Univerisità Degli Studi di Salerno Hasna Arioua

Software dependability Report

Apache Commons Text Project

1. Introduction

The project chosen for the Software Dependability exam was Apache Commons Text. Apache Commons Text is a library within the Apache Commons project that provides utilities for working with text in Java. The library provides various functionalities to efficiently handle and manipulate textual data. After selecting the project to analyze, we have chosen a fork of the Apache Commons Text project found on GitHub.

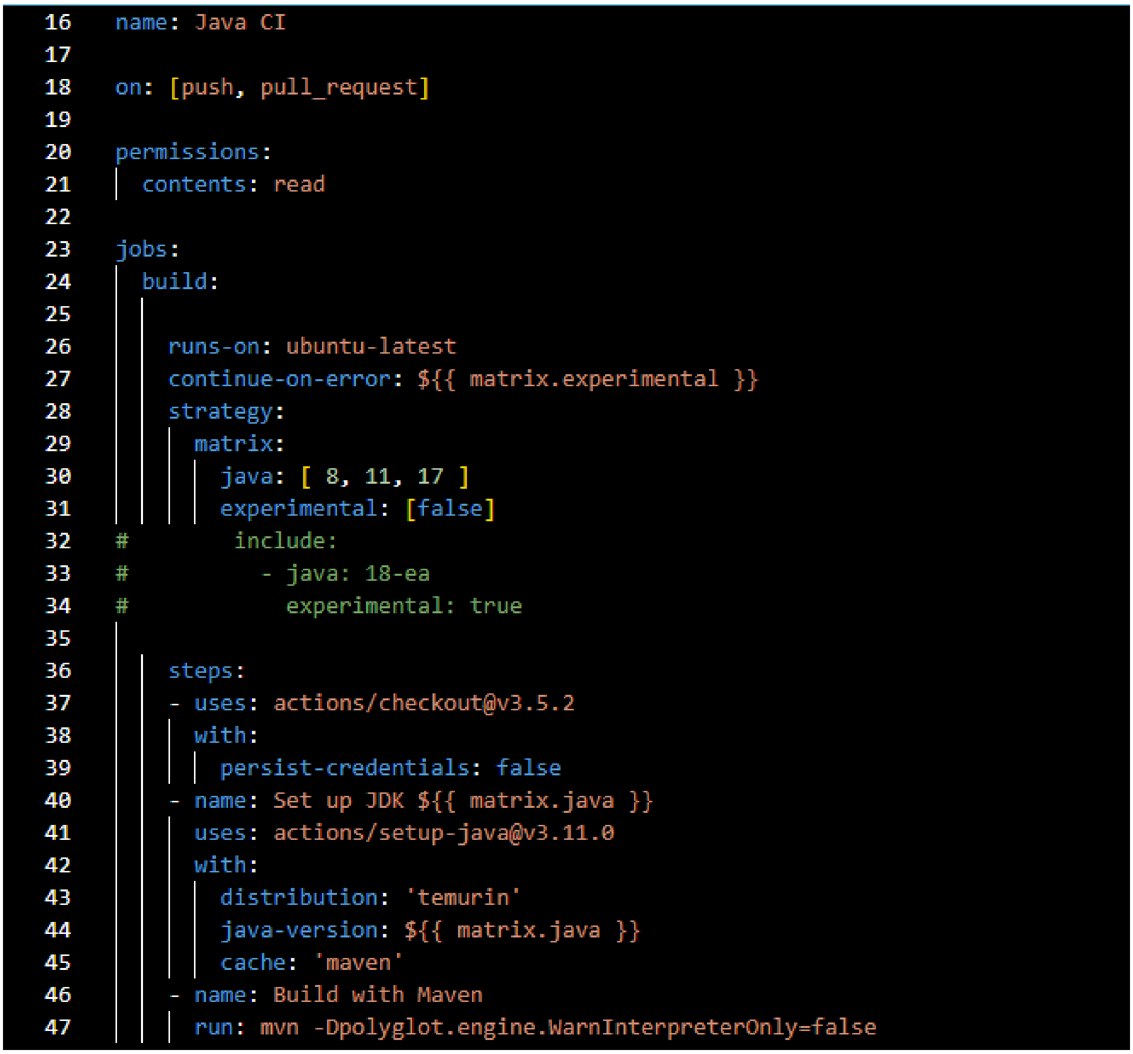
This report will provide a detailed explanation of the analysis that will be performed on the Commons Text project. The following chapters will discuss the various steps that will be taken in more detail.

1. The project can be built both in CI/CD and locally.
2. A thorough software quality analysis is conducted using SonarCloud. The issues found by Sonarcloud are categorized. The issues are fixed through refactoring, or a rationale for the skipped refactoring operations is provided.
3. A Docker image for the project is available on DockerHub and can be orchestrated. The Docker container can run as a web app or a Java application.
4. Code coverage analyzed using Cobertura or Jacoco, and a mutation.
5. testing campaign is conducted to analyze the project's test cases using PiTest.
6. used ecocode to analyze the energy greediness of the project and found the issues found by Ecocode and fixe the issues through refactoring, or a rationale.
7. implement Performance tests using JMH to stress test the most resource intensive components of the project.
8. EvoSuite and Randoop used to generate tests that cover code components that have not been adequately tested.
9. The security of the project will analyse using OWASP FindSecBugs, OWASP Dependency-Check, or OWASP ZAP.

The first thing that was done after forking the project was to clone the project on our personal pc and check, with the use of the visual studio if the project was buildable locally.

2- Built the project locally and using Continuous Integration.

After i confirmed that the project could be built locally on my PC, the next step was to implement a Github Action to automate the process of building, testing and deploying.the project. Being an open-source project from GitHub.

