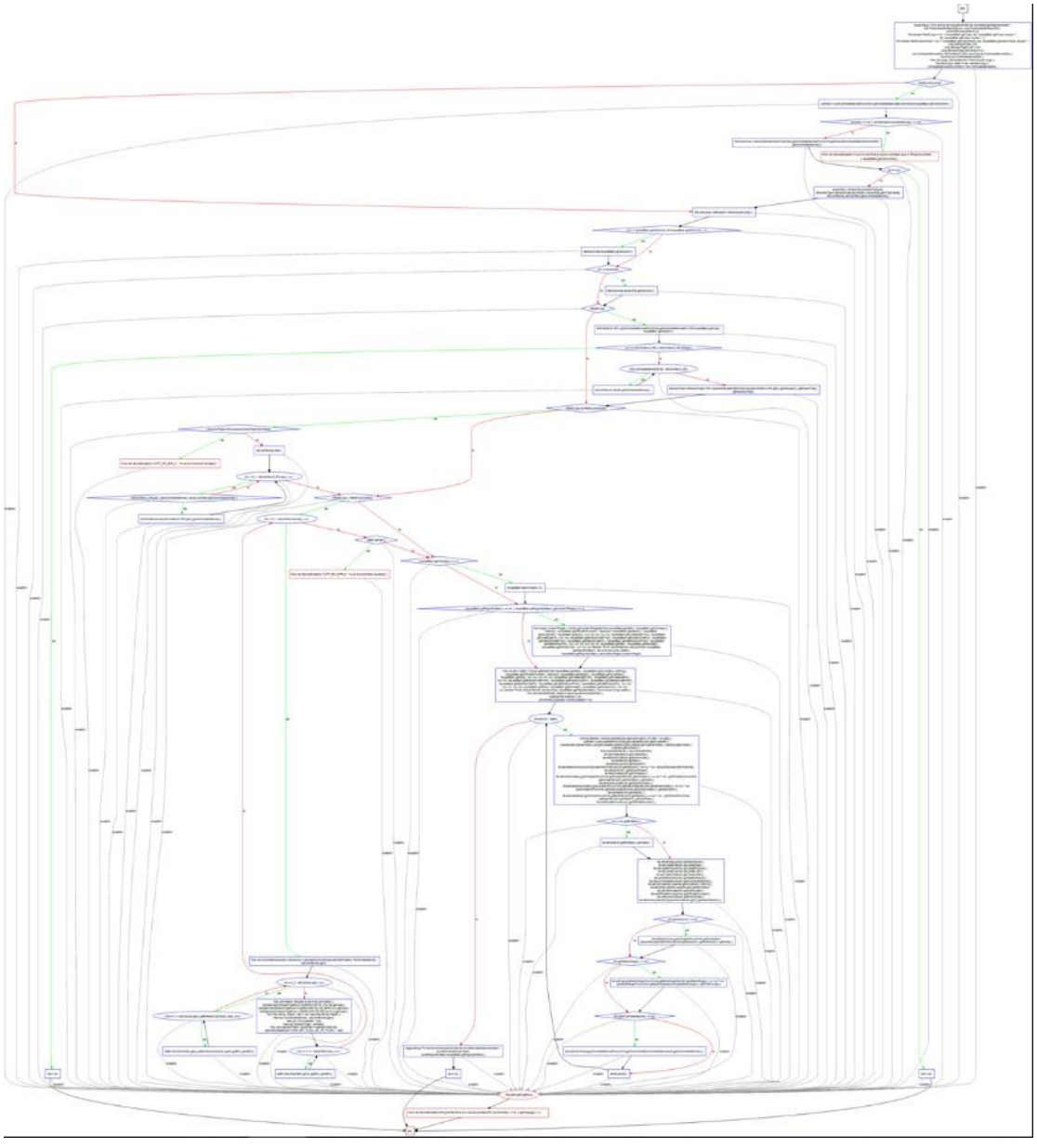
Ejemplo de diagrama de flujo. Detección de año bisiesto

¿Para qué sirve la complejidad ciclomática?

Esta métrica fue propuesta por Thomas McCabe en 1976 y permite tomar la temperatura de nuestro código respecto de su nivel de mantenibilidad, la probabilidad de incluir fallos o el esfuerzo necesario para poder probar todos sus caminos.

La idea es sencilla, cuanto más compleja sea la lógica de un código, más difícil será de entender, mantener y probar.

Esto mismo de manera gráfica, podemos verlo en este diagrama de flujo que muestra “la realidad” de un método con complejidad ciclomática alta. ¿Parece fácil de mantener? Claramente… ¡no!



