PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERÍA

CALIDAD DE CÓDIGO
ARQUITECTURA DE SOFTWARE

NOVIEMBRE 2025

Juan Diego Romero

Kamilt Bejarano Diaz

Pablo Enrique Quintero

Arquitectura de Software

Universidad Javeriana, Noviembre 2025

Juan Romero, Kamilt Diaz, Pablo Quintero

Índice

Arquitectura de Software

Universidad Javeriana, Noviembre 2025

Juan Romero, Kamilt Diaz, Pablo Quintero

Definición

La calidad de código es el conjunto de propiedades internas del código fuente que permiten que el software sea mantenible, comprensible, seguro, eficiente y fácil de evolucionar enfocándose específicamente en los atributos técnicos que dependen de cómo los desarrolladores escriben, estructuras y mantienen el código.

Es un concepto que abarca aspectos como la legibilidad del código, simplicidad y baja complejidad, ausencia de duplicaciones, modularidad, eficiencia algorítmica, seguridad, facilidad de prueba, uso apropiado de patrones de diseño, etc.

Existen diferentes estándares internacionales para la calidad de código, por ejemplo:

- ISO/IEC 25010: También conocido como SQuaRE (Software product Quality Requirements and Evaluation), establece un marco de referencia para la calidad de código dividido en 8 áreas.
- ISO 9001: Requisitos genéricos y aplicables para cualquier organización.
- ISO 10005 (2018): Plan de calidad para todo el ciclo de vida.
- ISO 33000: También conocido como SPICE (Software Process Improvement and Capability Determination), hace seguimiento a la evolución del sistema para identificar puntos de mejora.
- ISO 12207: Estándar para el seguimiento de la evolución de los procesos durante el ciclo de vida.
- IEEE 730 (2002): Define que es software de calidad y establece un plan para asegurar la calidad del software.
- ISO 5055: Mide debilidades críticas del sistema, enfocandose en seguridad, confianza, rendimiento y mantenibilidad.

ISO/IEC 25010

La calidad de código no existe de forma aislada, es parte de la calidad general del software, lo que termina influyendo en la experiencia final del usuario. Para contextualizar este concepto, el estándar ISO/IEC 25010 define el modelo de calidad del producto de software mediante un conjunto de características y subcaracterísticas que permiten evaluar el software de forma integral. Aunque el modelo evalúa la calidad del producto completo, varias de sus dimensiones dependen directamente de la calidad interna del código fuente.

La siguiente tabla muestra estás características:

CALIDAD DEL PRODUCTO SOFTWARE								
ADECUACIÓN FUNCIONAL	EFICIENCIA DE DESEMPEÑO	COMPATIBILIDAD	CAPACIDAD DE INTERACCIÓN	FIABILIDAD	SEGURIDAD	MANTENIBILIDAD	FLEXIBILIDAD	PROTECCIÓN
COMPLETITUD FUNCIONAL CORRECCIÓN FUNCIONAL PERTINENCIA FUNCIONAL iso25000.com	COMPORTAMIENTO TEMPORAL UTILIZACIÓN DE RECURSOS CAPACIDAD	COEXISTENCIA INTEROPERABILIDAD	RECONOCIBILIDAD DE ADECUACIÓN APRENDIZABILIDAD OPERABILIDAD PROTECCIÓN FRENTE A ERRORES DE USUARIO INVOLUCRACIÓN DEL USUARIO INCLUSIVIDAD ASISTENCIA AL USUARIO AUTO-DESCRIPTIVIDAD	AUSENCIA DE FALLOS DISPONIBILIDAD TOLERANCIA A FALLOS RECUPERABILIDAD	CONFIDENCIALIDAD INTEGRIDAD NO-REPUDIO RESPONSABILIDAD AUTENTICIDAD RESISTENCIA	MODULARIDAD REUSABILIDAD ANALIZABILIDAD CAPACIDAD DE SER MODIFICADO CAPACIDAD DE SER PROBADO	ADAPTABILIDAD ESCALABILIDAD INSTALABILIDAD REEMPLAZABILIDAD	RESTRICCIÓN OPERATIVA IDENTIFICACIÓN DE RIESGOS PROTECCIÓN ANTE FALLOS ADVERTENCIA DE PELIGRO INTEGRACIÓN SEGURA

Fig 1. Modelo de calidad del producto software según ISO/IEC 25010 (2024)

Relación del ISO/IEC 25010 con la calidad de código

La calidad de código puede evaluarse con distintas métricas y herramientas, pero su propósito final es contribuir con la calidad del producto final. Es decir, un código bien estructurado, legible, modular y eficiente no es un fin en sí, sino el medio que permite que el software sea mantenible, seguro, escalable y confiable. Por ende, la calidad técnica debe alinearse explícitamente con los atributos del modelo internacional.

Atributos de calidad de código

La ISO/IEC 25010 define los siguientes atributos de calidad de código:

Mantenibilidad

Asegurar que el código existente sea fácil de modificar y actualizar

- Estructura del código
- Complejidad ciclomática
- Duplicación
- Claridad de diseño
- Principios SOLID
- Patrones Arquitectonicos

Arquitectura de Software

Universidad Javeriana, Noviembre 2025

Juan Romero, Kamilt Diaz, Pablo Quintero

Seguridad

Minimizar las posibles vulnerabilidades del programa para disminuir el riesgo de ataques.

- Validación de entradas
- Manejo de excepciones
- Gestión de secretos
- Dependencia de librerías con vulnerabilidades

Fiabilidad

Asegurar que el sistema entero sea capaz de ejecutar sus tareas sin errores, y sea capaz de manejar errores de entrada o salida sin bloquearse.

- manejo correcto de errores
- robustez del código
- tests automatizados
- mecanismos de fallbacks

Eficiencia

Minimizar todo lo posible el uso de recursos, ya sea tiempo o memoria.

- Eficiencia algorítmica
- Estructura de los datos
- Paralelismo
- Uso de memoria

Eficiencia algorítmica

La eficiencia de un algoritmo, también llamado complejidad, hace referencia a su consumo de recursos, ya sea el tiempo de ejecución o el consumo de memoria. El análisis de la eficiencia en tiempo es el más común, realizado mediante la notación de la O grande (Big O Notation),

Big O mide el tiempo que le toma al algoritmo realizar una tarea en el peor de los casos a medida que aumenta el tamaño de su entrada. Por ejemplo, cuánto tiempo le toma a un algoritmo encontrar el número más grande de un arreglo de tamaño N, dado que en el peor caso, se debe revisar cada objeto dentro del arreglo, se dice que este algoritmo tiene una complejidad de O(n).

