Natalia Marcinkowska, Maciej Maj, Damian Maślanka

Wstęp

SonarQube to platforma typu open source służąca do ciągłej analizy kodu statycznego pod względem jego jakości. Umożliwia automatyczne wykrywanie błędów, słabych punktów powodujących obniżenie bezpieczeństwa oraz tak zwanych "code smells".

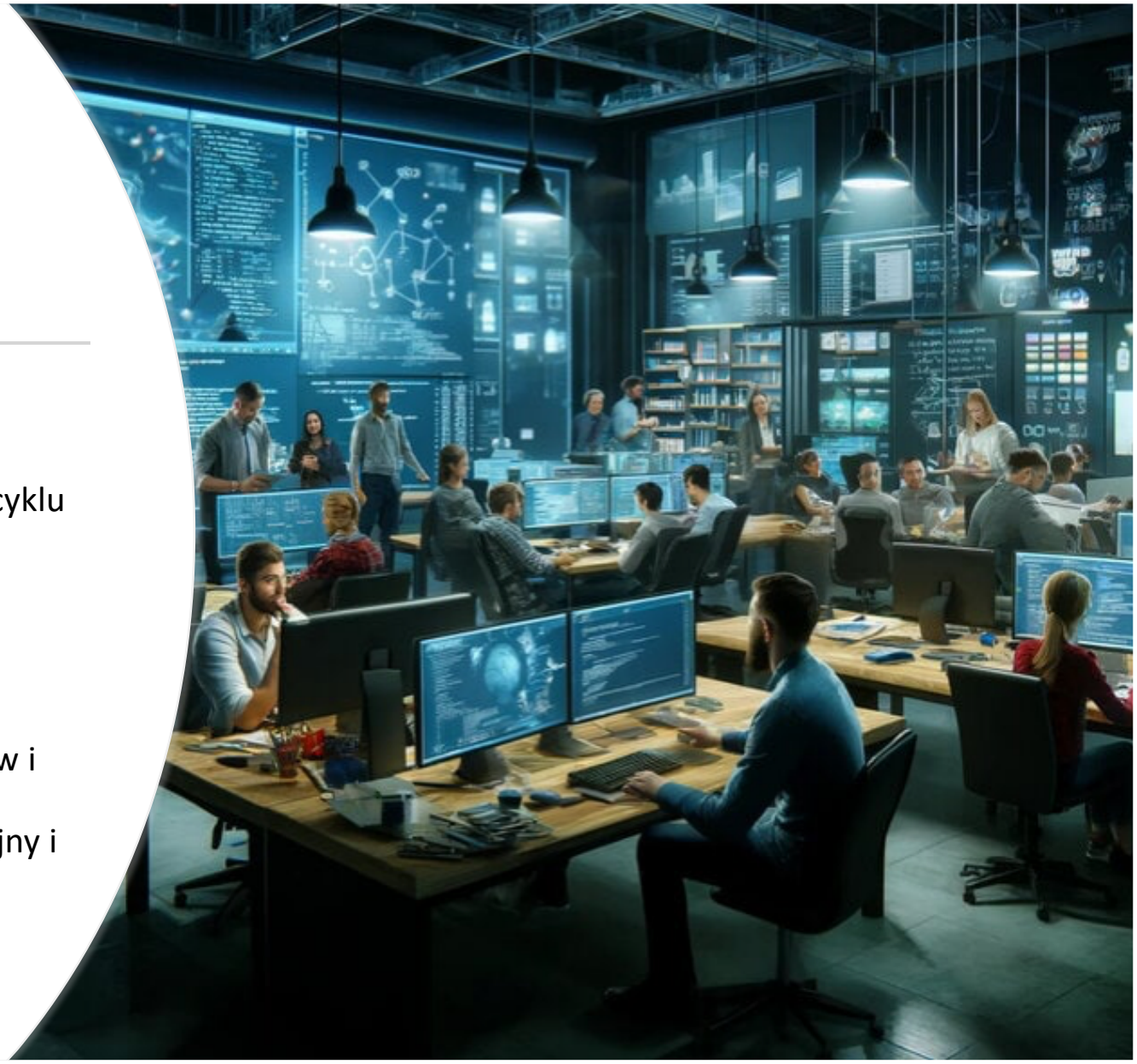
Wspiera ponad 30 różnych języków programowania, co czyni go wyjątkowym narzędziem analitycznym.

Cele

- **Poprawa jakości kodu** – SonarQube automatycznie analizuje kod, identyfikując problemy związane z jakością, takie jak duplikaty kodu, zła struktura czy nieefektywne rozwiązania.
- **Zapewnienie bezpieczeństwa** – Narzędzie pomaga wykrywać luki bezpieczeństwa i problemy związane z podatnościami, np. SQL Injection, Cross-Site Scripting (XSS) czy ataki typu Buffer Overflow.

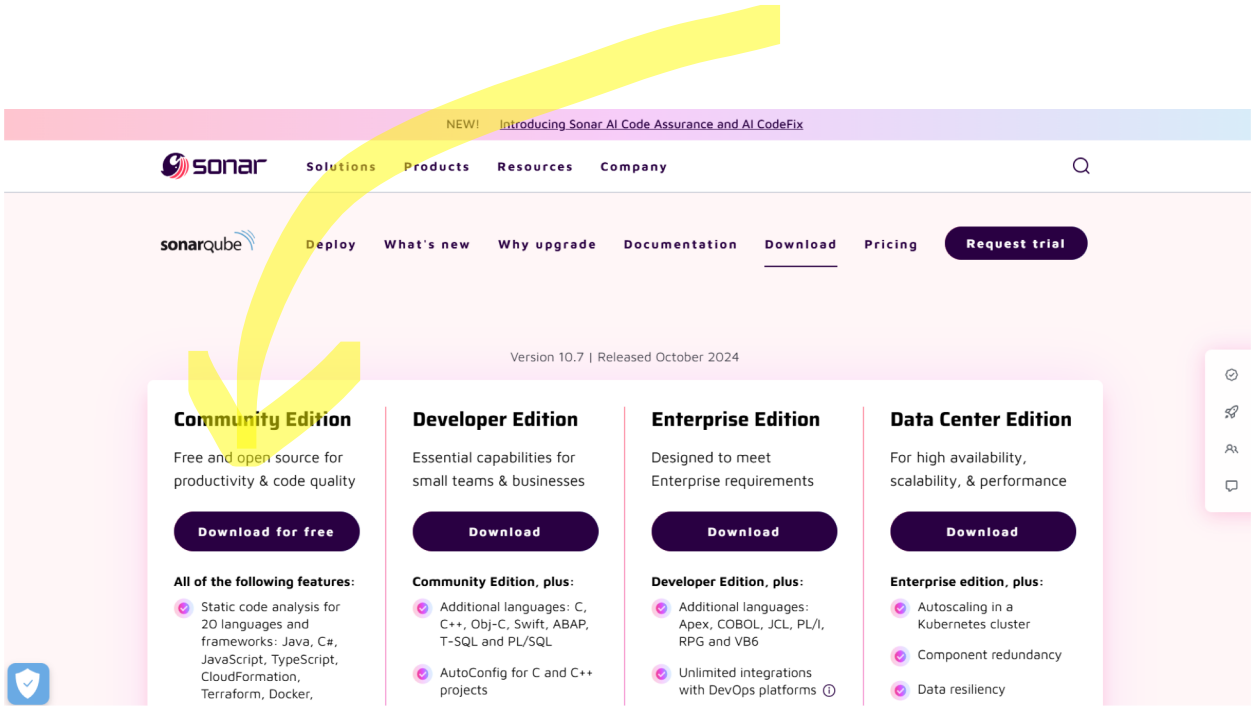
Cele ciąg dalszy

- **Automatyzacja i optymalizacja pracy** – Narzędzie pozwala na automatyczne przeprowadzanie analiz w różnych fazach cyklu życia aplikacji, dzięki czemu błędy są wykrywane wcześniej i mogą być szybciej korygowane.
- **Utrzymanie zgodności z najlepszymi praktykami** – SonarQube wspiera deweloperów w przestrzeganiu standardów i zasad programowania, co jest kluczowe w zespołach developerskich, aby kod był spójny i łatwy w utrzymaniu.



