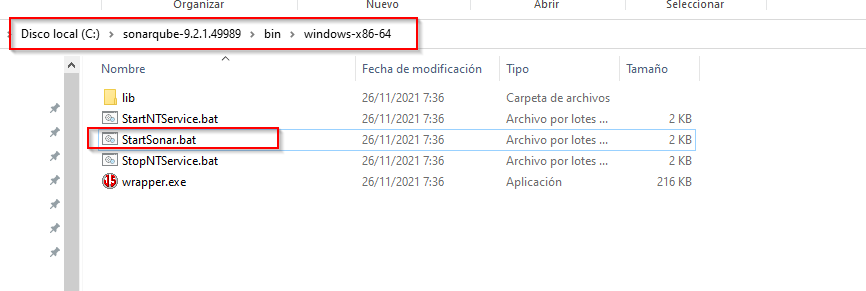
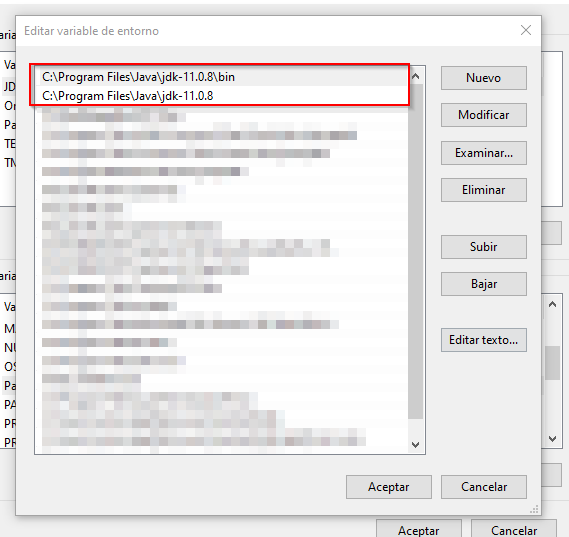
Una vez descargado el paquete de SonarQube lo descomprimimos. Para ejecutarlo nos dirigimos al directorio “..\sonarqube-9.2.1.49989\bin\windows-x86-64” y ejecutamos el archivo (si estamos en Windows) “StartSonar.bat”.



La última versión de Sonar (9.2.1.49989 a diciembre de 2021) necesita un jdk 11. Incluso para instalar el plugin SonarLint en eclipse se tiene que configurar el eclipse.ini para que utilice jdk 11. A lo mejor es necesario tocar la variable del sistema (PATH) para que apunte a un jdk 11.





Una vez levantado el Sonar se puede acceder a él a través de localhost:9000.

Una vez accedido las credenciales para acceder son “admin” “admin”. En el primer acceso nos pedirá que cambiemos la clave.

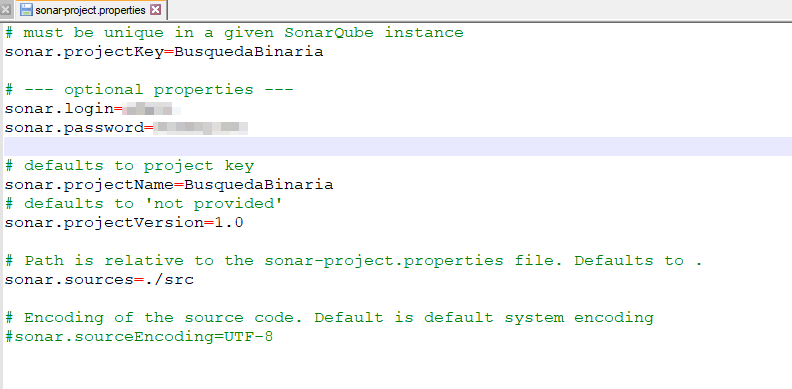
Para empezar a monitorear un programa tendremos que “crear el proyecto” dentro de la plataforma de SonarQube y seguir los pasos del asistente, en el que se nos proporcionará un token y un comando para ejecutar el proyecto en caso de ser un proyecto Maven:

mvn clean verify sonar:sonar -Dsonar.projectKey=prueba:prueba -Dsonar.host.url=http://localhost:9000 -Dsonar.login=0ce471e12a077eaab77a6c072cf8953936f83769

prueba:prueba es el nombre de paquete y proyecto que se pide en el asistente, no tiene por qué coincidir con el nombre del proyecto Java.