Big O tiene diferentes notaciones, cada una representado como el tiempo del algoritmo cambia a medida que el tamaño de N aumenta, unos ejemplos son:

- O(1): tiempo constante
- O(n): tiempo lineal
- O(log n): tiempo logarítmico
- O(n^2): tiempo cuadrático
- O(c^n): tiempo exponencial

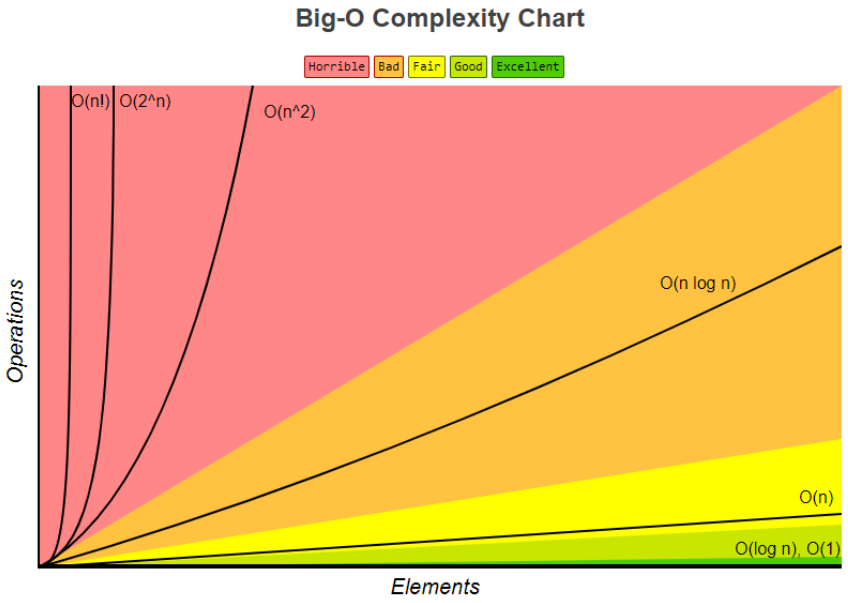

Fig 2. Visualización gráfica de la Big O

Compatibilidad

Asegurar que el sistema es capaz de operar sin problemas independientemente del entorno donde se encuentre.

- Modularidad
- Uso de API limpias
- Separación de responsabilidades
- Estándares de comunicación

Arquitectura de Software

Universidad Javeriana, Noviembre 2025

Juan Romero, Kamilt Diaz, Pablo Quintero

Políticas y buenas prácticas de desarrollo

Seguir buenas prácticas de desarrollo ayuda a que el proceso de creación del producto sea más ordenado y simple, además de ayudar a asegurar la calidad del producto final y que satisfaga las necesidades del cliente.

Una de las prácticas más importantes al momento de desarrollar código es mantenerlo legible. Al computador no le importa que una variable se llame X, pero en un futuro, alguien deberá mantener ese código, ya sea el mismo desarrollador u otro, y entre más fácil sea entender el código escrito, más eficiente será todo desarrollo futuro. Esto se puede conseguir por medio de ciertas prácticas:

- Código autodocumentado: el código debe ser lo suficientemente fácil de entender para que no sea necesario tener que explicarlo
- Refactorización constante: reescribir y reestructurar el código existente permite que sea más flexible, simplificando su mantenimiento y haciendo que sea más fácil de adaptar para posibles cambios en los requisitos.
- Reducir deuda técnica: la deuda técnica de un sistema representa el costo futuro causado por problemas relacionados con documentación, diseño o atajos tomados durante el desarrollo. A medida que la deuda técnica se acumula, es más difícil mantener el código.

Adicionalmente, existen otros principios aparte de SOLID para ayudar a que el código sea mantenible:

- KISS (Keep It Simple, Stupid): No sobre complicar el código. No es necesario crear un microservicio o un sistema distribuido cuando una simple función o un monolito pueden cumplir la misma tarea.
- DRY (Don’t Repeat Yourself): No escribir código duplicado. Código duplicado es más código que debe ser mantenido.
- YAGNI (You Ain’t Gonna Need It): Desarrollar únicamente el código que se va a utilizar. No agregar funcionalidades para requisitos futuros antes de que sean requeridos.

Herramientas

Las siguientes herramientas fueron implementadas:

Análisis estático - SonarQube

También llamados Linters, estas herramientas realizan un análisis continuo del código mientras el desarrollador trabaja sin necesidad de compilación, enfocándose en calidad general, errores y vulnerabilidades.

CI/CD Pipeline - Qodana

Estas herramientas se integran en la pipeline de CI/CD del repositorio de elección para realizar las tareas de análisis cada vez que se realiza un PR a la rama principal.

Configuración

SonarQube + Visual Studio

1. Abrir Visual Studio y dirigirse a la sección de extensiones

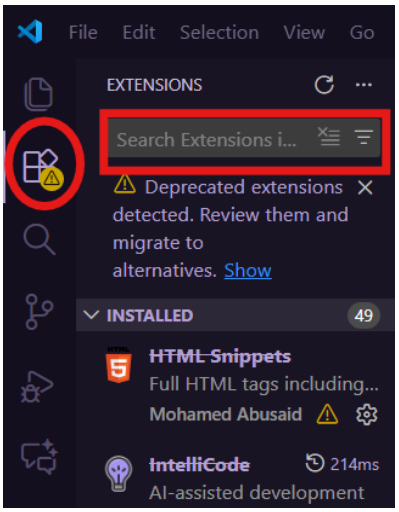

Fig 3. Menú de extensiones VSCode

2. Buscar SonarQube en la barra de búsqueda y dar click en SonarQube for IDE y luego en instalar.

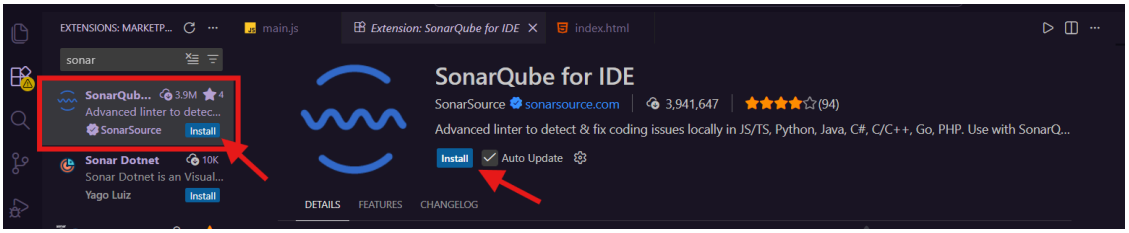

Fig 4. Extensión SonarQube VSCode

Una vez hecho click en instalar se nos abrirá la siguiente ventana en la que tendremos que darle click al botón azul para confirmar la instalación.