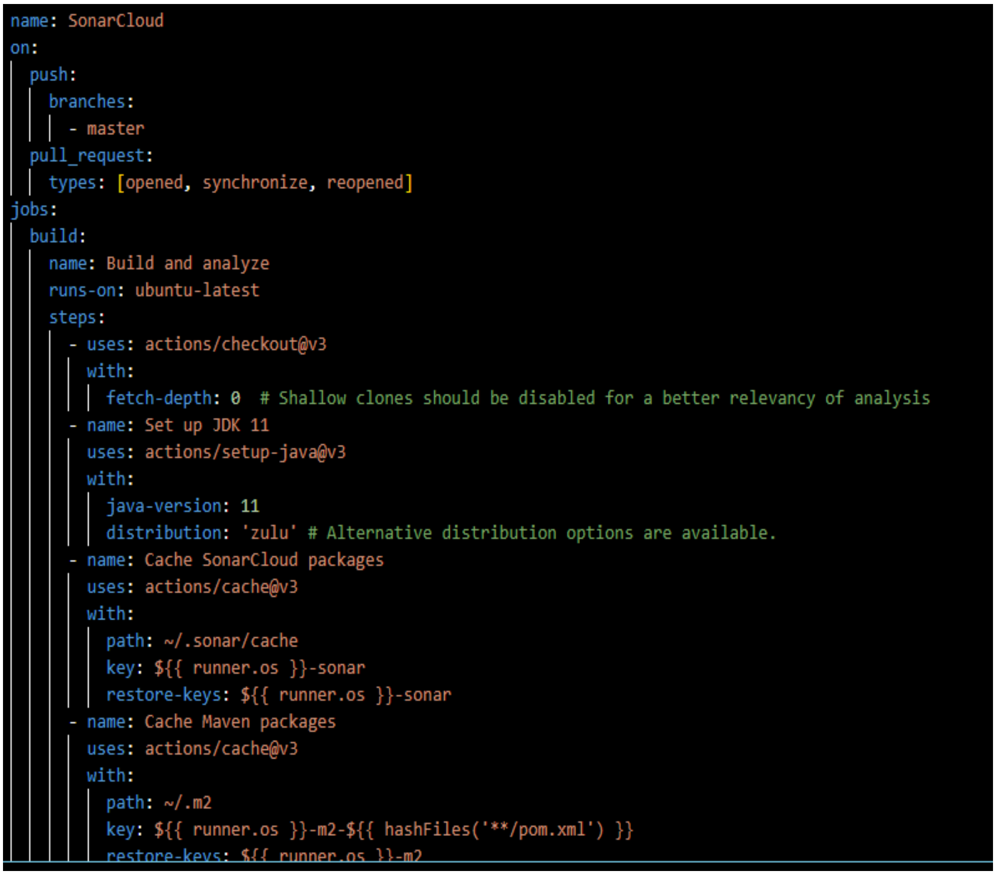


```yaml
name: Java CI

on: [push, pull_request]

permissions:
  contents: read

jobs:
  build:

    runs-on: ubuntu-latest
    continue-on-error: ${{ matrix.experimental }}
    strategy:
      matrix:
        java: [ 8, 11, 17 ]
        experimental: [false]
#        include:
#          - java: 18-ea
#            experimental: true

    steps:
    - uses: actions/checkout@v3.5.2
      with:
        persist-credentials: false
    - name: Set up JDK ${{ matrix.java }}
      uses: actions/setup-java@v3.11.0
      with:
        distribution: 'temurin'
        java-version: ${{ matrix.java }}
        cache: 'maven'
    - name: Build with Maven
      run: mvn -Dpolyglot.engine.WarnInterpreterOnly=false
```

Figure 1. GitHub Action Build Maven

2. SonarCloud Analysis

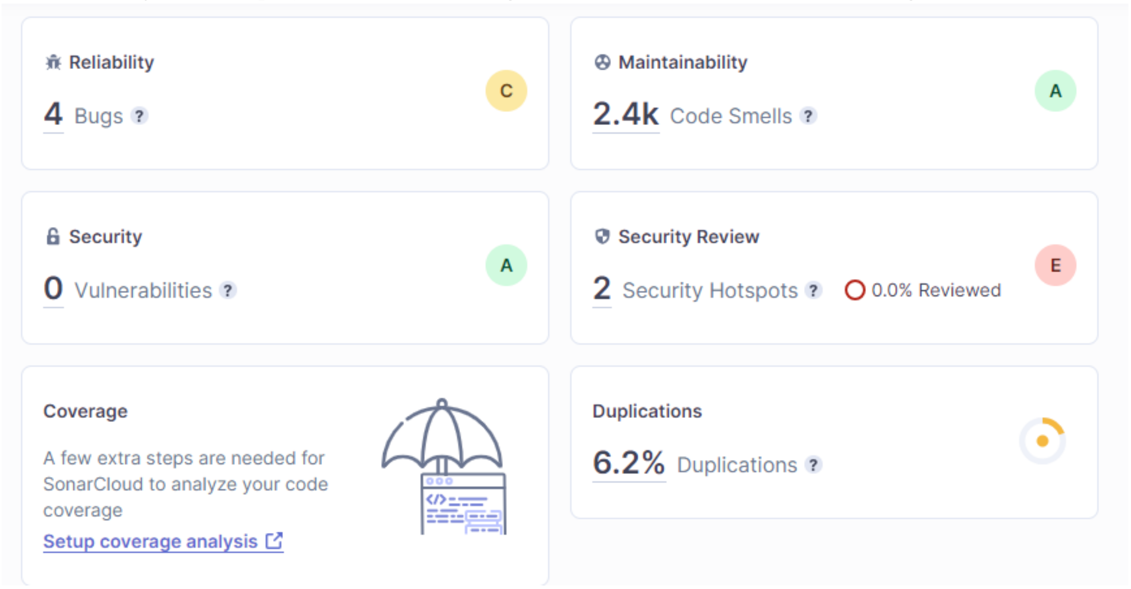
SonarCloud is a cloud-based platform offered by SonarSource to continuously perform quality assurance on code. This extension of the open-source SonarQube platform is hosted in the cloud, Facilitating the analysis and monitoring of codebase issues and vulnerabilities by making it con venient for teams.



```yaml
name: SonarCloud
on:
  push:
    branches:
      - master
  pull_request:
    types: [opened, synchronize, reopened]
jobs:
  build:
    name: Build and analyze
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
        with:
          fetch-depth: 0  # Shallow clones should be disabled for a better relevancy of analysis
      - name: Set up JDK 11
        uses: actions/setup-java@v3
        with:
          java-version: 11
          distribution: 'zulu' # Alternative distribution options are available.
      - name: Cache SonarCloud packages
        uses: actions/cache@v3
        with:
          path: ~/.sonar/cache
          key: ${{ runner.os }}-sonar
          restore-keys: ${{ runner.os }}-sonar
      - name: Cache Maven packages
        uses: actions/cache@v3
        with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2
```

Figure 2. GitHub Action SonarCloud

The first analysis of the project carried out using SonarCloud resulted in the following outcomes:



Reliability
4 Bugs ?
C

Maintainability
2.4k Code Smells ?
A

Security
0 Vulnerabilities ?
A

Security Review
2 Security Hotspots ? 0.0% Reviewed
E

Coverage
A few extra steps are needed for SonarCloud to analyze your code coverage
Setup coverage analysis

Duplications
6.2% Duplications ?