Para analizar un proyecto de cualquier otro tipo (no Maven) se tiene que descargar el sonar-scanner e introducir la carpeta bin en la variable Path de Windows.

Una vez configurado el Path, abrimos una terminal y nos situamos en la raíz del proyecto que queremos analizar y ejecutamos el comando sonar-scanner. Previamente tendremos que tener un archivo sonar-project.properties el que definir los parámetros para el escaneo del proyecto.



Las métricas de SonarQube te facilitan la información objetiva de la calidad actual de tu proyecto.

SonarQube divide las métricas en las siguientes categorías:

* Evidencias
* Duplicados
* Complejidad
* Tamaño
* Pruebas
* Fiabilidad
* Seguridad
* Mantenibilidad
* Evidencias / Issues

**Evidencias**

Las evidencias son los fragmentos de código de un proyecto que SonarQube detecta que incumplen con algunas de las reglas establecidas para cada lenguaje en su respectivo perfil de calidad. En esta categoría puedes encontrar:

Nuevas evidencias (new\_violations)

Número de evidencias detectadas desde el principio del periodo de “Nuevo código”, que es un parámetro que se debe configurar en SonarQube.

Nuevas evidencias [severidad] (new\_[severity]\_violations)

Indica el número de evidencias sobre la severidad especificada, detectadas desde el inicio del periodo de “Nuevo código” que puede ser: bloqueante, crítica, mayor, menor o info (bloquer, critical, major, minor o info respectivamente)

Evidencias (violations)

Es el valor total de todas las evidencias posibles.

Evidencias [severidad] ([severity]\_violations)

Muestra el valor total de evidencias por cada severidad: bloqueante, crítica, mayor, menor o info (bloquer, critical, major, minor o info respectivamente)

Falsos positivos (false\_positive\_issues)

En este caso se trata del total de evidencias que se han marcado como falso positivo

Evidencias abiertas (open\_issues)

Indica el número de evidencias que se encuentran con el estado “Abierto”, es el estado inicial de una evidencia.

Evidencias confirmadas (confirmed\_issues)

Refleja el número de evidencias que han sido marcadas como “Confirmadas”, estado creado para indicar que esa evidencia se va a resolver

Evidencias reabiertas (reopened\_issues)

Número total de evidencias en estado “Reabierta”, que son aquellas que se han vuelto a abrir tras haber sido cerradas por algún motivo.

**Duplicados**

La programación permite evitar duplicados gracias a constantes y métodos, lo que ayuda en gran medida a evitar resultados distintos en operaciones iguales, que producirían errores. Por eso es importante mantener un código limpio y evitar los duplicados.

Sobre fragmentos duplicados puedes encontrar las siguientes métricas de SonarQube

Bloques duplicados (duplicated blocks)

Número de bloques de líneas de código que se duplican. Para que un bloque de código se considere duplicado:

* Proyecto Java
  + Hay al menos 10 sentencias sucesivas y duplicadas con cualquier numero de tokens y líneas. Se ignoran las diferencias en indentación y literales de strings.
* Proyecto No Java
  + Hay al menos 100 tokens sucesivos y duplicados
  + Dichos tokens deben encontrarse en al menos:
  + 30 líneas de código para COBOL
  + 20 líneas de código para ABAP
  + 10 líneas de código para el resto de lenguajes

Archivos duplicados (duplicated\_files)

Número de archivos involucrados en la duplicación

Líneas duplicadas (duplicated\_lines)

Número de líneas involcradas en la duplicación

Porcentaje de líneas duplicadas (%) (duplicated\_lines\_density)

Se obtiene de dividir el número de líneas duplicadas entre el total de líneas y multiplicarlo por 100.