¿Y cómo se calcula?

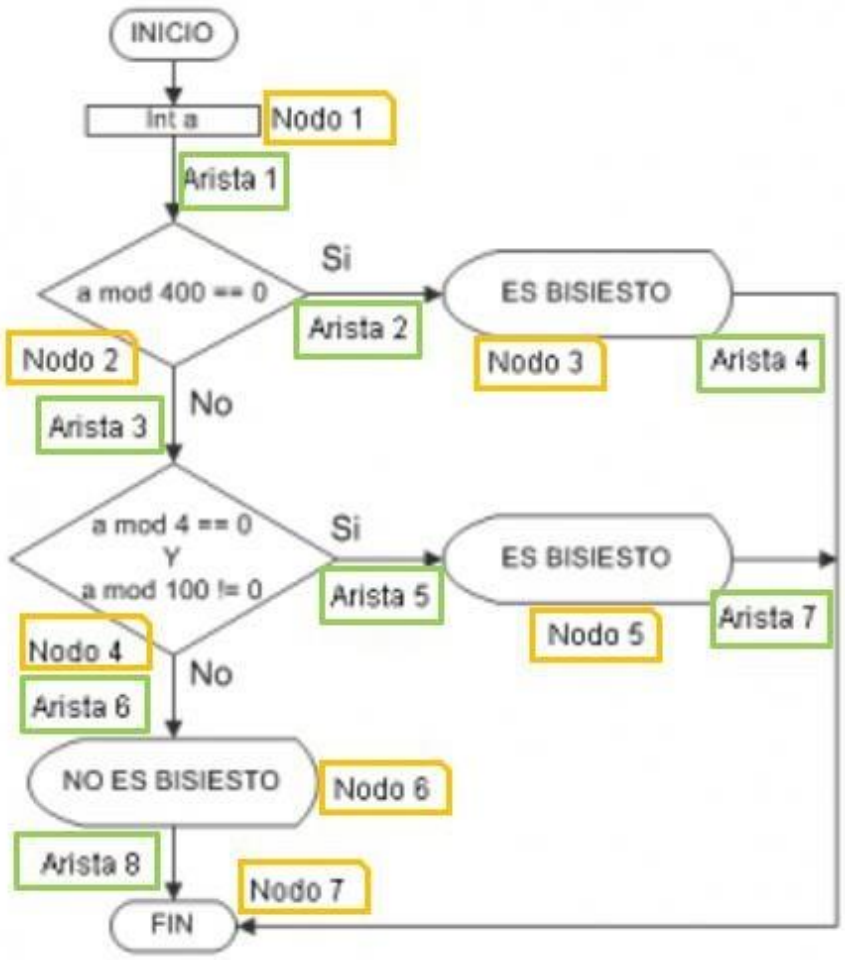
Si nos atenemos a la definición de complejidad ciclomática, como detección y recuento de caminos independientes, la idea más “purista” es acudir al diagrama de flujo, tratarlo como un grafo, con sus nodos y aristas, y buscar cuántos caminos diferentes hay desde el nodo inicial al final.

En el ejemplo anterior sobre calcular si un año es bisiesto o no, tendremos 3 posibles caminos *(SI, NO-SI, NO-NO)*.

Otra manera de calcular la complejidad ciclomática sería con la fórmula (simplificada para un único punto de entrada y salida) …

v(G) = e – n + 2, *donde e representa el número de aristas y n el número de nodos.*

Si para nuestro ejemplo numeramos cada instrucción (los nodos) y las transiciones entre las mismas (las aristas) nos queda:



Grafo numerando nodos y aristas
***v(G) = 8 -7 + 2 = 3**, siendo e = 8; n = 7*

Por último, otra alternativa para calcular la complejidad ciclomática utilizando el grafo es:

v(G) = número de regiones cerradas en el grafo + 1

Que en nuestro caso vuelvo a coincidir con 3 (2 + 1), al haber 2 regiones cerradas.

Valores de referencia

Como se puede intuir, una complejidad ciclomática de 3 “habla” de un método sencillo, con poca lógica.

Thomas McCabe, establece en sus trabajos los siguientes valores de referencia:

- <= 10, métodos sencillos, sin mucho riesgo.
- > 10, <= 20, métodos medianamente complejos, con riesgo moderado.
- > 20, <= 50, métodos complejos, con alto riesgo.
- > 50, métodos inestables, de altísimo riesgo.

