Univerisità Degli Studi di Salerno (2023), 1, 1–16

SOFTWARE DEPENDABILITY REPORT

Dependability Analysis of Apache Commons Text

Gaetano De Filippo, Shahrzad Abedibarzi

1. Introduction

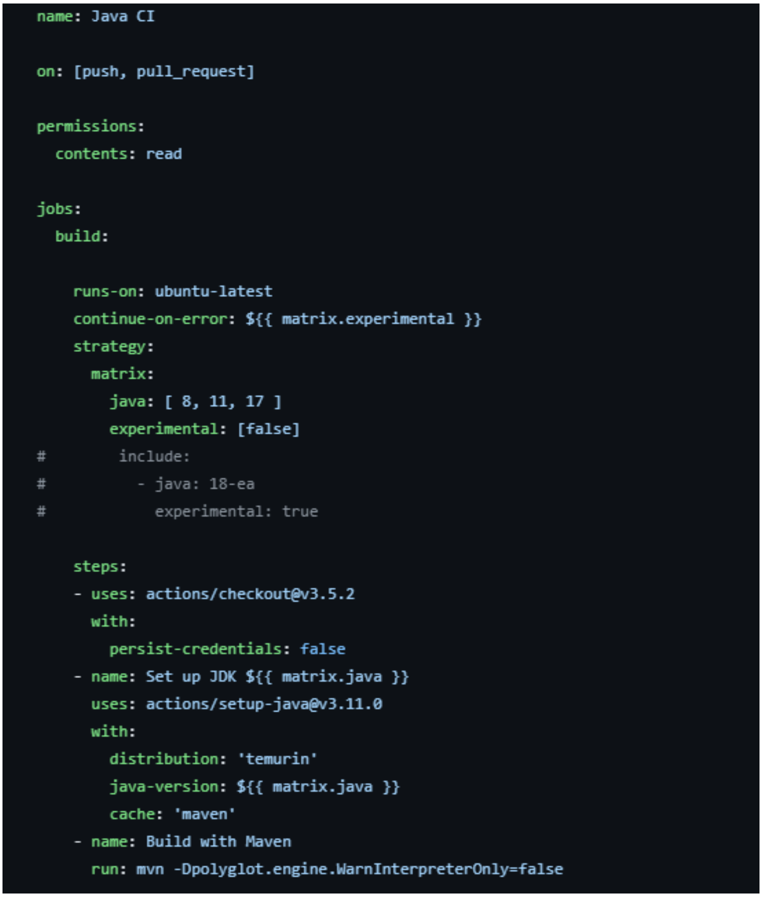
The project choice for the Software Dependability exam fell on Apache Commons Text. Apache Commons Text is an open-source library developed by the Apache Software Foundation designed to provide a set of utilities for text manipulation and processing in the Java language. The main goal of Commons Text is to simplify work with text by providing useful classes and methods for common operations such as formatting, processing, validation, and string manipulation.
After we have chosen the project to analyze, a fork of the project **apache/commons-text** found in GitHub was performed in our reference repository (**https://github.com/gadf00/commons-text**).
This report will explain in detail how and what analysis will be performed on the Commons Text project. Here are listed the various steps that will be performed and will be discussed in more detail in the following chapters:

1. A GitHub Action will be created and it will be in charge of automating code compilation and checking that project is built correctly;
2. An in-depth analysis of the project using SonarCloud will be performed. SonarCloud is a static code analysis platform that provides services for code quality control, detection of bugs, vulnerabilities and security issues, as well as measurements of code coverage and software maintenance metrics;
3. A Docker Image of the project will be created and used to create a container. In addition, a GitHub Action will be implemented that will allow a Docker Image to be created with each build of the project;
4. The code coverage of the entire project will be calculated using JaCoCo;
5. A mutation testing campaign using PITest will be conducted. Mutation testing is a testing technique that aims to assess the quality of test cases provided for the project;
6. SonarQube will be used with the addition of the EcoCode plugin to analyze the energy greediness of the project;
7. Performance Tests will be created using JMH (Java Microbenchmark Harness) to stress the most onerous components of the project;
8. EvoSuite will be used to generate tests to cover code components that have not been adequately tested;
9. The security of the project will be analyzed using the OWASP FindSecBugs, OWASP DC and OWASP ZAP tools.