Figure 3. First SonarCloud Analysis

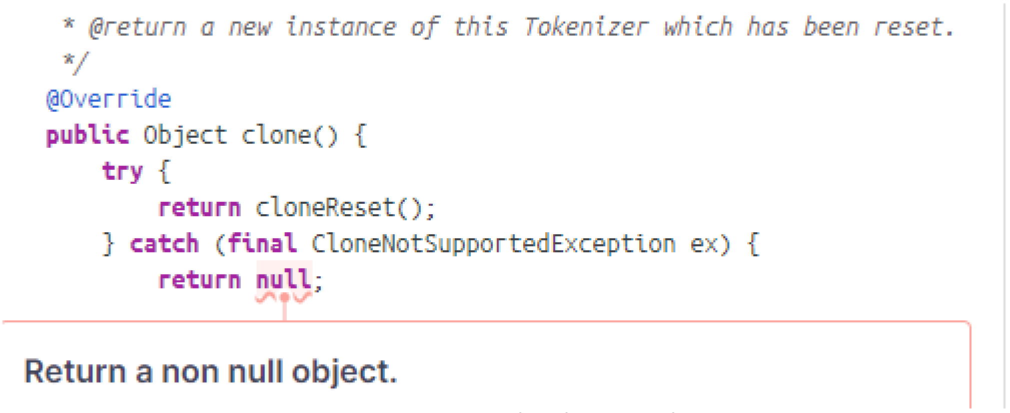
As we can see from the figure SonarCloud found:

- **4 Bugs** and classifying reliability with grade C
- **2.4k Code Smells** and classifying maintainability with grade A
- **0 Vulnerabilities** and classifying security with grade A
- **2 Security Hot-spots** and classifying the security review with grade E

2.1 Bugs :

SonarCloud has found 4 bugs classifying the Reliability ad a grade C. Reliability in the context of software dependability refers to the ability of a software system to consistently perform its intended functions without failure or errors under specified conditions and within a given period. It is a critical aspect of software quality assurance, and it contributes to the overall dependability of a system
The bugs classified by SonarCloud into categories:

- 2 major bugs
- 2 minor bugs
- In the classes StrTokenizer and StringTokenizer, the bug in the StrTok- enizer class was rectified by removing the whole class, which had the @Deprecated tag. The figure below shows the bug in the StringTokenizer class.

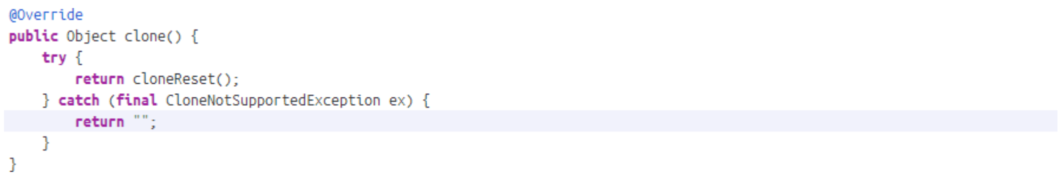


```
 * @return a new instance of this Tokenizer which has been reset.
 */
@Override
public Object clone() {
    try {
        return cloneReset();
    } catch (final CloneNotSupportedException ex) {
        return null;
```

Return a non null object.

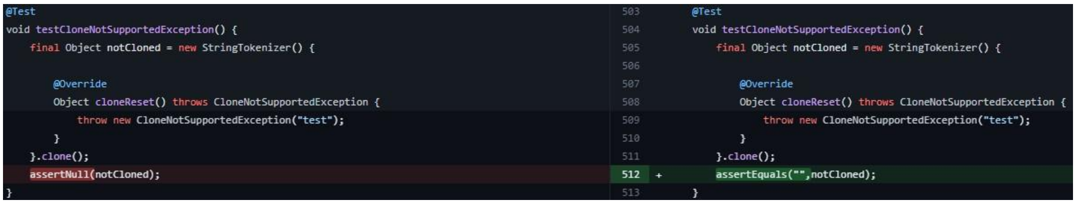
Figure 4. Major bug class StringTokenizer

This bug Calling clone () on an object should always return a string or an object. Returning null instead contravenes the method’s implicit contract, and for this reason the clone () method has been modified to return an empty string instead of null. Below is shown the figure with the class change:



```
@Override
public Object clone() {
    try {
        return cloneReset();
    } catch (final CloneNotSupportedException ex) {
        return "";
    }
}
```

Figure 5. Refactor method clone ()