**Complejidad**

La complejidad de un proyecto de software dificulta su mantenimiento y escalabilidad, por eso es tan importante que tu código sea lo más sencillo posible.

Complejidad (complexity)

Es la [complejidad ciclomática](http://oscarmoreno.com/la-complejidad-ciclomatica/), calculada en el número posible de rutas que puede seguir el código.

Cuando el flujo de control de una función crea dos posibles caminos, el contador de la complejidad se incrementa en uno.

Cada función tiene un mínimo de complejidad de 1. Éste cálculo puede variar ligeramente entre lenguajes debido a sus funcionalidades.

Lo mejor para calcular la complejidad ciclomática es pasar el código a un grafo y seguir uno de los siguientes algoritmos.

Vamos a tomar como ejemplo el siguiente fragmento de código:

int x,y,r;

if (x<0 || y<0) {

system.out.println(“X o Y son negativos”);

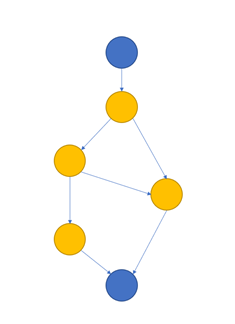
} else {

r=(x+y)/2;

system.out.println(“La media de X e Y es:” + r );

}

Y sobre él obtendrías el siguiente diagrama, en el que los círculos serán los nodos, y las flechas las aristas. Como puedes ver los nodos azules (2), representan la entrada y la salida de la ejecución, los naranjas (4) las condiciones, y las aristas (7) las rutas de ejecución.



Para calcularla, tienes tres opciones:

* Restar las aristas menos los nodos y sumar 2:

V(G) = Aristas – Nodos + 2

V(G) = 7 – 6 + 2 = 3

* Sumar 1 al número de nodos predicados (aquellos de los que salen dos flechas)

V(G) = Nodos predicados + 1

V(G) = 2 + 1 = 3

* Contar el número de regiones (espacios «encerrados entre nodos y aristas», también se tiene en cuenta el espacio «exterior» a todos los nodos y aristas.

V(G) = Regiones

V(G) = 3

Existe un cierto «acuerdo» en cuanto a la simplicidad de un código, en función de su complejidad ciclomática. Por lo que se establece que, según el valor obtenido, podemos determinar cómo es:

|  |  |
| --- | --- |
| Complejidad ciclomática | Tipo de código |
| 1-10 | Simple |
| 11-20 | Algo complejo |
| 21-50 | Complejo |
| 50 | No testeable |

Para poder verla en Sónar, sólo tienes que acceder, dentro del proyecto que hayas analizado, a *Measures > Complexity > Cyclomatic Complexity*

Ahí podrás ver el valor total de todo el proyecto, así como el de cada archivo o directorio.

Complejidad cognitiva (cognitive\_complexity)

Como de difícil es el ciclo que sigue el código. La Complejidad Cognitiva es una medida de cómo es de difícil entender intuitivamente un bloque de código. A diferencia de la Complejidad Ciclomática, que determina qué dificultad tiene probar el código.

Complejidad Algorítmica

No se puede medir en tiempo ya que es dependiente del hardware. La notación BIG O nos da una idea de la eficiencia del algoritmo (en términos de tiempo y memoria). Por ejemplo:

O(1) 🡪 Tiempo constantes, el algoritmo no se verá afectado por el tamaño del input.

O(n) 🡪 Expresión de crecimiento lineal, a medida que crece el input crecerá la complejidad del algoritmo (con la misma proporción de crecimiento).

O(n2) 🡪 Expresión de crecimiento cuadrático, la complejidad crece proporcionalmente al cuadrado del tamaño del input.

O(log n) 🡪 Reducen su complejidad a medida que el input aumenta.