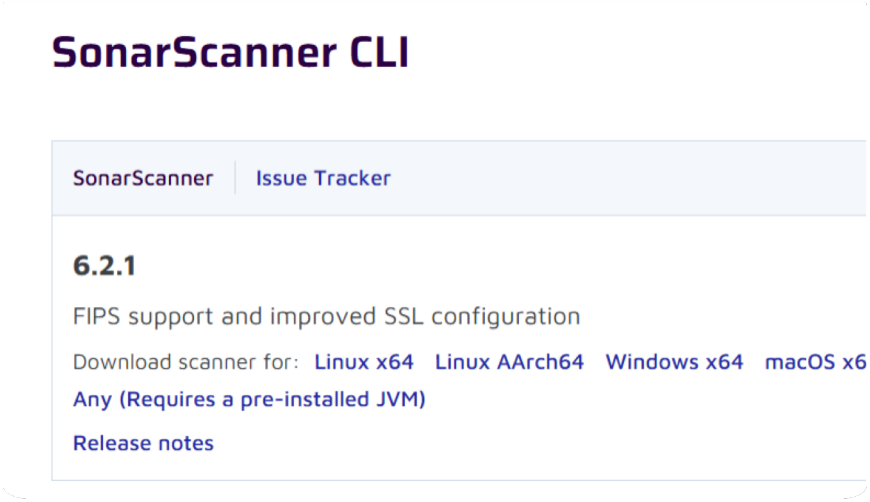
Instalacja

- Pobierz bezpłatną wersję SonarQube ze strony: https://www.sonarsource.com/products/sonarqube/downloads/
- Rozpakuj pobrany folder



SonarQube wymaga posiadania Javy w wersji 11, albo 17

Instalacja cd.



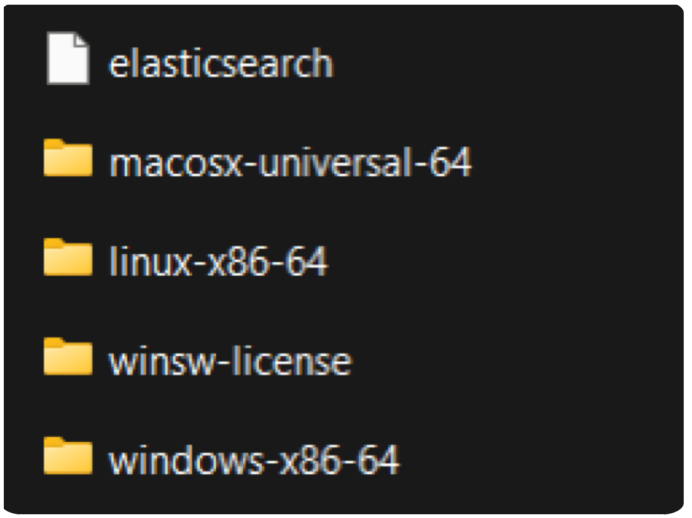
- Pobierz SonarScanner ze strony: https://docs.sonarsource.com/sonarqube/latest/analyzing-source-code/scanners/sonarscanner/, wybierając odpowiednie oprogramowanie.
- Rozpakuj
- Dodaj ścieżkę do katalogu bin SonarScannera do zmiennej środowiskowej PATH*.

*Wchodzimy w Ustawienia -> System -> Zaawansowane ustawienia systemu -> Zmienne środowiskowe -> W Zmiennych systemowych wybieramy PATH -> Edytuj

Instalacja cd.

- Wejdź w terminalu do katalogu bin SonarQube
- Wybierz folder z odpowiednim systemem operacyjnym i również do niego wejdź
- Aby uruchomić serwer SonarQube użyj komendy:

StartSonar.bat lub sonar.sh



Instalacja cd.

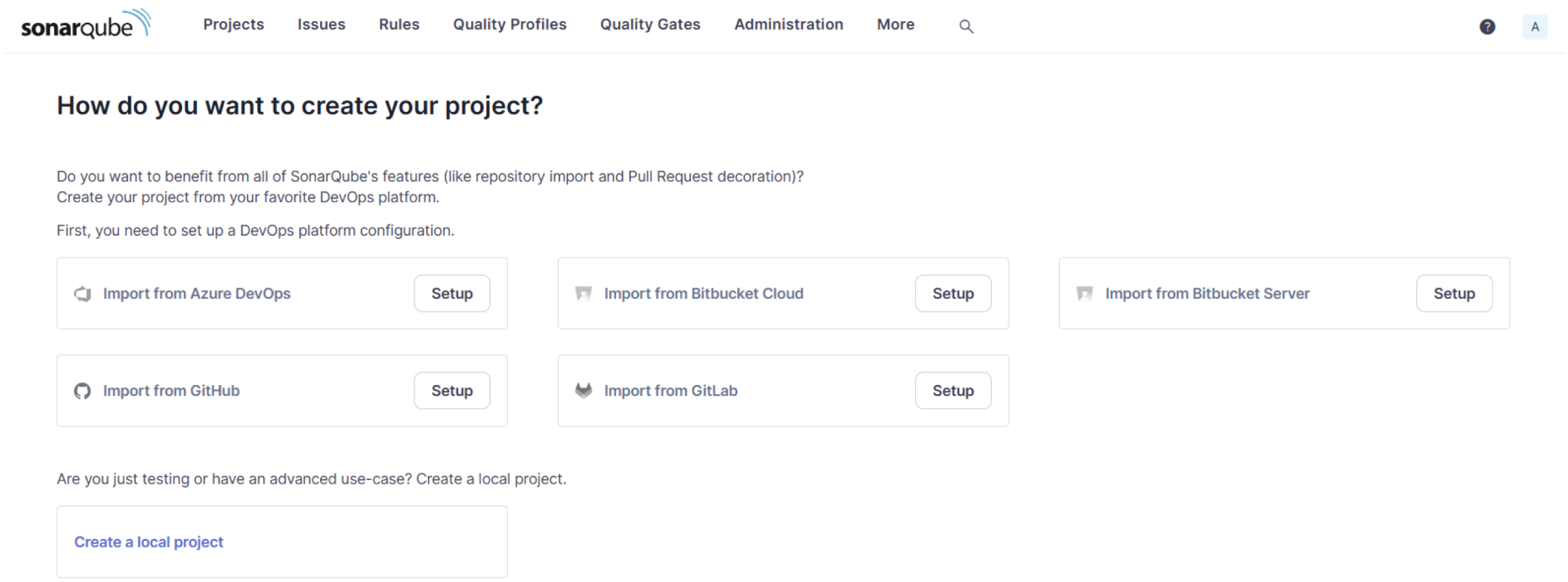
- Wejdź na stronę: http://localhost:9000
- Zaloguj się:

Login = admin

Hasło = admin

- Ustaw swoje nowe hasło

Strona SonarQube



Typy błędów

Bugs to błędy logiczne lub funkcjonalne w kodzie, które powodują, że program nie działa zgodnie z oczekiwaniami.