2. Building the project locally and in Continuous Integration

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 IntelliJ IDE, if the project was buildable locally.

After we checked that the project was actually buildable locally on our PC, the next thing that was done was to implement a Github Action to automate the process of building, testing and deploying the project. Being a project taken from GitHub, it already had this type of GitHub Action already configured in Continuous Integration (CI) in order to be automatically executed when commits or pushes are made to the repository.
This allows us to immediately check if the code is correct and to identify any errors or compilation problems when we push new code on GitHub.

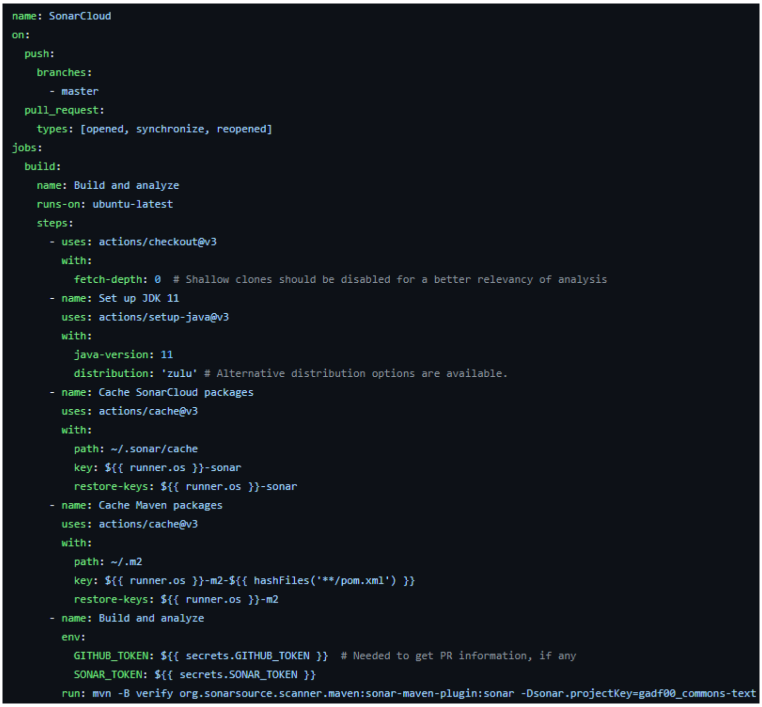


```yaml
name: Java CI

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

3. SonarCloud Analysis

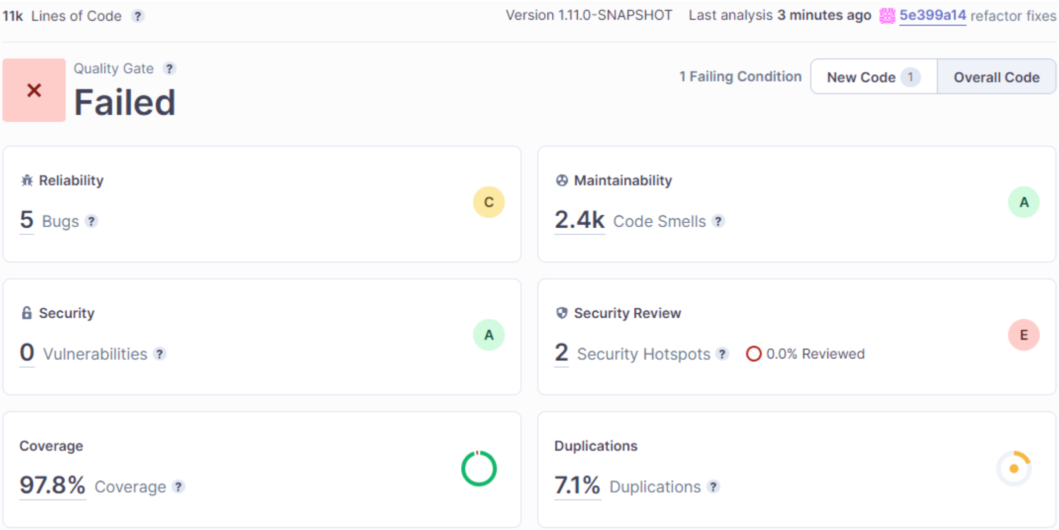
We used SonarCloud for a code quality analysis of the project.
SonarCloud is a cloud-based platform for static code analysis, software quality assessment. It performs a thorough analysis of the source code to identify potential problems, vulnerabilities, bugs and violations of programming best practices.
To integrate SonarCloud into the project, a workflow with GitHub Actions has been implemented that allows SonarCloud to perform a source code analysis every time a commit is pushed on GitHub.
The SonarCloud workflow implementation is shown in the figure below.