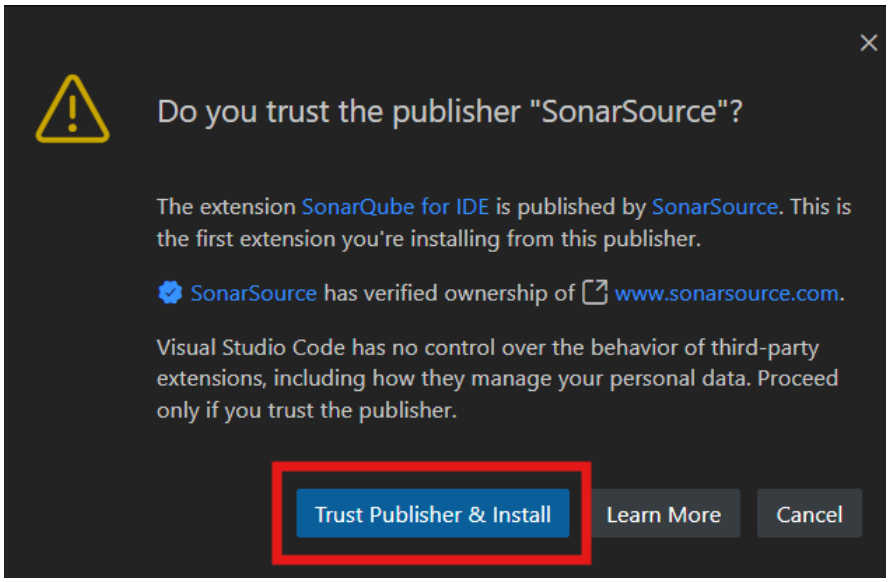

Fig 5. Confirmación instalación de la instalación.

Finalmente, pasados unos minutos terminará la instalación y abajo a la derecha nos aparecerá un mensaje pidiendo recargar la página actual para terminar la instalación.

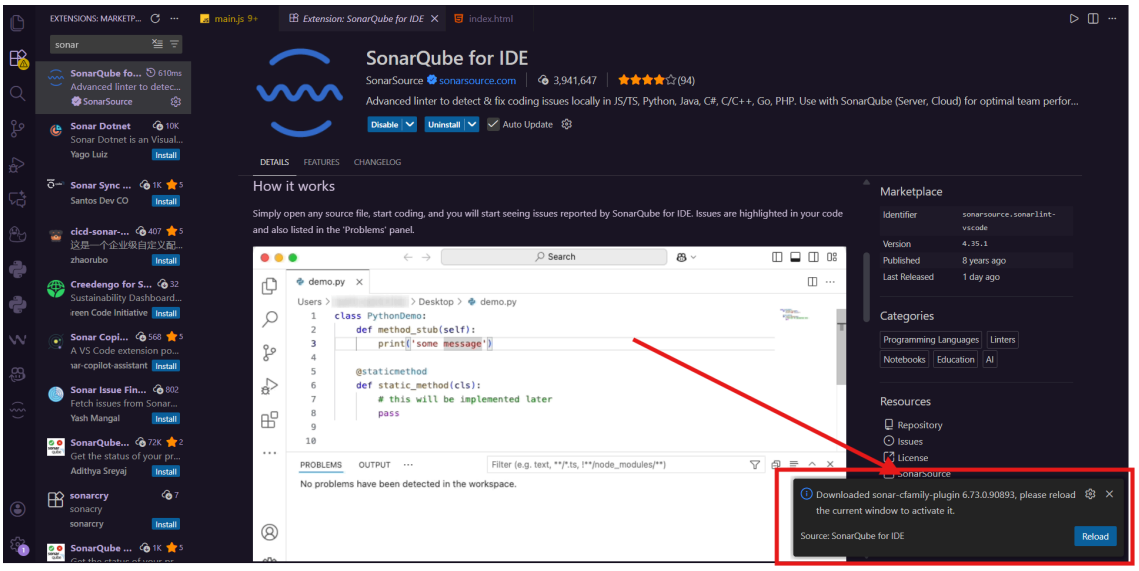

Fig 6. Finalización de la instalación.

Esto nos añadirá un nueva sección a la barra de herramientas llamada SonarQube setup, en donde podremos iniciar sesión para tener acceso a más lenguajes y reglas de

seguridad, para efectos prácticos se seguirá la demostración con la versión base de la extensión.

El proceso de instalación para PMD es similar.

Qodana + GitHub Actions

El uso de Qodana requiere una licencia, aunque es posible obtener una prueba gratuita por 60 días. El precio de la licencia básica es de 5$ mensuales por cada contribuidor activo en el proyecto, con un mínimo requerido de 3 contribuidores.

Para empezar, en nuestra cuenta de JetBrains creamos un nuevo proyecto

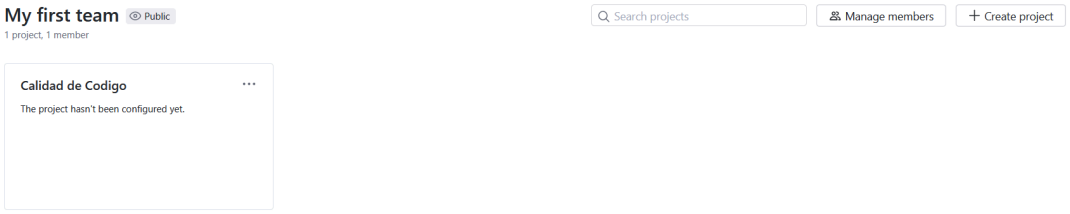

Fig 7. Proyectos en JetBrains

Escogemos el CI/CD que vamos a utilizar, aunque también es posible crear un archivo Docker compatible con cualquier entorno.

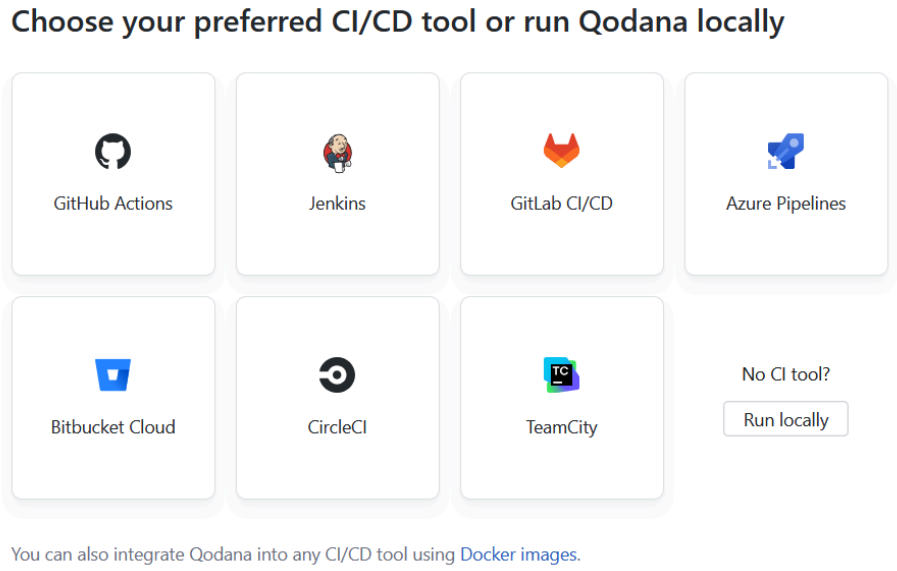

Fig 8. Selección de la herramienta de CI/CD

Al seleccionar GitHub actions, deberemos autorizar a Qodana el acceso a la información de nuestra cuenta. Como es la primera vez que usamos Qodana, también debemos agregar Qodana Cloud en nuestra cuenta para tener acceso a nuestros repositorios.