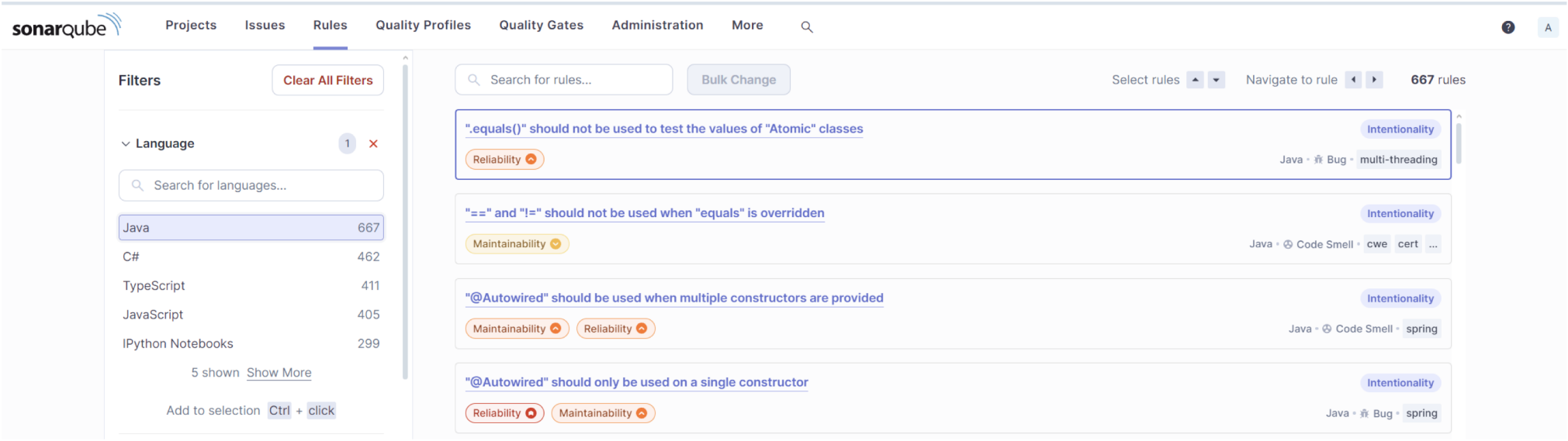
Vulnerabilities (podatności) to punkty w kodzie, które są narażone na ataki.

Code Smells (problemy stylistyczne lub projektowe) to fragmenty kodu, które choć nie powodują bezpośrednich błędów, wskazują na obszary wymagające poprawy pod kątem czytelności, wydajności lub utrzymania.

Każdy z tych problemów ma jeden z poziomów powagi (severity): blocker, critical, major, minor, info.

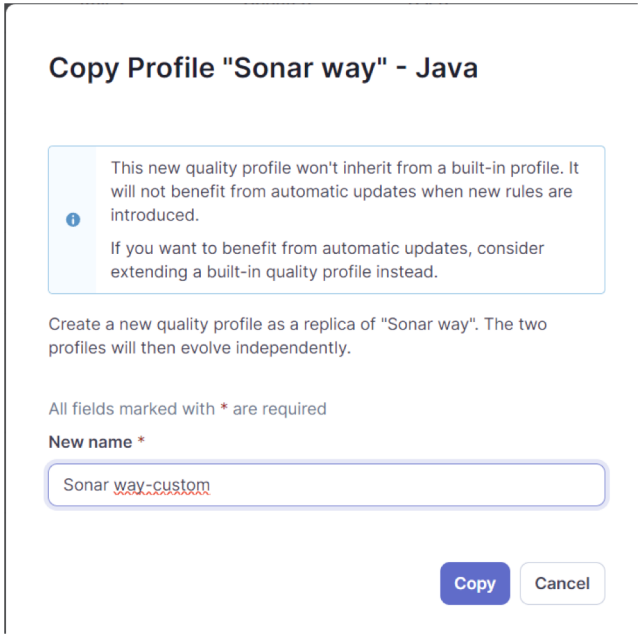
Rules

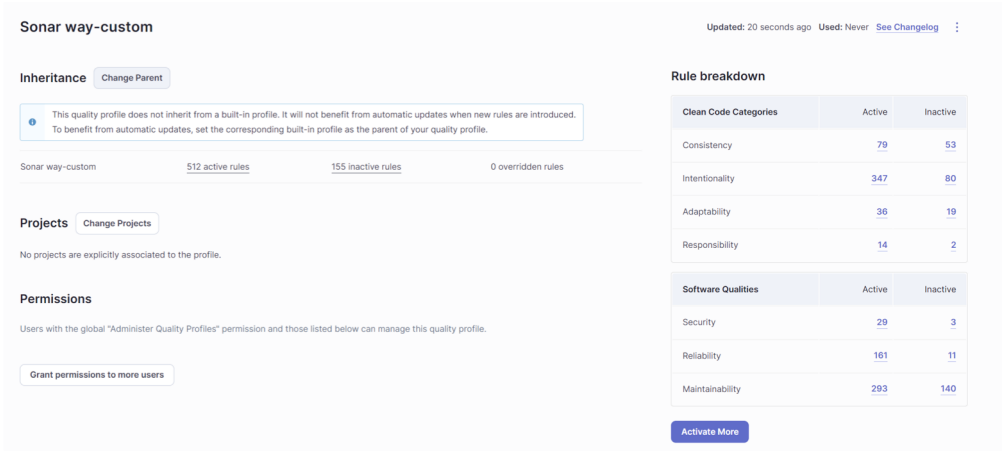
Zakładka "Rules" służy do zarządzania regułami, które SonarQube wykorzystuje do analizy kodu. Reguły definiują, jakie aspekty kodu są sprawdzane pod kątem jakości, bezpieczeństwa, wydajności, zgodności ze standardami oraz innych dobrych praktyk. Dzięki nim SonarQube może znaleźć w kodzie bugs, code smells, issues itd. Możemy filtrować je np. względem Javy.



Quality Profiles

Zakładka "Quality Profiles" pozwala na grupowanie reguł w zestawy, które są dostosowane do konkretnych technologii, typów projektów, lub wymagań. Profile jakości są stosowane do projektów, aby określić, jakie reguły mają być stosowane podczas analizy kodu.





Quality Gates

Zakładka "Quality Gates" umożliwia definiowanie kryteriów, które projekt musi spełnić, aby zostać uznanym za gotowy do wdrożenia. Są to zestawy warunków, które muszą być spełnione przez kod po analizie, aby można było uznać go za akceptowalny.