Se puede marcar un valor de referencia de 20 como máxima complejidad ciclomática aceptable para un método, a partir de la cual hay que estudiar opciones de optimización o refactorización.

Hay que tener en cuenta que existen métodos de por sí complejos, que no admiten simplificación, y que refactorizarlos al extremo pueden introducir una complejidad innecesaria en un sistema, perdiéndose igualmente la mantenibilidad del código.

Por otra parte, no hay que olvidarse de identificar un valor mínimo para la complejidad ciclomática, a partir del cual hay que poner la lupa en nuestros métodos, para ver si “no aportan valor añadido” a nuestro sistema.

¿Cómo se calcula la complejidad ciclomática en la práctica?

Herramientas como *SonarQube* no se complican tanto la vida, y hacen una simplificación de esta métrica, con un simple recuento de estructuras de control en nuestro código.

Complejidad ciclomática ¿Nuestro código es fácil de mantener y probar? (II)

Hemos definido el concepto de complejidad ciclomática, una métrica de calidad software que nos permite saber "cómo de fácil es de probar y mantener nuestro software".

Además, vimos que su cálculo pasaba por identificar los caminos independientes en la estructura de flujo de nuestros métodos, y que para esto mismo podíamos utilizar una representación de grafo, donde los nodos son las instrucciones, incluyendo if, while, for… y las aristas van formando los posibles caminos de ejecución.

Una posible fórmula para el cálculo de la complejidad ciclomática (simplificada para el caso de un único punto de entrada y salida) es:

***v(G) = e – n + 2**, donde e representa el número de aristas y n el número de nodos.*

Por tanto, si pensamos en un método muy sencillo:

```
void numPorDos(int x) {
int r = x * 2; //Nodo 1
return r;  // Nodo 2
}
```

sólo tendremos dos nodos conectados con una arista, y por tanto una complejidad de **v(G) = 1 -2 + 2 = 1**

Si añadimos un nodo más…

```
void numPorDosPorDos(int x) {
int aux = x * 2; //Nodo 1
(arista)
int r = aux * 2; // Nodo 2
(arista)
return r;  // Nodo 3
}
```

…tendremos tres nodos conectados con dos aristas… y una complejidad ciclomática de **v(G) = 2 -3 +2 = 1**

¿Qué podemos hacer para hacer crecer la complejidad ciclomática? Probemos añadiendo una condición if…

```
void numExtra(int x) {
int aux = x * 2; //Nodo 1
(arista)
int r = aux * 2; //Nodo 2
(arista)
if(r > 25)  { // Nodo 3 + (arista a Nodo 5)
(arista)
r = r -10; //Nodo 4
(arista)
}
return r;  //Nodo 5
}
```

La complejidad ciclomática quedaría… **v(G) = 5 – 5 + 2 = 2**. Podemos apreciar cómo el if ha sumado en 1 la complejidad ciclomática.

¿Y si planteamos una estructura while? La cosa quedaría…

```
void numExtra(int x) {
int aux = x * 2; //Nodo 1
(arista)
int r = aux * 2; //Nodo 2
(arista)
while(r < 25)  { // Nodo 3 + (arista a Nodo 5)
(arista)
r = r + 1; //Nodo 4
(arista a Nodo 5) (arista a Nodo 3)
```

```
}
return r;   //Nodo 5
}
```

La complejidad ciclomática entonces será: **V(G) = 6 – 5 + 2 = 3 …** El while ha aumentado en 2 la complejidad ciclomática.

Entonces… si el aumento de la complejidad ciclomática tiene una relación lineal con la existencia de estos puntos de decisión ¿podemos simplificar el cálculo de la complejidad ciclomática simplemente sumando estructuras de control (if, while, …)?

¿Cómo se calcula la complejidad ciclomática en la práctica?

La herramienta *SonarQube* hace la siguiente interpretación de la complejidad ciclomática (en este caso para Java):

Todo se reduce simplemente a contar las declaraciones "if", "for", "while", etc. en un método. Cuando el flujo de control de un método se divide, el contador ciclomático se incrementa en uno.

Cada método tiene un valor mínimo de 1 por defecto. Para cada una de las siguientes palabras clave / declaraciones de Java, este valor se incrementa en uno:

if | **for** | **while** | **case** | **catch** | **throw** | **return** (que no sea la última declaración de un método | **&&** | **||** | **?**

Por tanto, SonarQube considera que un método tiene por defecto una complejidad ciclomática de 1 (excepto los getters y setters que no se consideran para este cálculo), y suma 1 cada vez que aparece una instrucción del listado anterior, no haciendo distinción sobre las mismas.