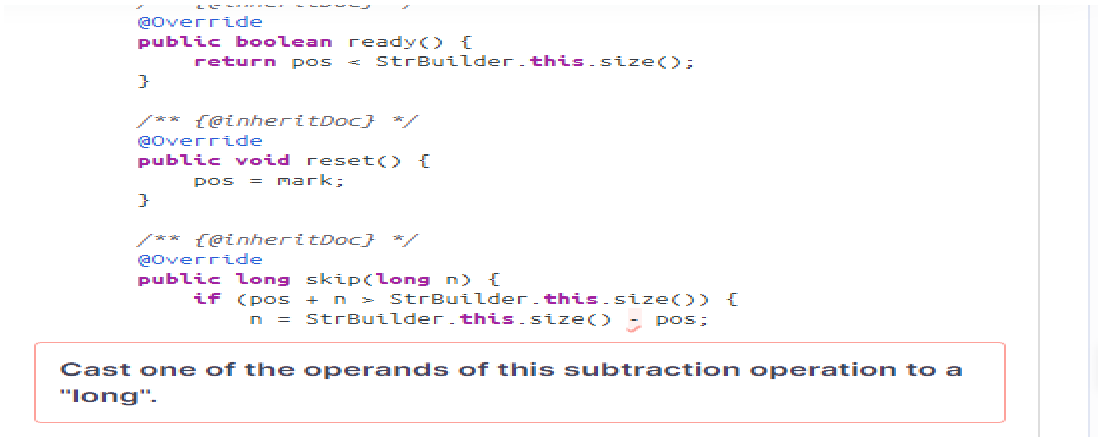


```java
@Test
void testCloneNotSupportedException() {
    final Object notCloned = new StringTokenizer() {

        @Override
        Object cloneReset() throws CloneNotSupportedException {
            throw new CloneNotSupportedException("test");
        }
    }.clone();
    assertNull(notCloned);
}
```

```java
@Test
void testCloneNotSupportedException() {
    final Object notCloned = new StringTokenizer() {

        @Override
        Object cloneReset() throws CloneNotSupportedException {
            throw new CloneNotSupportedException("test");
        }
    }.clone();
+   assertEquals("",notCloned);
}
```

Figure 6. Refactor Test Class

By removing the @Deprecated StrBuilder class and then adding a cast (long) to the variable 'pos', the second bug identified by SonarCloud as 'Cast one of the operands of this subtraction operation to a long' has been fixed.



```java
@Override
public boolean ready() {
    return pos < StrBuilder.this.size();
}

/** {@inheritDoc} */
@Override
public void reset() {
    pos = mark;
}

/** {@inheritDoc} */
@Override
public long skip(long n) {
    if (pos + n > StrBuilder.this.size()) {
        n = StrBuilder.this.size() - pos;
```

Cast one of the operands of this subtraction operation to a "long".

"

Figure 7. **Fix Bug "Cast one of the operands of this subtraction operation to a long**

2.2 *Code Smells*

Code smells are patterns or characteristics in the source code that might indicate a deeper problem. Identifying and addressing these smells can lead to improved code maintainability, readability, and overall software quality.in our case SonarCloud found 2.4k code smells , Here are some common categories of code smells:

1. Duplication:
Duplicated Code:
Repeated blocks of code that perform the same or similar functionality.
2. Complexity:
Long Method:Methods that are excessively long and perform multiple tasks.
Large Class:Classes that have grown too large, containing too many fields, methods, or responsibilities.
Switch Statements:The use of switch or case statements that can be indicative of a lack of polymorphism or abstraction.
Cyclomatic Complexity: High complexity in a method due to a large number of decision points.
3. Encapsulation:
Feature Envy: A method that seems more interested in the data of another class than in its own class.
Inappropriate Intimacy:Classes that are too closely connected or dependent on each other.
4. Data Organization:
Data Clumps: A set of parameters that often appear together, suggesting they should be grouped into a separate class.

Primitive Obsession: Excessive use of primitive data types instead of creating small, domain-specific classes.
Global State: Reliance on global variables or shared state, making it difficult to track and manage the flow of data.

5. Maintenance:

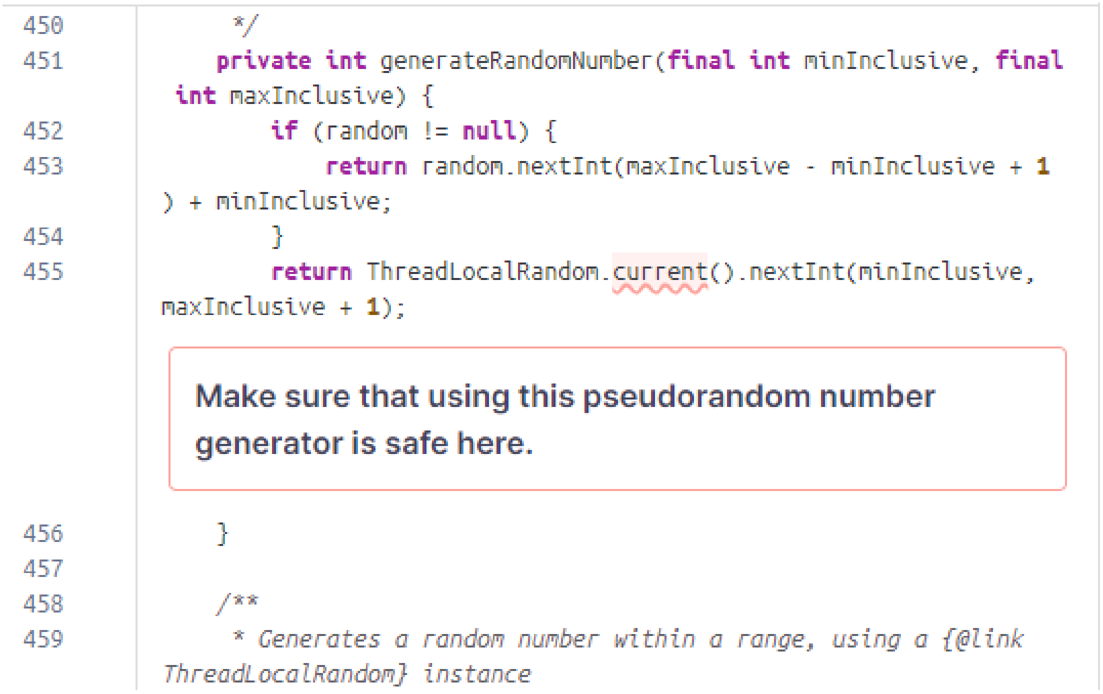
Shotgun Surgery:A single change that requires modifications to many parts of the codebase.

6. Readability:

Hard-Coded Values: Literal values that are hard-coded in the code, making it less flexible and harder to maintain.