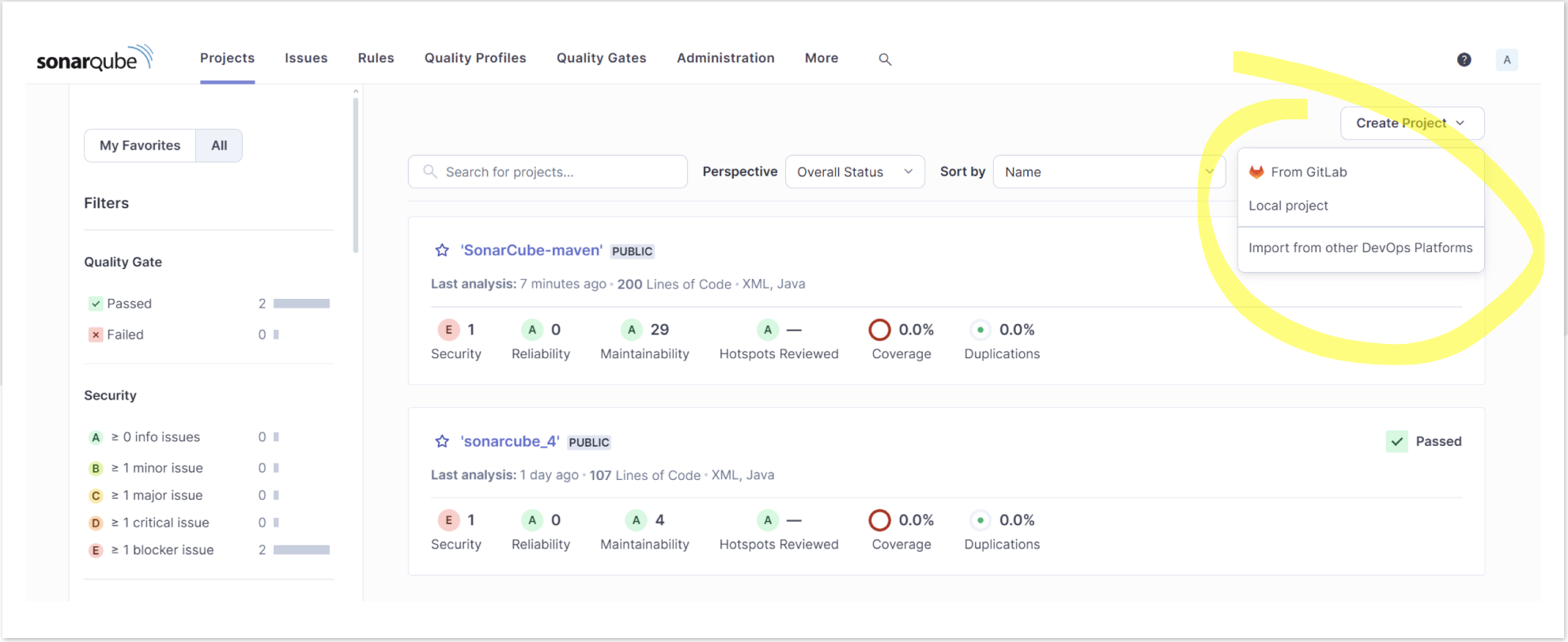
Conditions ?

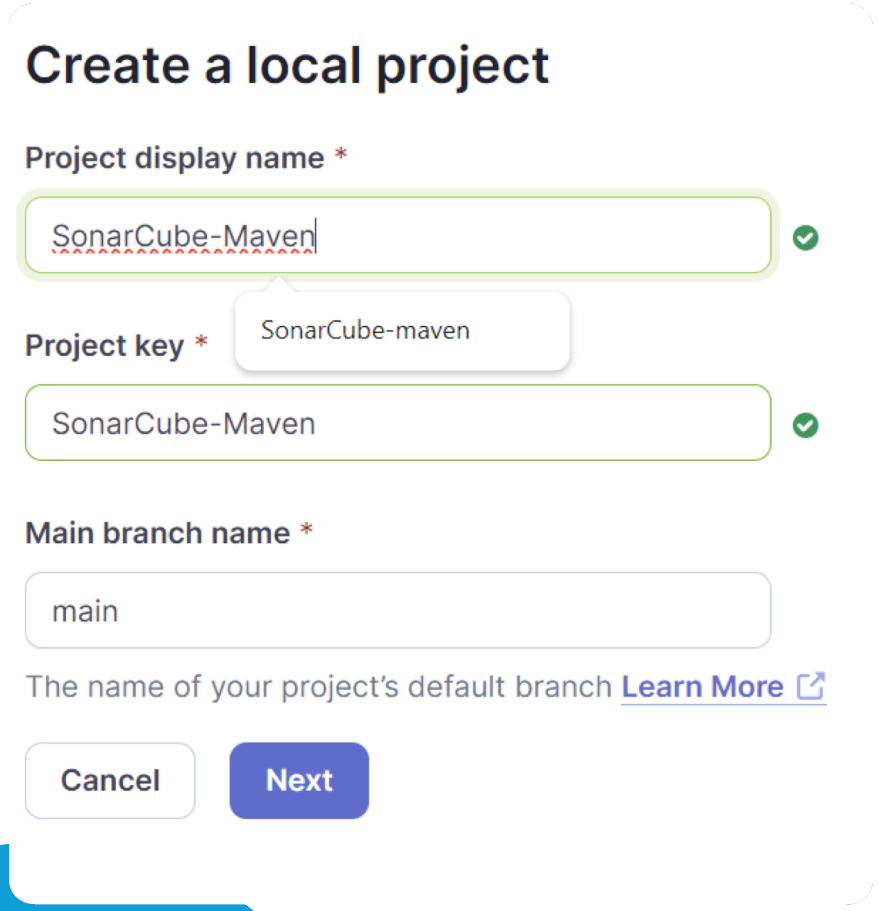
Conditions on New Code

Metric	Operator	Value
Issues	is greater than	0
Security Hotspots Reviewed	is less than	100%
Coverage	is less than	80.0%

Generacja raportu (Maven)

1. Dodaj projekt



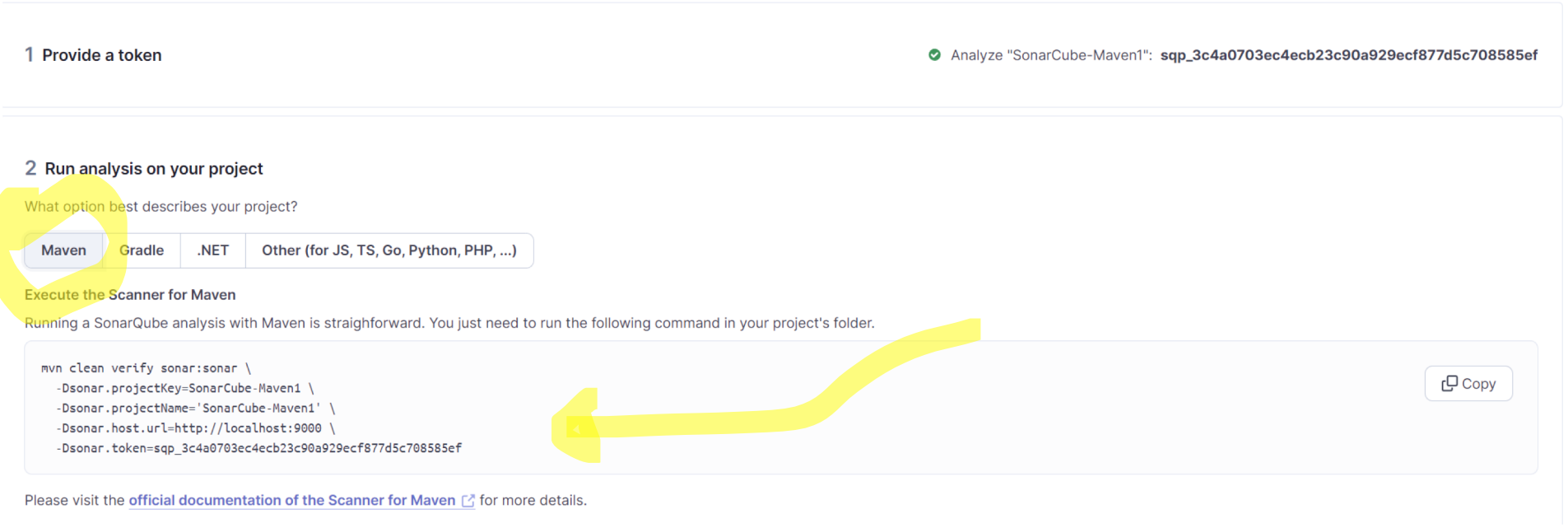


Generacja raportu (Maven)