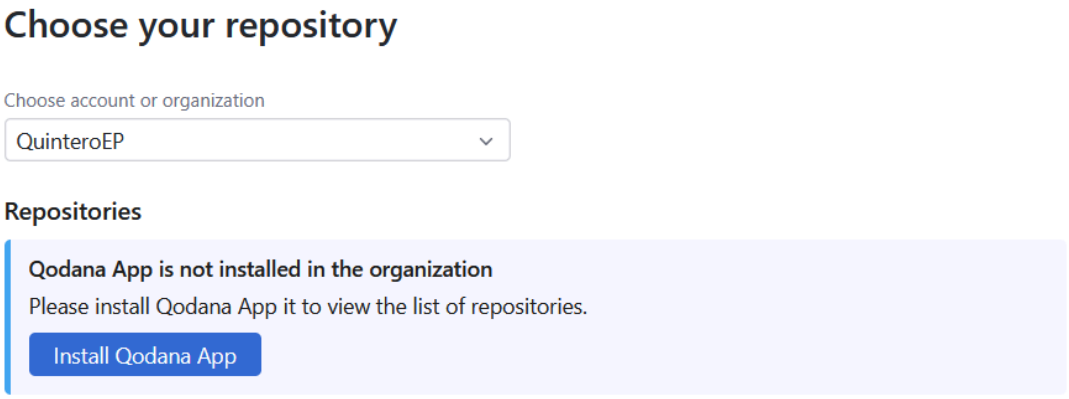

Fig 9. Agregar Qodana Cloud

Una vez instalado, Todos nuestros repositorios, o los que hayamos seleccionado durante la instalación, aparecerán en la interfaz de Qodana

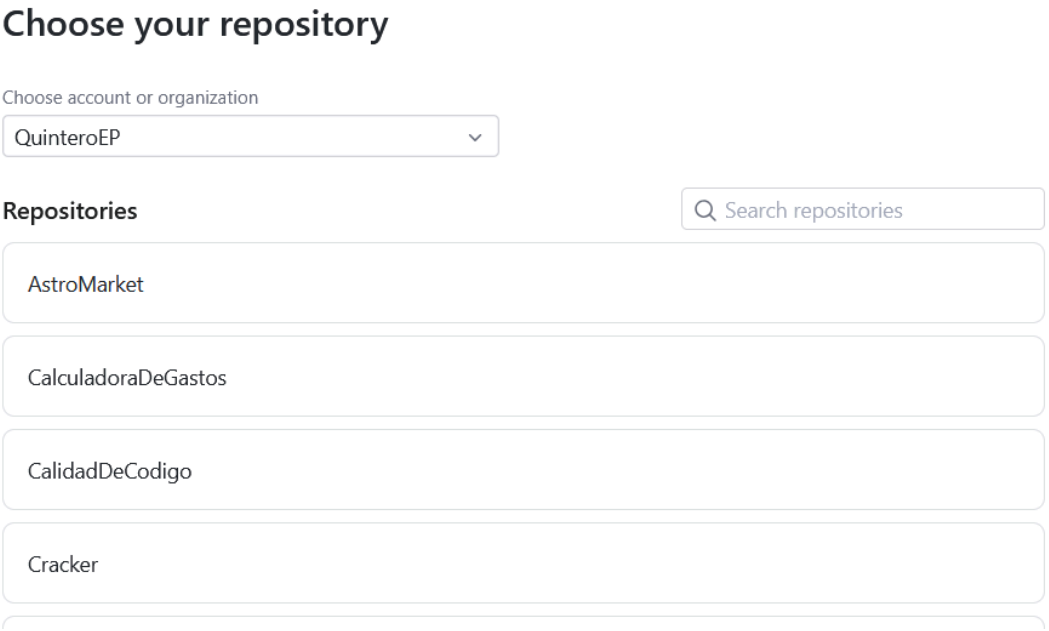

Fig 10. Repositorios disponibles

Podemos realizar la configuración de forma automática o manual, en cualquier caso, se configurará un linter de nuestra elección para el proyecto y se agregará un archivo Yaml al repositorio, un token como secret y se creará una Pull Request en nuestro repositorio en la rama principal.

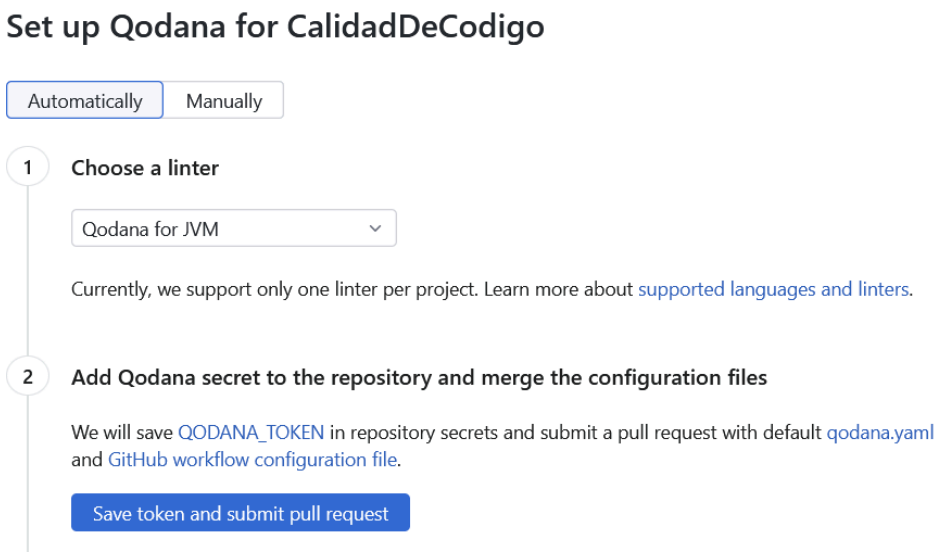

Fig 11. Configuración automática de Qodana

El linter seleccionado es para Java, aunque existen múltiples linters para diferentes lenguajes. Una vez terminada la configuración podremos encontrar la PR en el repositorio.