2.3 Security Hot-spots

A "security hotspot" typically refers to a specific area or segment of a codebase that has been identified as potentially susceptible to security vulnerabilities or issues. These are areas that may require special attention and thorough review due to their critical nature regarding security. Security hotspots are often identified through static code analysis, manual code reviews, or automated security scanning tools.
SonarCloud found in the project 2 security hot-spots for the RandomStringGenerator class.



```
     */
    private int generateRandomNumber(final int minInclusive, final int maxInclusive) {
        if (random != null) {
            return random.nextInt(maxInclusive - minInclusive + 1) + minInclusive;
        }
        return ThreadLocalRandom.current().nextInt(minInclusive, maxInclusive + 1);
```

Make sure that using this pseudorandom number generator is safe here.

```
    }

    /**
     * Generates a random number within a range, using a {@link ThreadLocalRandom} instance
```

Figure 9. Security-Hotspot fix.

Declaring SecureRandom in both methods led to a code smell, which was later fixed by removing the SecureRandom declaration from the methods and inserting it as a global variable.

2.4 Final result in SonarCloud

After all the changes made in this step, we are now in this situation:

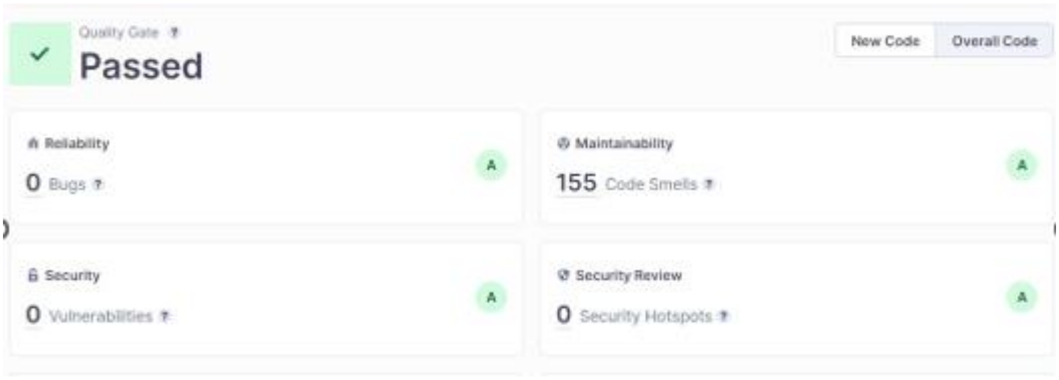
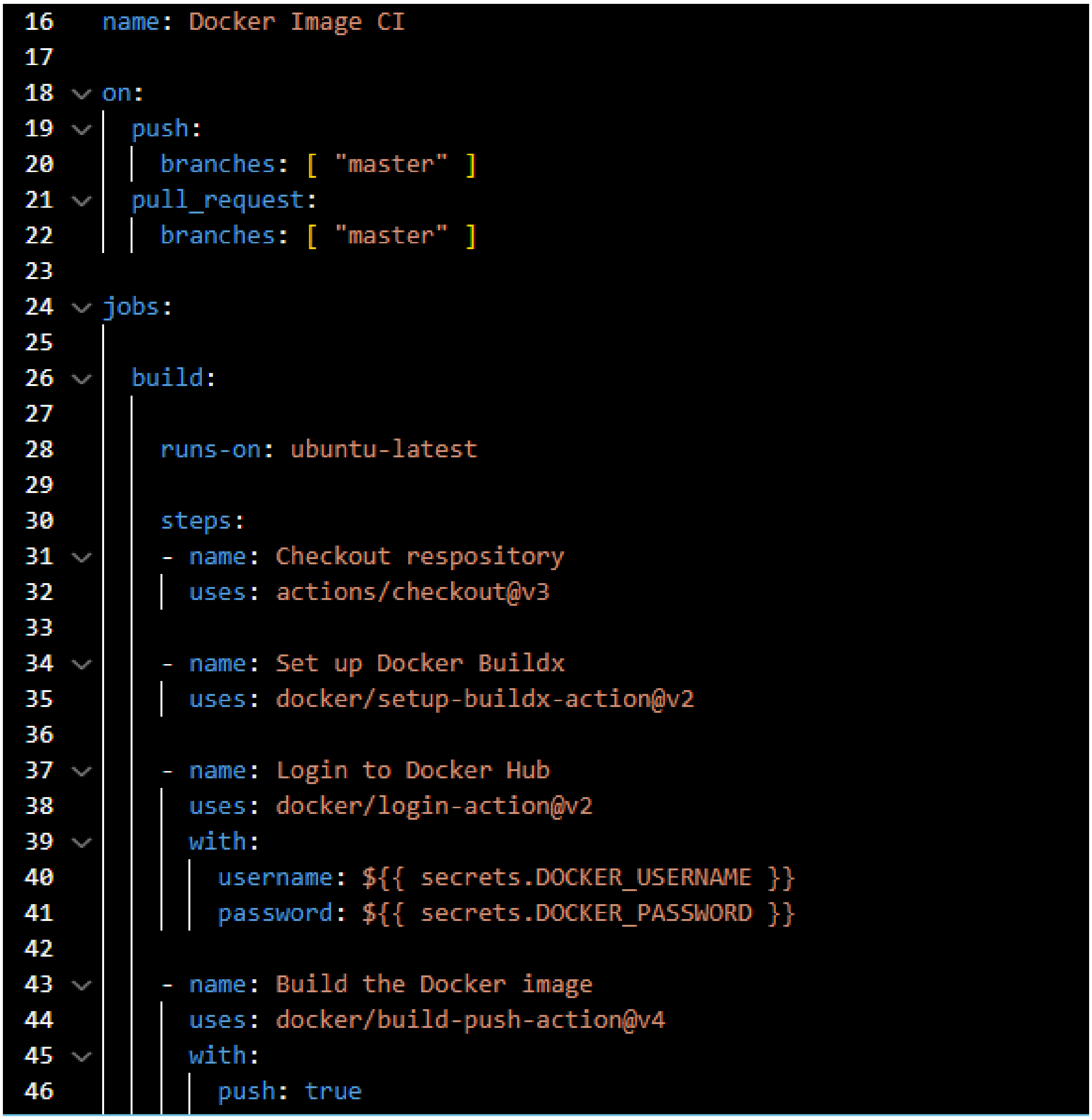


Figure 10. SonarCloud Analysis after Fixing bugs.