2. Wygeneruj Token

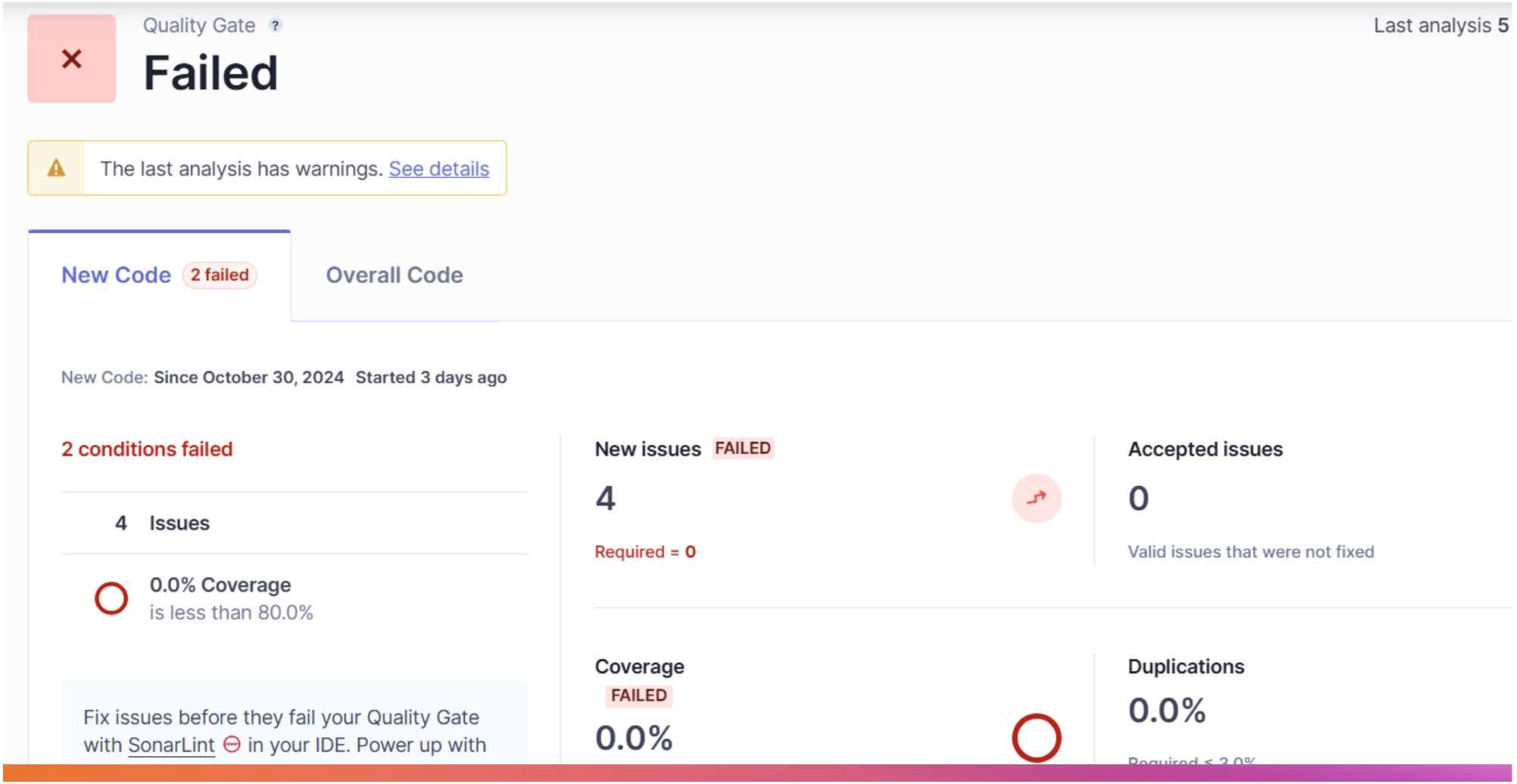
Generacja raportu (Maven)

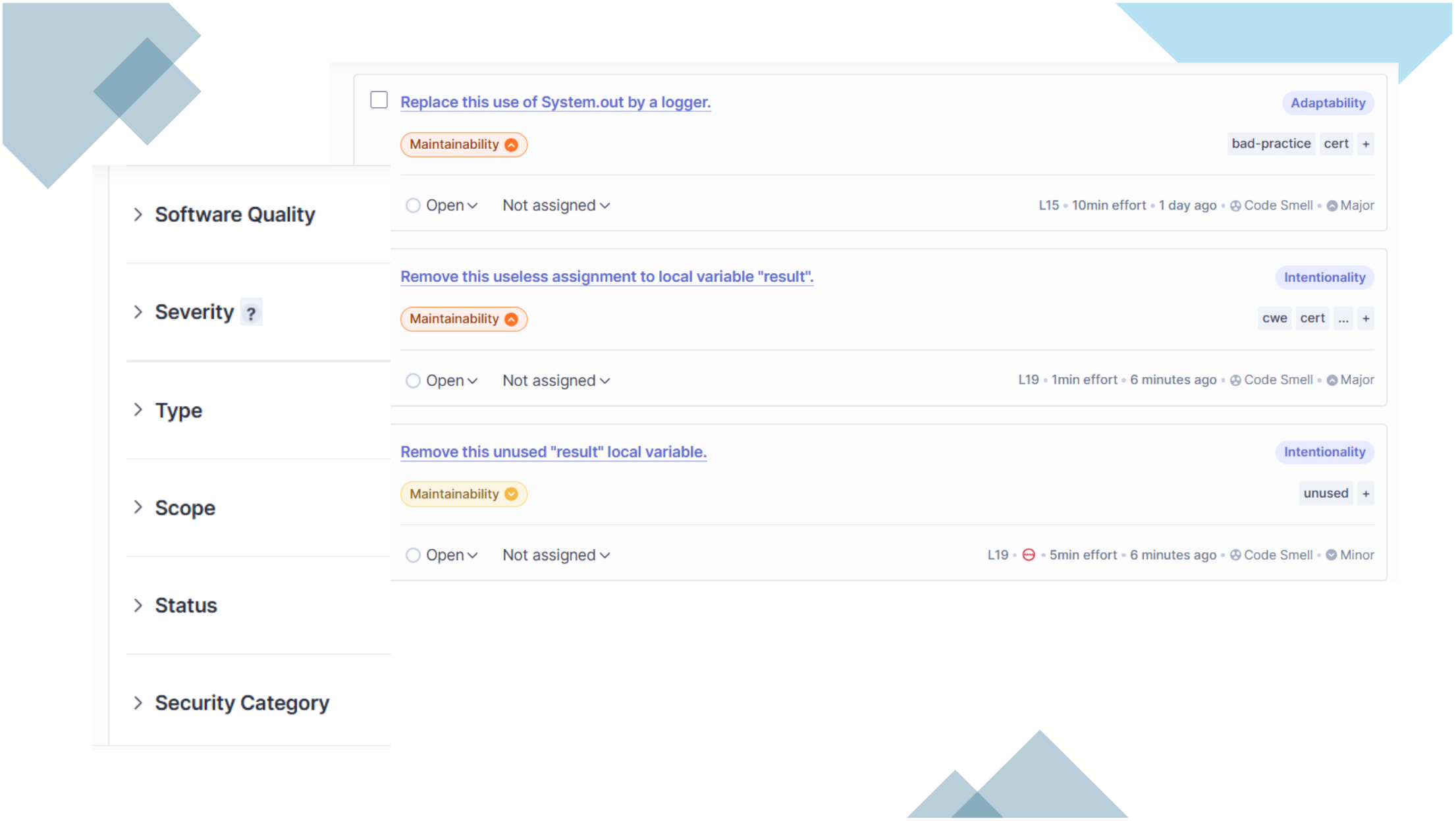
3. Wybierz opcje Maven, wejdź w konsoli do folderu ze swoim projektem i uruchom poniższą komende bez \