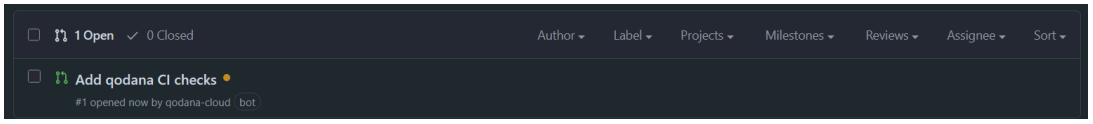

Fig 12. Job inicial de Qodana

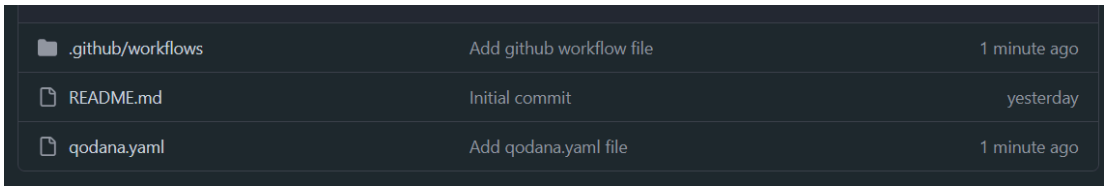

Fig 13. Directorio de Github Worflows creado automáticamente

La configuración automática de Qodana ya nos crea el Job de Workflows.

qodana-bot Add github workflow file

Code Blame 28 lines (27 loc) · 816 Bytes

```yaml
name: Qodana
on:
  workflow_dispatch:
  pull_request:
  push:
    branches: # Specify your branches here
      - main # The 'main' branch
      - 'releases/*' # The release branches

jobs:
  qodana:
    runs-on: ubuntu-latest
    permissions:
      contents: write
      pull-requests: write
      checks: write
    steps:
      - uses: actions/checkout@v3
        with:
          ref: ${{ github.event.pull_request.head.sha }}  # to check out the actual pull request commit, not the merge commit
          fetch-depth: 0  # a full history is required for pull request analysis
      - name: 'Qodana Scan'
        uses: JetBrains/qodana-action@v2025.2
        with:
          pr-mode: false
        env:
          QODANA_TOKEN: ${{ secrets.QODANA_TOKEN_1143395227 }}
          QODANA_ENDPOINT: 'https://qodana.cloud'
```

Fig 14. Contenido del Job

A partir de ahora, cada Pull Request que reciba nuestra rama principal será evaluada por Qodana.

Al abrir nuestro proyecto en Jetbrains encontraremos la interfaz de Qodana.

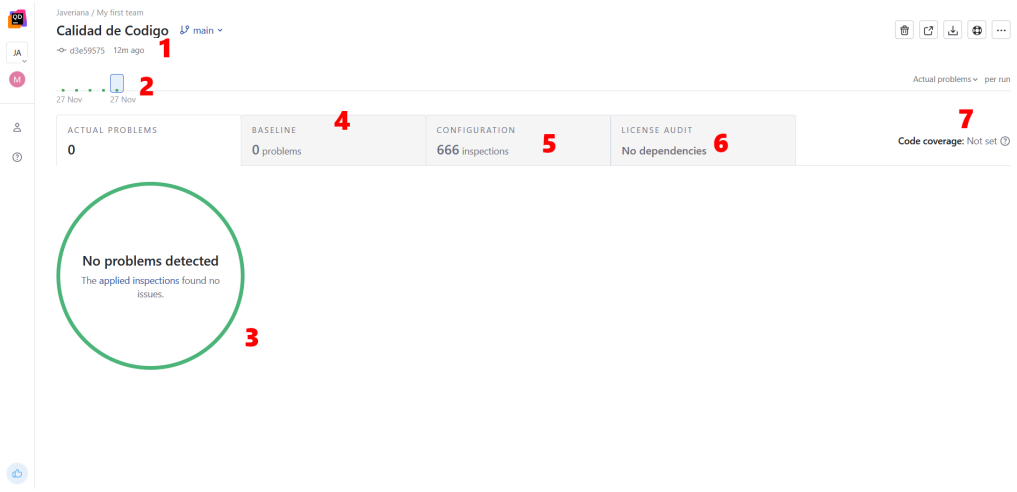

Fig 15. Interfaz de Qodana

Aquí encontramos los siguientes elementos:

1. El repositorio y la rama actual
2. Historial de análisis
3. Los errores detectados
4. Errores que han sido marcados para corregir en un futuro
5. Tipos de error que Qodana está analizando
6. Auditoria de licencias
7. Configuración de cobertura de código

Si seleccionamos la sección Configuration, podemos agregar o eliminar los tipos de errores que Qodana detecta. Al hacer una modificación tendremos que descargar el nuevo archivo Yaml, reemplazarlo en el repositorio

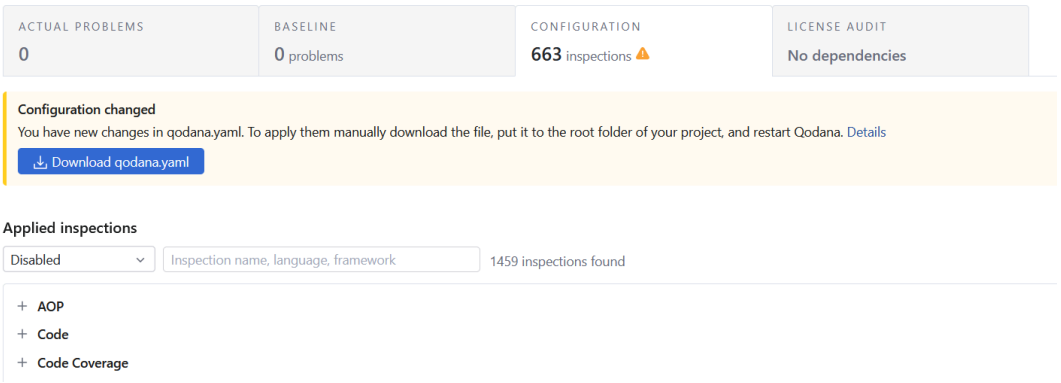

Fig 16. Agregar errores a Qodana

Ejemplo práctico

SonarQube

En las siguientes fotos podemos ver varias cosas, primero nos subraya todos los errores en el editor, en la barra de desplazamiento y en la visualización general del código, además en el panel inferior (Console, problems, etc) se pueden ver los problemas detectados por sonarQube y aún más interesante se crea una nueva pestaña dedicada a la aplicación, en la que nos muestra los errores y al darle click a cualquier error nos muestra el tipo de error y una explicación muy completa de porque es un error y cómo tratarlo.