```
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
      - name: Build and analyze
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  # Needed to get PR information, if any
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=gadf00_commons-text
```

Figure 2. GitHub Action SonarCloud

The first analysis of the project performed with SonarCloud produced the following results:



11k Lines of Code ?

Version 1.11.0-SNAPSHOT Last analysis 3 minutes ago 5e399a14 refactor fixes

Quality Gate ?

✕ Failed

1 Failing Condition

Reliability
5 Bugs ? C

Maintainability
2.4k Code Smells ? A

Security
0 Vulnerabilities ? A

Security Review
2 Security Hotspots ? ○ 0.0% Reviewed E

Coverage
97.8% Coverage ?

Duplications
7.1% Duplications ?

Figure 3. First SonarCloud Analysis

As we can see from the figure shown above, SonarCloud analyzed about 11k lines of code and divided the issues into bugs, code smells, vulnerabilities and security hotspots. SonarCloud found:

- 5 **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

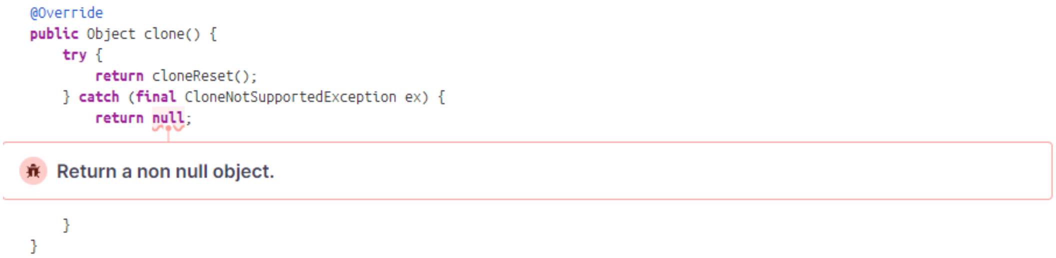
Subsequently, all the problems were examined individually and an attempt was made to solve them to increase the quality of the code.

3.1 Bugs

SonarCloud has found 5 bugs classifying the Reliability ad a grade C. Reliability is one of the fundamental dimensions of software quality and is often considered one of the most important characteristics for critical software systems. It refers to the ability of a software system to consistently and predictably perform its required functions over time, while maintaining an acceptable level of performance and without causing malfunctions or failures.
The bugs have been classified by SonarCloud into categories:

- 2 major bugs
- 3 minor bugs

The 2 majors are present in the classes StrTokenizer and StringTokenizer. The bug in the StrTokenizer class was fixed by removing the whole class because it had the @Deprecated tag. Instead the bug present in the StringTokenizer class is shown in the figure below:



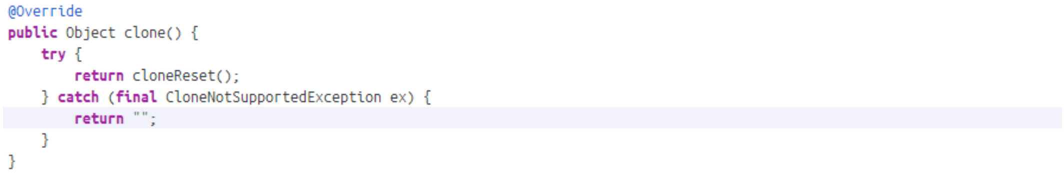
```
@Override
public Object clone() {
    try {
        return cloneReset();
    } catch (final CloneNotSupportedException ex) {
        return null;
```

Return a non null object.