Przykładowy kod

```java
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class App {
    public static void main(String[] args) {
        System.out.println("Hello, World!");

        int additionResult = addNumbers( a: 5,  b: 10);
        System.out.println("Addition result: " + additionResult);

        int subtractionResult = subtractNumbers( a: 15,  b: 10);
        System.out.println("Subtraction result: " + subtractionResult);
    }

    private static int addNumbers(int a, int b) {  1 usage
        int result = a + b + 0;
        return a + b + 0;
    }

    private static int subtractNumbers(int a, int b) {  1 usage
        int result = a + b;
        result = a + b;
        return result;
    }
}
```


Quality Gate
Failed
Last analysis 5
The last analysis has warnings. See details
New Code 2 failed
Overall Code
New Code: Since October 30, 2024 Started 3 days ago
2 conditions failed
4 Issues
0.0% Coverage
is less than 80.0%
Fix issues before they fail your Quality Gate with SonarLint in your IDE. Power up with
New issues FAILED
4
Required = 0
Accepted issues
0
Valid issues that were not fixed
Coverage
FAILED
0.0%
Duplications
0.0%


Replace this use of System.out by a logger.
Adaptability
Maintainability
bad-practice
cert
+
Open
Not assigned
L15 • 10min effort • 1 day ago • Code Smell • Major
Remove this useless assignment to local variable "result".
Intentionality
Maintainability
cwe
cert
...
+
Open
Not assigned
L19 • 1min effort • 6 minutes ago • Code Smell • Major
Remove this unused "result" local variable.
Intentionality
Maintainability
unused
+
Open
Not assigned
L19 • 5min effort • 6 minutes ago • Code Smell • Minor
Software Quality
Severity ?
Type
Scope
Status
Security Category

Badanie pokrycia kodu - coverage

- Pokrycie kodu pomaga upewnić się, że kod jest dobrze przetestowany, co przyspiesza wykrywanie nowych błędów.
- Analiza pokrycia wskazuje nieprzetestowane obszary, które mogą być podatne na błędy przy zmianach.
- JaCoCo (Java Code Coverage) to narzędzie do analizy pokrycia kodu przez testy w projektach opartych na Javie.
- JaCoCo pozwala programistom skupić się na kluczowych ścieżkach, co usprawnia budowanie testów i zarządzanie czasem ich wykonania.
- JaCoCo wspiera utrzymanie standardów i efektywne zarządzanie testowaniem, szczególnie w dużych projektach.
- JaCoCo jest dostępne jako wtyczka do Maven i Gradle, co ułatwia jego integrację z popularnymi narzędziami do budowania projektów Java.

Jak korzystać z JaCoCo

- **Dodaj wtyczkę do projektu**: W pliku `pom.xml` dodaj wtyczkę JaCoCo, aby była uruchamiana w fazie testowej.
- **Uruchom testy z JaCoCo**: Po dodaniu wtyczki wystarczy wykonać komendę `mvn test`, aby wygenerować raport pokrycia kodu.
- **Analiza wyników**: Po zakończeniu testów JaCoCo wygeneruje raporty pokrycia kodu w formacie HTML. Wygenerowane pliki można znaleźć w katalogu `target/site/jacoco`.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.8.7</version>
            <executions>
                <execution>
                    <goals>
                        <goal>prepare-agent</goal>
                        <goal>report</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Co robi JaCoCo?

- JaCoCo generuje kompleksowe raporty, które dokładnie pokazują, w jakim stopniu kod aplikacji został objęty testami. Raporty te dostarczają szczegółowej analizy pokrycia na różnych poziomach:
- **Instrukcje (Linie Kodów)**

  Analiza pokrycia poszczególnych linii kodu, która wskazuje, ile instrukcji zostało przetestowanych. Stanowi najniższy poziom szczegółowości, identyfikując linie pominięte przez testy.
- **Ścieżki Decyzyjne (Gałęzie)**

  Pokrycie logiki warunkowej, które śledzi wszystkie możliwe ścieżki wykonania, np. w instrukcjach warunkowych if-else. Dzięki temu raporty pomagają dostrzec potencjalnie pominięte scenariusze.
- **Metody**

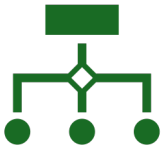
  Analiza pokazująca, czy wszystkie metody w kodzie zostały uruchomione podczas testów, co pozwala ocenić kompletność testów na poziomie poszczególnych funkcji.
- **Klasy**

  Ocena pokrycia klas w aplikacji. Ten poziom wskazuje, czy każda klasa została w pełni objęta testami, co pozwala kontrolować pokrycie na poziomie ogólnych jednostek kodu.

Co to są raporty JaCoCo?

Czym są raporty JaCoCo?

Raporty generowane przez JaCoCo pokazują, jaki procent kodu aplikacji został pokryty testami, analizując szczegółowo klasy, metody i linie kodu.



Jak działają?

JaCoCo monitoruje wykonanie testów jednostkowych i integracyjnych, generując raporty w różnych formatach (HTML, XML, CSV). Wyniki pozwalają szybko zidentyfikować nieprzetestowane części kodu. Można go zintegrować z narzędziami CI/CD, np. Jenkins, dla automatycznego raportowania.

Zalety

Raporty JaCoCo umożliwiają szczegółowe monitorowanie pokrycia kodu testami, pomagając zespołowi identyfikować nieprzetestowane fragmenty kodu i zwiększając ogólną stabilność oraz jakość aplikacji.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.jacoco.core		97%		91%	143	1,537	125	3,631	19	746	2	147
org.jacoco.examples		58%		64%	24	53	97	193	19	38	6	12
org.jacoco.agent.rt		75%		83%	32	130	75	344	21	80	7	22
jacoco-maven-plugin		90%		82%	35	193	49	465	8	116	1	23
org.jacoco.cli		97%		100%	4	109	10	275	4	74	0	20
org.jacoco.report		99%		99%	4	572	2	1,345	1	371	0	64
org.jacoco.ant		98%		99%	4	162	8	428	3	110	0	19
org.jacoco.agent		86%		75%	2	10	3	27	0	6	0	1
Total	1,438 of 28,925	95%	183 of 2,386	92%	248	2,766	369	6,708	75	1,541	16	308

Raporty

- **Tabela pokrycia kodu**: Zawiera wskaźniki procentowe dla instrukcji, gałęzi, metod, klas.
- **Raport szczegółowy dla klas**: Pozwala sprawdzić dokładne linie kodu pokryte i niepokryte przez testy. Linie pokryte mogą być zaznaczone na zielono, a niepokryte na czerwono.
- **Podsumowanie pokrycia projektu**: Procentowe zestawienie pokrycia dla całego projektu, które można używać do mierzenia postępów w testach regresyjnych.