Fig 17. Funcionamiento general SonarQube

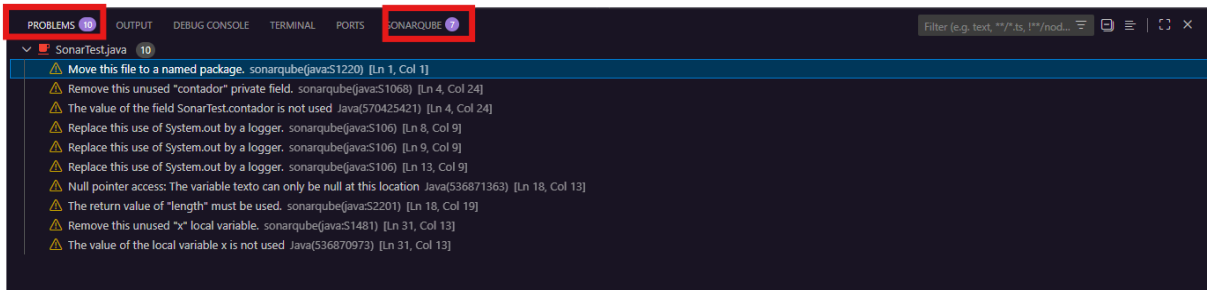

Fig 18. Panel de problemas

Fig 19. Panel dedicado SonarQube

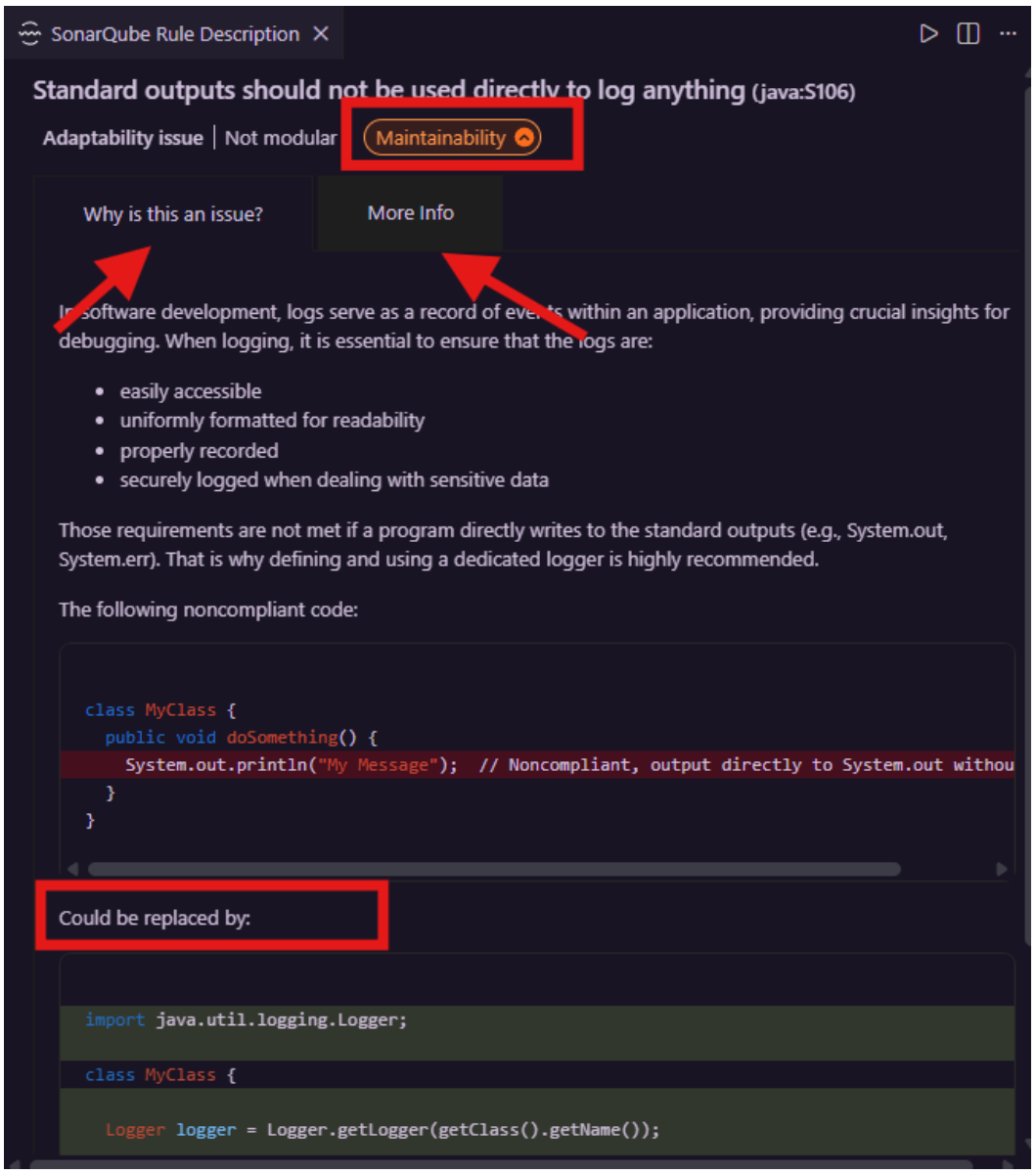

Fig 20. Panel de soluciones SonarQube.

Qodana

Hacemos PR de nuevo código hacia Main, este código es una simple función para sumar dos números.

```java
public static void main(String[] args) {
    double a, b;

    Scanner sc = new Scanner(System.in);

    System.out.print(s: "Enter first number: ");
    while (!sc.hasNextDouble()) {
        System.out.print(s: "Please enter a valid number: ");
        sc.next();
    }
    a = sc.nextDouble();

    System.out.print(s: "Enter second number: ");
    while (!sc.hasNextDouble()) {
        System.out.print(s: "Please enter a valid number: ");
        sc.next();
    }
    b = sc.nextDouble();
    sc.close();

    double result;
    result = a + b;
    System.out.print("Result: " + result);
}
```

Fig 21. Código sin errores

Al momento de realizar el PR, El Job de Qodana empezará

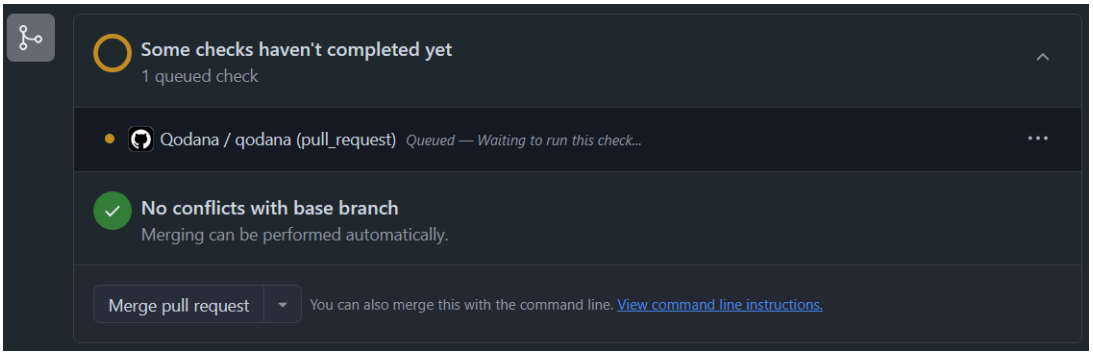

Fig 22. Job de Qodana al abrir PR

Después de un momento, obtendremos los resultados.