3. Docker image and containerization

The project was containerized using the open-source platform Docker. The process of using Docker begins by creating a file known as 'Dockerfile' that defines the step-by-step process of creating a Docker image.

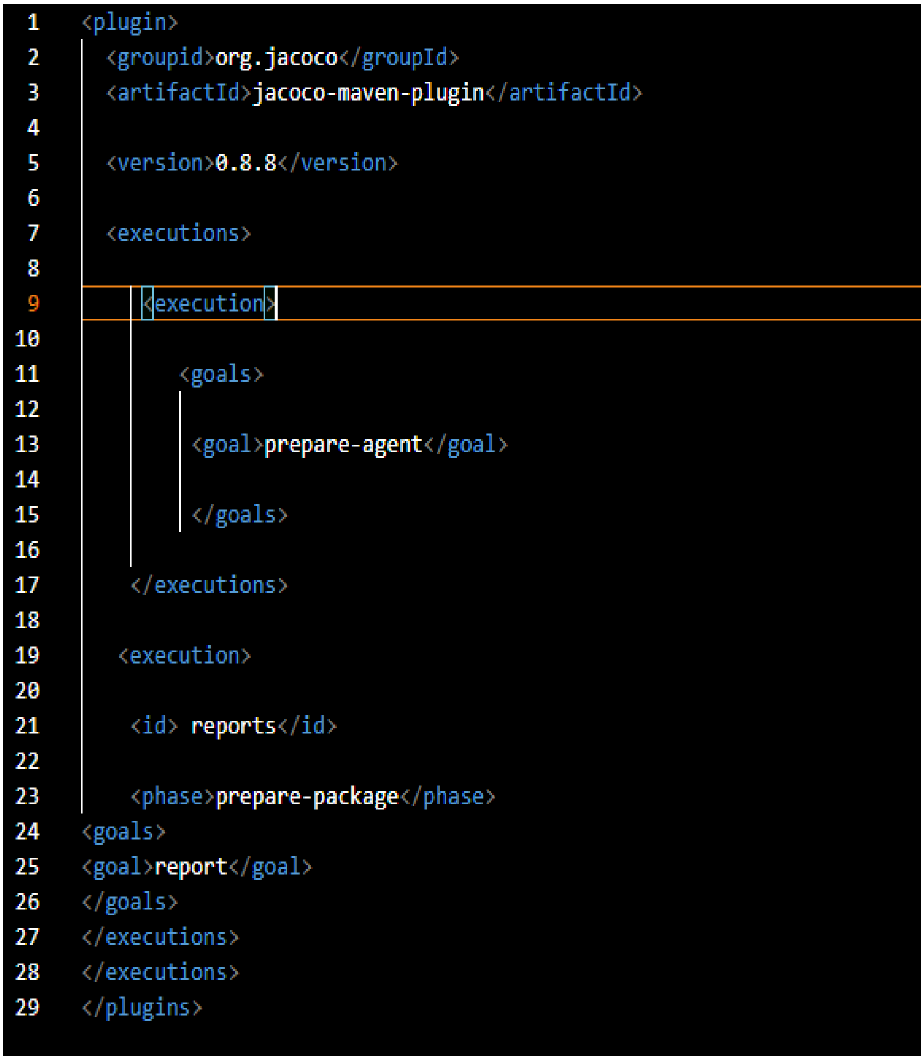


```yaml
name: Docker Image CI

on:
  push:
    branches: [ "master" ]
  pull_request:
    branches: [ "master" ]

jobs:

  build:

    runs-on: ubuntu-latest

    steps:
    - name: Checkout respository
      uses: actions/checkout@v3

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2

    - name: Login to Docker Hub
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}

    - name: Build the Docker image
      uses: docker/build-push-action@v4
      with:
        push: true
```

Figure 11. GitHub Action for the creation of the Docker Image.

4. Code Coverage:

When we talk about "Jacoco integration" it usually refers to integrating the JaCoCo (Java Code Coverage) tool into your Java project. JaCoCo is a popular code coverage tool for Java applications, providing insights into how much of your codebase is covered by tests.



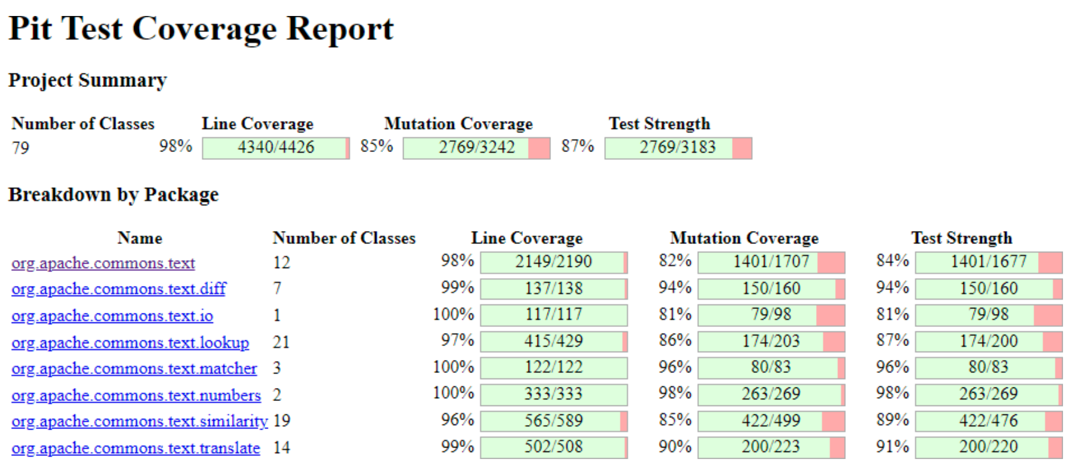
```
<plugin>
  <groupid>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>

  <version>0.8.8</version>

  <executions>

    <execution>

        <goals>

         <goal>prepare-agent</goal>

         </goals>

    </executions>

   <execution>

    <id> reports</id>

    <phase>prepare-package</phase>
<goals>
<goal>report</goal>
</goals>
</executions>
</executions>
</plugins>
```

Figure 12. JaCoCo integration.