Badanie pokrycia kodu testami

- Stworzenie projektu Maven
- Dodanie JUnita5 do pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.11.3</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Badanie pokrycia kodu testami

- Dodanie dowolnego testu, np.. w pliku MainTest.java i stworzenie test testGetText

```java
package pl.edu.agh.kis.pz1;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

import java.io.ByteArrayOutputStream;
import java.io.PrintStream;

public class MainTest {

    @Test
    public void testGetText() {
        // Test the getText method
        String expected = "Some text 2";
        assertEquals(expected, Main.getText(),
                message: "The getText method should return 'Some text 2'");
    }

}
```

```java
package pl.edu.agh.kis.pz1;

public class Main {
    public static void main(String[] args)
    {
        System.out.println("Hello World! " + getText());
    }

    public static String getText() { return "Some text 2"; }
}
```

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.8.12</version>
            <executions>
                <execution>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                </execution>
                <execution>
                    <id>report</id>
                    <phase>test</phase>
                    <goals>
                        <goal>report</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Badanie pokrycia kodu testami

- Stworzenie raportów code coverage poprzez jacoco
- Dodanie jacoco w pom.xml
- Po odpaleniu "mvn test" pojawi się

raport w pliku:

target/site/jacoco/index.html

Badanie pokrycia kodu testami

- Dodanie SonarQube do projektu w pliku pom.xml:

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.sonarsource.scanner.maven</groupId>
            <artifactId>sonar-maven-plugin</artifactId>
            <version>4.0.0.4121</version>
        </plugin>
```

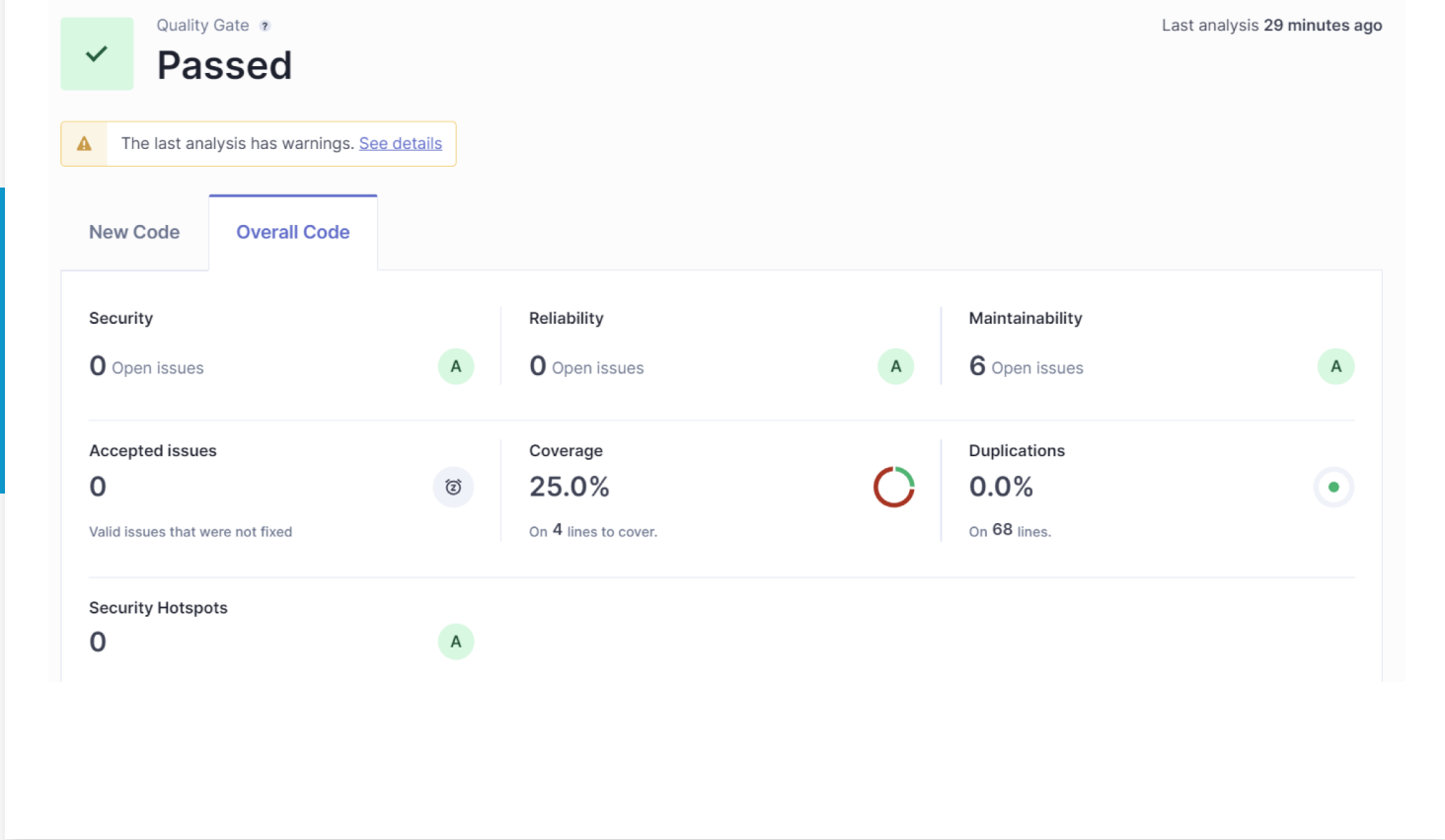
Stworzenie pliku na token np.. Token.txt. Przykładowa zawartość pliku: token.txt (token wygenerowany ze strony http://localhost:9000): `sqp_cb6eecdb88c9e94fe85437962972bc2c5ee7966`

Badanie pokrycia kodu testami

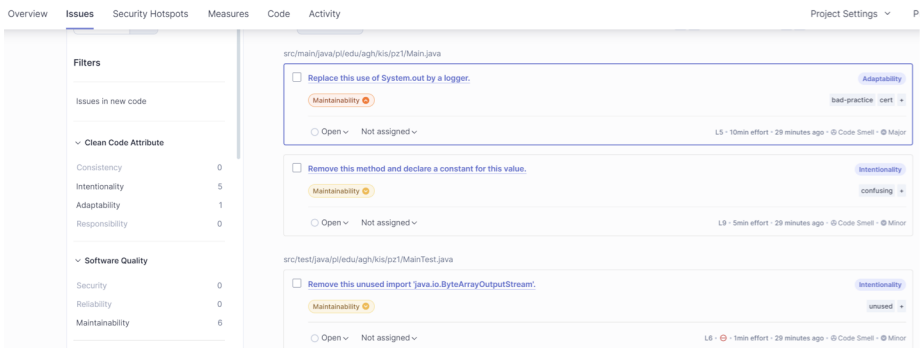
- Stworzenie pliku script.sh, który będzie budować projekt i odpali statyczną analizę i pokrycie kodu:

```
TOKEN="$(cat ./token.txt)"

mvn clean verify sonar:sonar \
  -Dsonar.projectKey=sonarqube-demo-1 \
  -Dsonar.projectName='SonarQube Demo 1' \
  -Dsonar.host.url=http://localhost:9000 \
  -Dsonar.token="$TOKEN"
```

Po odpaleniu skryptu za pomocą komendy ./script.sh w panelu SonarQube powinno pojawić się Issues oraz informacja o pokryciu kodu.
Jeśli po stworzeniu pliku script.sh brakuje uprawnień to używamy komendy chmod +x script.sh

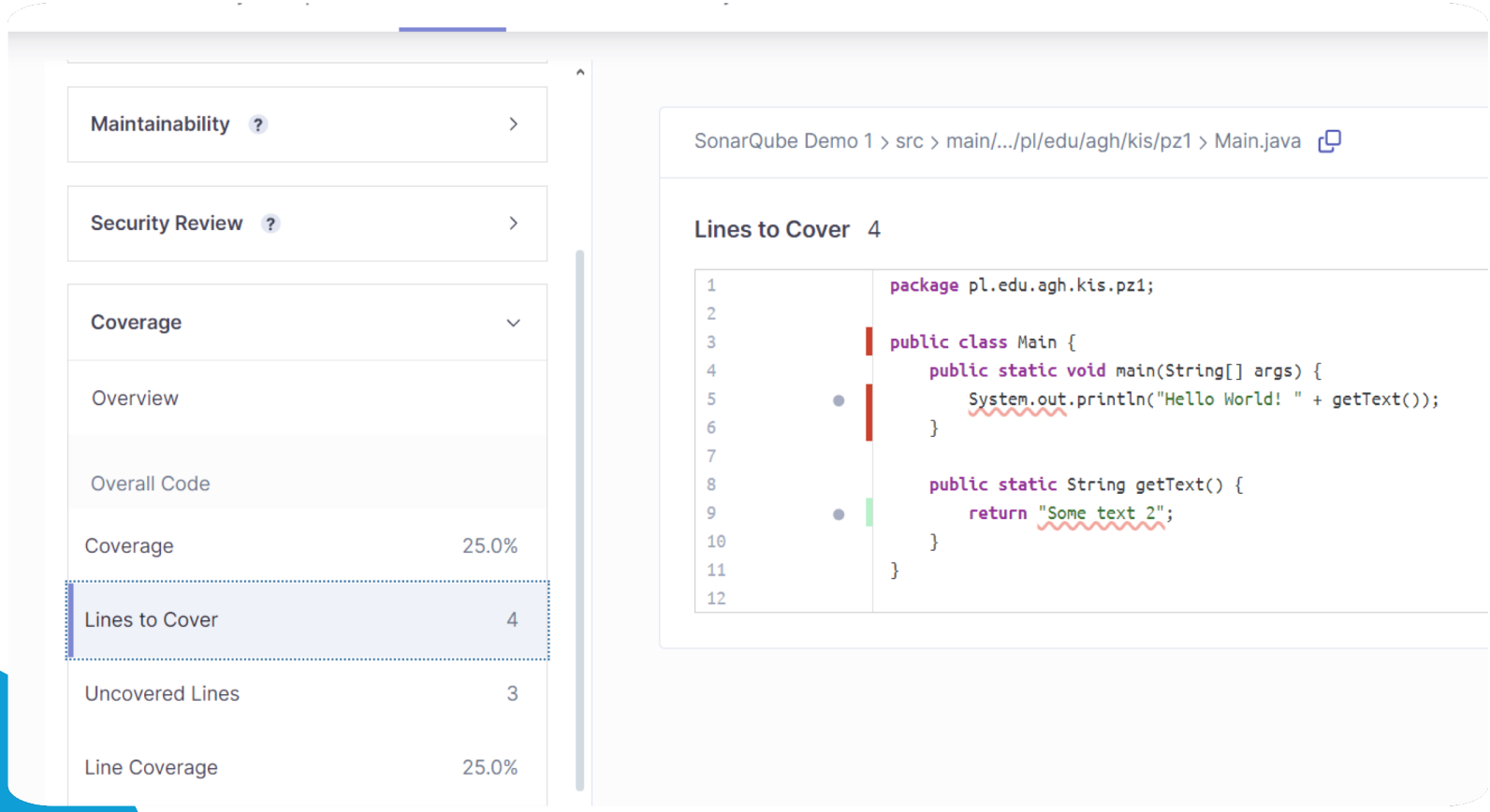


Badanie pokrycia kodu testami



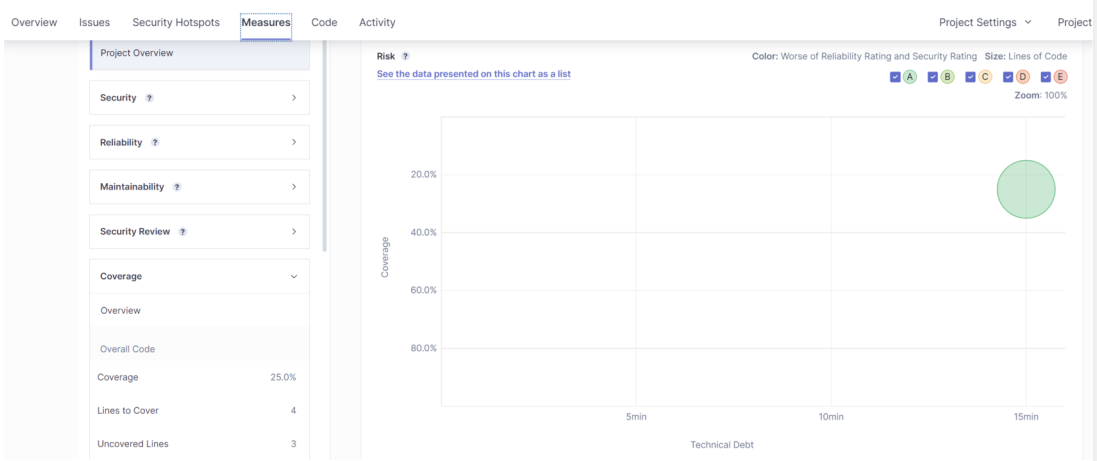
Badanie pokrycia kodu testami

Measures


Maintainability
Security Review
Coverage
Overview
Overall Code
Coverage 25.0%
Lines to Cover 4
Uncovered Lines 3
Line Coverage 25.0%
SonarQube Demo 1 > src > main/.../pl/edu/agh/kis/pz1 > Main.java
Lines to Cover 4
1 package pl.edu.agh.kis.pz1;
2
3 public class Main {
4 public static void main(String[] args) {
5 System.out.println("Hello World! " + getText());
6 }
7
8 public static String getText() {
9 return "Some text 2";
10 }
11 }
12

Wykres pokrycia kodu

- **Oś Y - Coverage (Pokrycie)**: Wskaźnik pokrycia kodu testami jednostkowymi, wyrażony w procentach.
Oś X - Czas (np. 5min, 10min, 15min): Przedstawia szacowany czas potrzebny do rozwiązania problemów z jakością kodu aby spełniał wymagania jakościowe.
- Każde kółko reprezentuje projekt lub część kodu, przy czym:
- **Kolor kółka - Ocena (Rating)**: Kolor zielony oznacza, że kod ma dobrą ocenę pod względem jakości i bezpieczeństwa
- **Rozmiar kółka - Lines of Code (Linie kodu)**: Wielkość kółka obrazuje ilość linii kodu.



Przydatne linki

Instalacja:

https://www.sonarsource.com/products/sonarqube/downloads/

https://docs.sonarsource.com/sonarqube/latest/analyzing-source-code/scanners/sonarscanner/

Dokumentacja:

https://docs.sonarsource.com/sonarqube/9.9/user-guide/issues/

Bibliografia

https://www.altkomsoftware.com/pl/blog/sonarqube-pierwsze-kroki/

https://docs.sonarsource.com/sonarqube/9.9/user-guide/issues/

https://www.eclemma.org/jacoco/

Dziękujemy za uwagę!