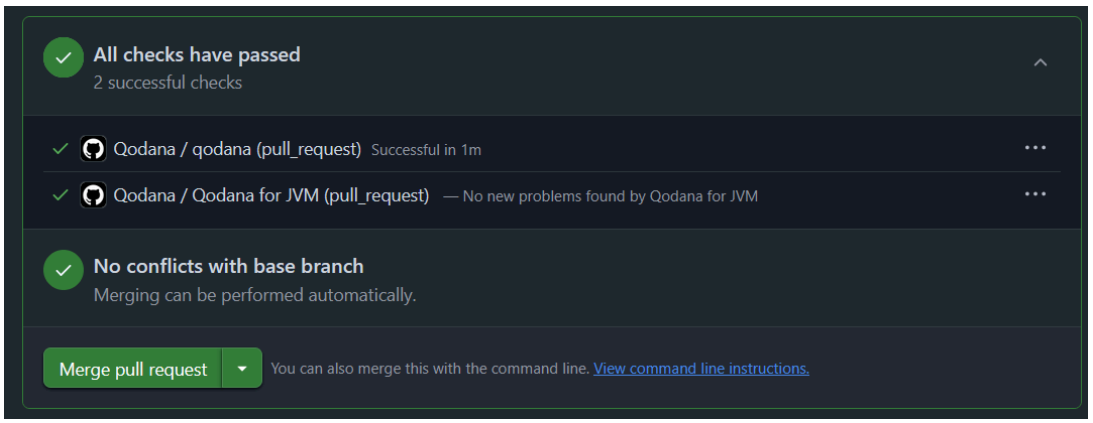

Fig 23. Job terminado

Al hacer merge, Qodana realizará otro análisis de forma automática

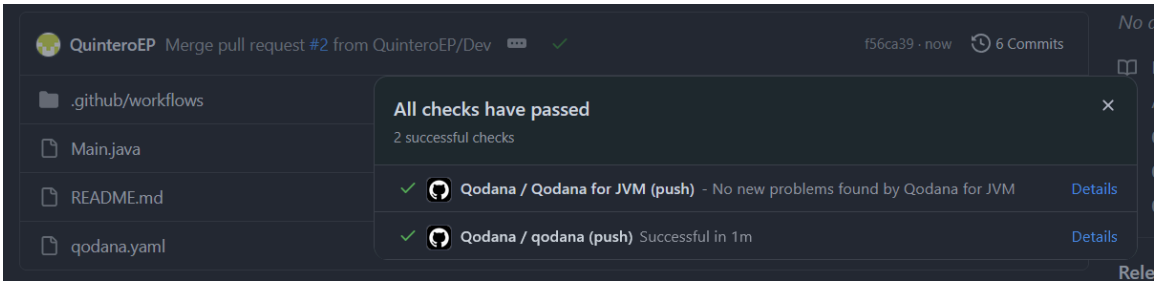

Fig 24. Resultados de Qodana

Si accedemos a nuestro proyecto de JetBrains en el navegador, podremos ver el resultado del análisis. Como podemos ver, ningún error fue encontrado.

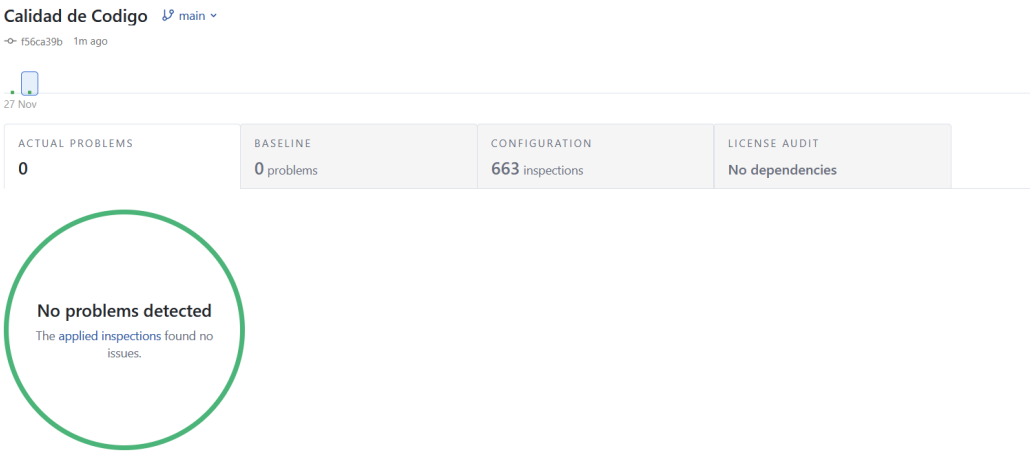

Fig 25. Interfaz de Qodana sin errores

Ahora, si agregamos código, esta vez con algunos errores que Qodana detecta, tales como imports sin usar o problemas de error handling, el check de Qodana durante el PR nos asiva de estos errores