After we added the plugin in the pom.xml, we ran the project build and obtained the JaCoCo report on coverage.

5. Mutation Testing using PITest

Mutation testing is a technique used to assess the effectiveness of a test suite by introducing small, intentional changes (mutations) to the source code and checking if the tests can detect these changes. The primary goal of mutation testing is to identify weaknesses in a test suite by creating mutant versions of the code and determining if the tests can detect and kill these mutants.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
79	98%	4340/4426	85%	2769/3242	87%	2769/3183

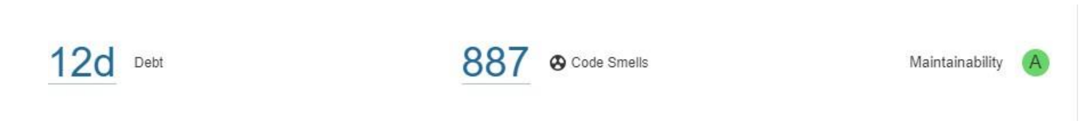
Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.apache.commons.text	12	98%	2149/2190	82%	1401/1707	84%	1401/1677
org.apache.commons.text.diff	7	99%	137/138	94%	150/160	94%	150/160
org.apache.commons.text.io	1	100%	117/117	81%	79/98	81%	79/98
org.apache.commons.text.lookup	21	97%	415/429	86%	174/203	87%	174/200
org.apache.commons.text.matcher	3	100%	122/122	96%	80/83	96%	80/83
org.apache.commons.text.numbers	2	100%	333/333	98%	263/269	98%	263/269
org.apache.commons.text.similarity	19	96%	565/589	85%	422/499	89%	422/476
org.apache.commons.text.translate	14	99%	502/508	90%	200/223	91%	200/220

Figure 13. PITest report.

6. SonarQube Analysis using EcoCode plugin:

SonarQube is an open-source platform designed for continuous inspection of code quality to perform static code analysis. It helps development teams identify and fix code quality issues, security vulnerabilities, and maintainability issues early in the software development process.



12d Debt

887 Code Smells

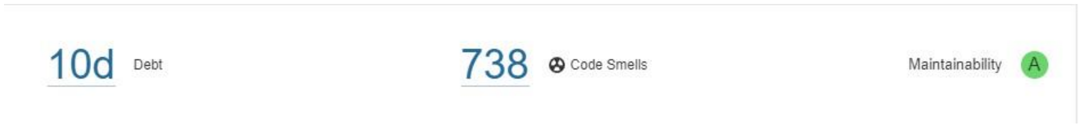
Maintainability A

Figure 14. SonaQube with Ecocode Analysis.

As we can see we can found 887 code smells and 12 debit also Maintainability class A

.

The final result after the refactoring we performed is shown in the following figure.



10d Debt

738 Code Smells

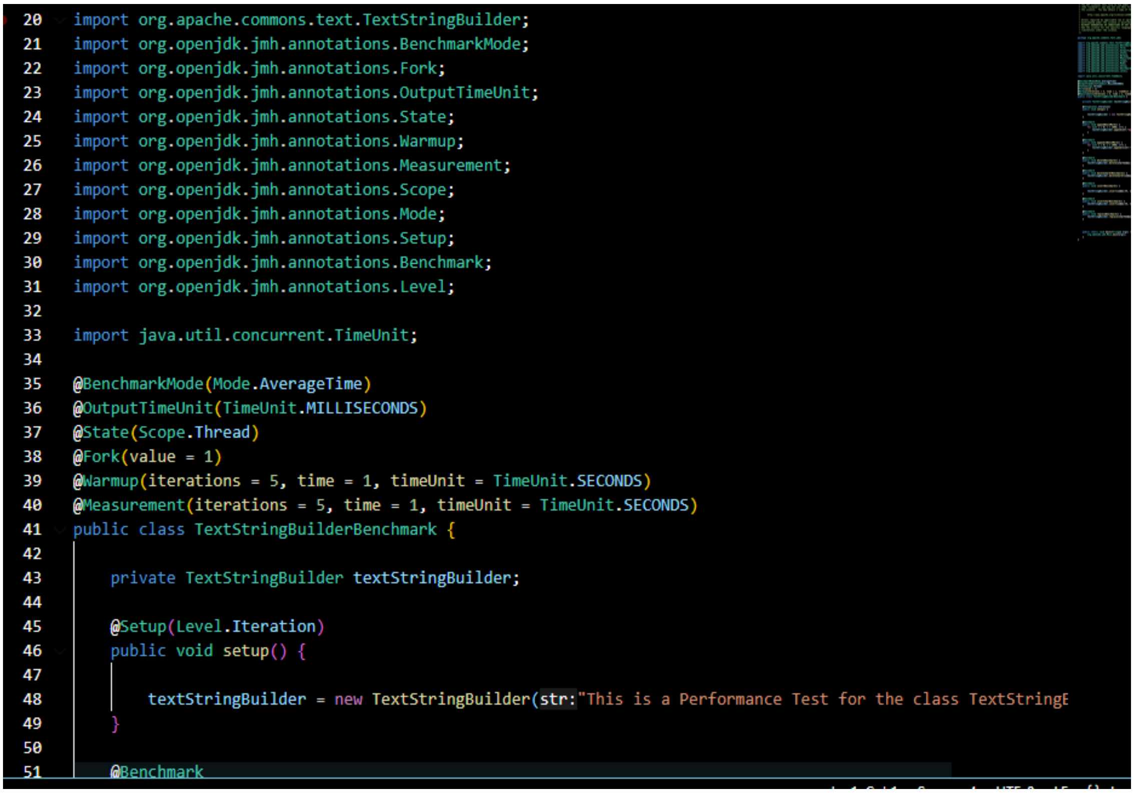
Maintainability A

Figure 15. SonaQube with Ecocode final analysis.