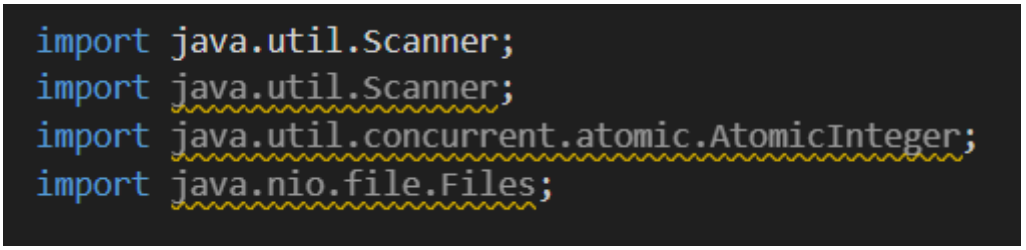

Fig 26. Imports no usados

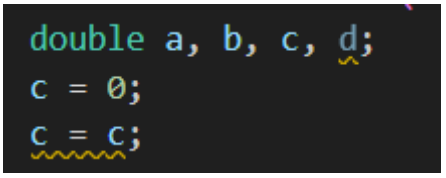

Fig 27. Variable sin usar

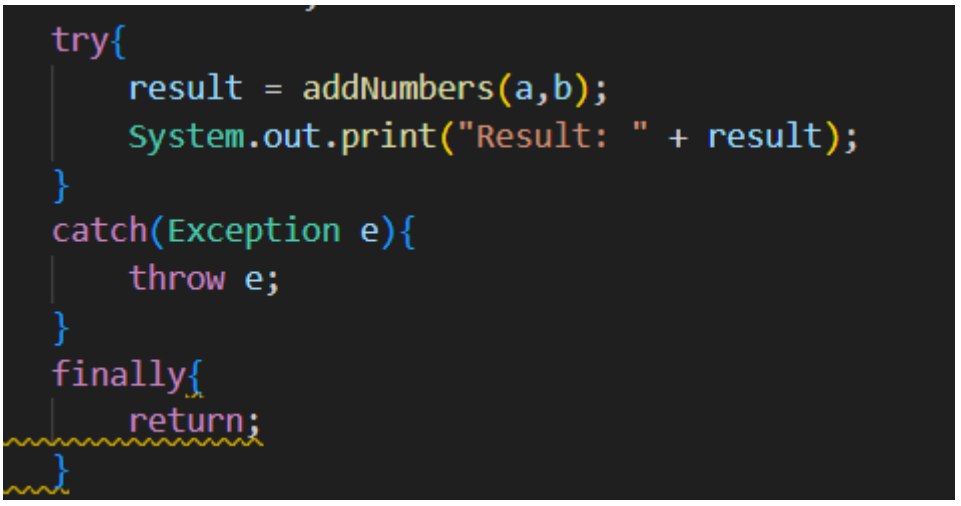

Fig 28. Catch Throw y return en Finally

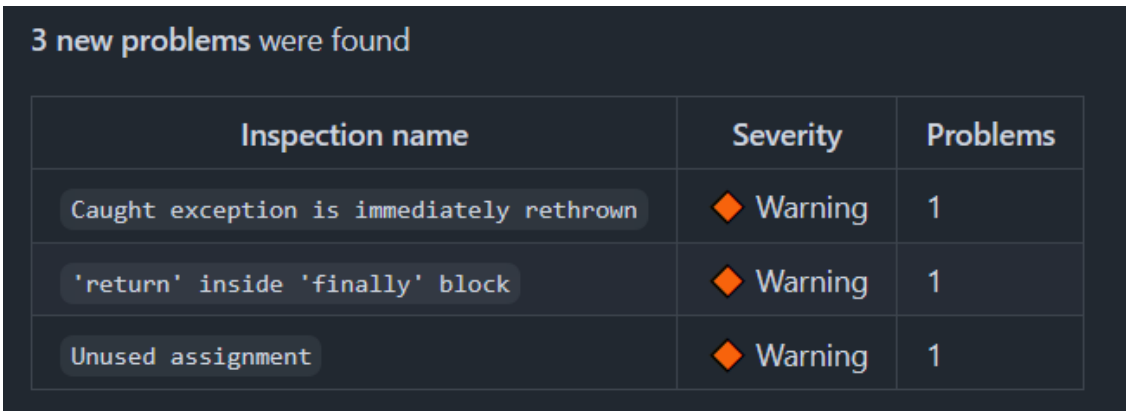

Fig 29. Alerta de Qodana

En la interfaz de Qodana, también podemos encontrar estos errores

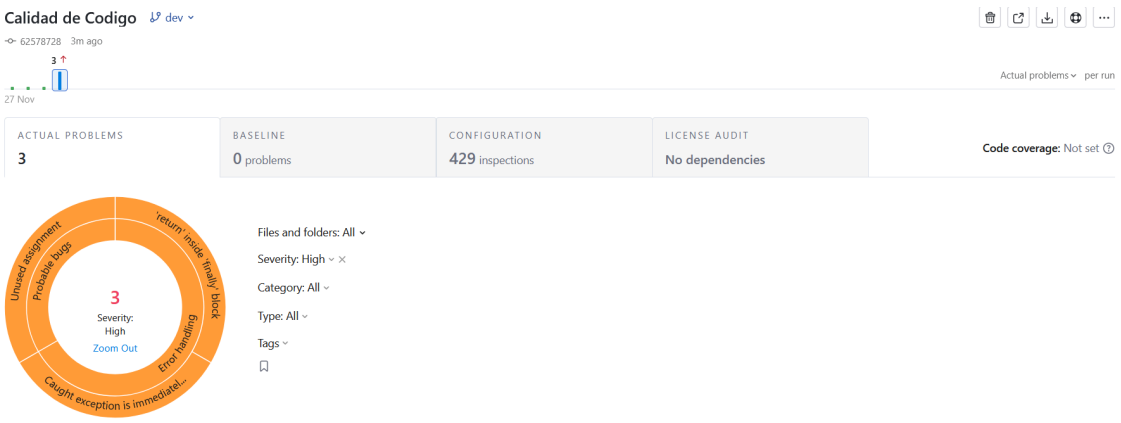

Fig 30. Interfaz de Qodana con los errores

lecciones aprendidas

Durante el desarrollo del trabajo entendimos que la calidad de código es un elemento central dentro de la arquitectura de software y no únicamente un conjunto de buenas prácticas aisladas. Identificamos que la calidad se construye desde decisiones técnicas tan simples como nombrar adecuadamente una variable hasta la integración de herramientas avanzadas de análisis estático o pipelines automatizados de CI/CD.

También aprendimos que estándares como ISO/IEC 25010 permiten contextualizar la calidad del código dentro de un marco más amplio, relacionado explícitamente con la mantenibilidad, seguridad, fiabilidad y eficiencia con la experiencia final del usuario. Explorar herramientas como SonarQube y Qodana nos permitió reconocer errores que normalmente pasarían desapercibidos y entender mejor cómo se mide la deuda técnica y la

complejidad del sistema y comprendimos la importancia del análisis continuo: no basta con escribir código funcional, sino que es necesario evaluarlo, refactorizar y someterlo a procesos automáticos que garanticen su evolución y estabilidad.

Conclusiones

La calidad de código es un componente indispensable para garantizar un software sostenible, seguro y confiable. Un código bien diseñado no solo reduce costos de mantenimiento, sino que facilita la evolución del producto y disminuye riesgos asociados a fallas o vulnerabilidades. A partir del estándar ISO/IEC 25010, confirmamos que la calidad técnica influye directamente en atributos como mantenibilidad, seguridad, eficiencia y compatibilidad, por lo que debe tratarse como un pilar fundamental desde las primeras etapas del desarrollo.

El uso de herramientas de análisis estático y pipelines de CI/CD demostró ser una práctica efectiva para detectar problemas temprano y mantener un control continuo sobre el estado del código. SonarQube y Qodana no solo identifican errores, sino que explican sus causas y sugieren acciones correctivas, lo cual aporta un valor formativo importante para los desarrolladores.

Referencias

1. https://stackoverflow.com/questions/2307283/what-does-olog-n-mean-exactly
2. https://es.wikipedia.org/wiki/Eficiencia_algor%C3%ADtmica
3. https://iso25000.com/index.php/normas-iso-25000/iso-25010
4. Franca, Joyce & Soares, Michel. (2015). SOAQM: Quality model for SOA applications based on ISO 25010. ICEIS 2015 - 17th International Conference on Enterprise Information Systems, Proceedings. 2. 10.5220/0005369100600070.
5. https://innevo.com/blog/mejores-practicas-desarrollo-software
6. https://icariatechnology.com/la-calidad-del-software-y-sus-estandares-mas-importantes/
7. https://www.jetbrains.com/help/qodana/github.html