7. Performance Testing using Java Microbenchmark Harness:

A microbenchmark harness is a framework or set of tools that facilitates the creation and execution of microbenchmarks. A microbenchmark is a specific type of benchmark designed to measure the performance of a small and isolated piece of code, often at a very fine-grained level. Microbenchmarking is useful for assessing the performance of specific operations or algorithms in a controlled environment.

The term "harness" in the context of microbenchmarks refers to the infrastructure or framework that provides a standardized way to write, run, and analyze microbenchmarks. The class that we have chosen for the performance test is TextStringBuilder because it turned out to be the class with the greatest computational complexity.



```java
import org.apache.commons.text.TextStringBuilder;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Fork;
import org.openjdk.jmh.annotations.OutputTimeUnit;
import org.openjdk.jmh.annotations.State;
import org.openjdk.jmh.annotations.Warmup;
import org.openjdk.jmh.annotations.Measurement;
import org.openjdk.jmh.annotations.Scope;
import org.openjdk.jmh.annotations.Mode;
import org.openjdk.jmh.annotations.Setup;
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.Level;

import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Thread)
@Fork(value = 1)
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
public class TextStringBuilderBenchmark {

    private TextStringBuilder textStringBuilder;

    @Setup(Level.Iteration)
    public void setup() {

        textStringBuilder = new TextStringBuilder(str:"This is a Performance Test for the class TextStringE
    }

    @Benchmark
```

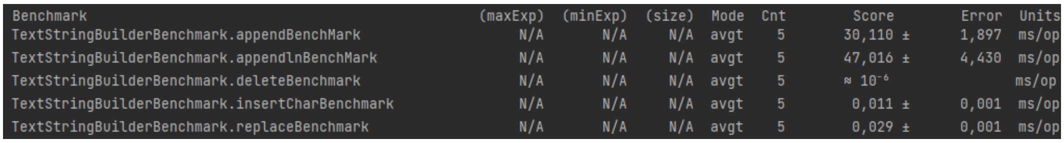
Figure 16. TextStringBuilderBenchmark implementation.

The test class was configured as follows:

@BenchmarkMode(Mode.AverageTime
@OutputTimeUnit(TimeUnit.MILLISECONDS
@State(Scope.Thread)
@Fork(value=1)
@Warmup
@Measurements

Afterward, a main was created to execute all the test class's methods and produce the results depicted in the figure.

Figure 17. JMH results.



```
Benchmark                                         (maxExp)  (minExp)  (size)  Mode  Cnt       Score       Error  Units
TextStringBuilderBenchmark.appendBenchMark             N/A       N/A     N/A  avgt    5      30,110 ±     1,897  ms/op
TextStringBuilderBenchmark.appendlnBenchMark           N/A       N/A     N/A  avgt    5      47,016 ±     4,430  ms/op
TextStringBuilderBenchmark.deleteBenchmark             N/A       N/A     N/A  avgt    5      ≈ 10⁻⁶              ms/op
TextStringBuilderBenchmark.insertCharBenchmark         N/A       N/A     N/A  avgt    5       0,011 ±     0,001  ms/op
TextStringBuilderBenchmark.replaceBenchmark            N/A       N/A     N/A  avgt    5       0,029 ±     0,001  ms/op
```

8. Test Cases gereration using Evo Suite

EvoSuite is a tool for automatic test case generation for Java programs. It is an evolutionary testing tool that uses a search-based approach to automatically generate unit tests with the goal of achieving high code coverage. EvoSuite can be integrated into your development environment, and it works with popular build

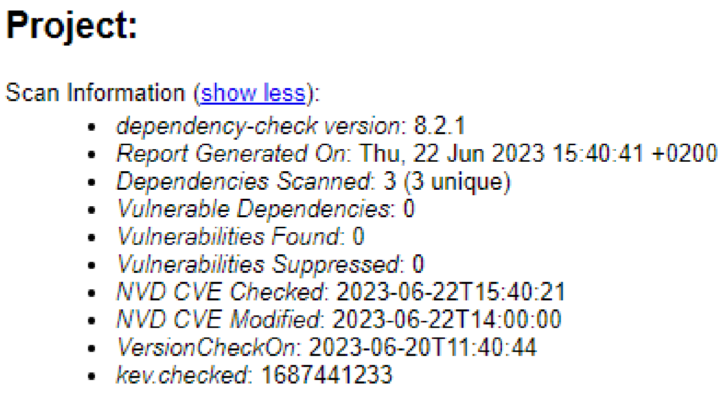
tools like Maven.
The tests created with Evo Suite are present in the evosuite-tests directory of commons-text.

9. FindSecBugs Analysis

FindSecBugs is a security-focused plugin for the FindBugs static analysis tool. It is designed to identify common security vulnerabilities in Java code. FindBugs is a widely used open-source static analysis tool that scans Java bytecode to find potential bugs, and FindSecBugs extends its capabilities to focus specifically on security-related issues .

10. OWASP DC Analysis

OWASP Dependency-Check is a tool designed to identify and alert about known vulnerabilities in project dependencies. It helps developers and organizations manage security risks associated with third-party libraries and components.A complete report can be found in **dependency-check-report.html** in the project's repository.



Project:

Scan Information (show less):

- *dependency-check version*: 8.2.1
- *Report Generated On*: Thu, 22 Jun 2023 15:40:41 +0200
- *Dependencies Scanned*: 3 (3 unique)
- *Vulnerable Dependencies*: 0
- *Vulnerabilities Found*: 0
- *Vulnerabilities Suppressed*: 0
- *NVD CVE Checked*: 2023-06-22T15:40:21
- *NVD CVE Modified*: 2023-06-22T14:00:00
- *VersionCheckOn*: 2023-06-20T11:40:44
- *kev.checked*: 1687441233

Figure 18. OWASP DC Analysis.

11. OWASP ZAP Analysis

OWASP ZAP (Zed Attack Proxy) is an open-source security testing tool designed for finding vulnerabilities in web applications. It is one of the tools maintained by the Open Web Application Security Project (OWASP). ZAP provides automated scanners and various tools for both manual and
Automated testing of web applications security

12. Conclusion

Ensuring software dependability is an ongoing process that involves careful design, thorough testing, continuous monitoring, and prompt response to issues. It is a critical aspect of delivering high-quality and reliable software to end-users.