```
    }
}
```

Figure 4. Major bug class StringTokenizer

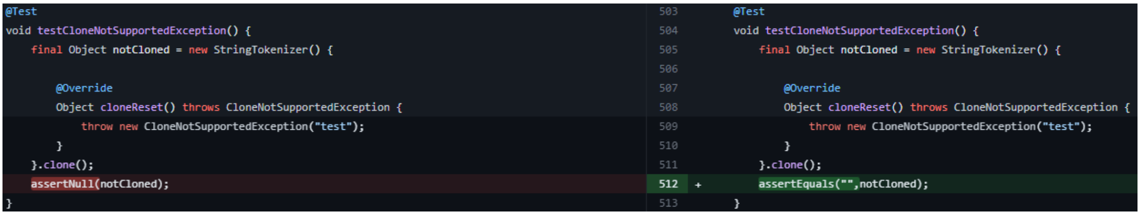
This is classified as a bug because "Calling clone() on an object should always return a string or an object. Returning null instead contravenes the method's implicit contract.", and for this reason the clone() method has been modified to return an empty string instead of null. Below is shown the figure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



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

Figure 5. Refactor method clone()

After modifying the method, in order not to receive an error during the build phase, the corresponding test method was modified:

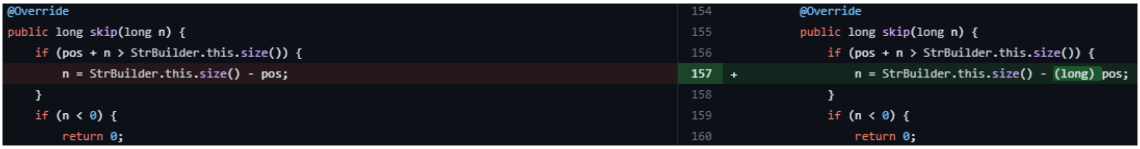


```
@Test
void testCloneNotSupportedException() {
    final Object notCloned = new StringTokenizer() {

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

```
    @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

The other 3 bugs have been classified as "Minor". A bug has been fixed by removing the @Deprecated StrBuilder class, the second bug tagged by SonarCloud as "Cast one of the operands of this subtraction operation to a long", has been fixed by adding a cast (long) to the variable "pos" as shown from the figure below:

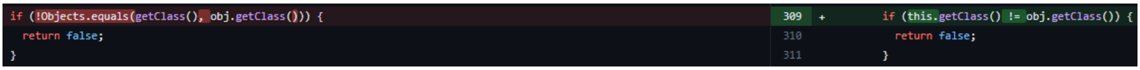


```
@Override
public long skip(long n) {
    if (pos + n > StrBuilder.this.size()) {
        n = StrBuilder.this.size() - pos;
    }
    if (n < 0) {
        return 0;
```

```
    @Override
    public long skip(long n) {
        if (pos + n > StrBuilder.this.size()) {
+           n = StrBuilder.this.size() - (long) pos;
        }
        if (n < 0) {
            return 0;
```

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

The last bug affecting the ExtendedMessageFormat class was tagged by SonarCloud with "Add a type test to this method.", it was fixed by removing the !Object.equals() method and replacing it with the "!=" operator as shown in the figure.



```
if (!Objects.equals(getClass(), obj.getClass())) {
  return false;
}
```

```
+   if (this.getClass() != obj.getClass()) {
      return false;
    }
```

Figure 8. Fix Bug "Cast one of the operands of this subtraction operation to a long"

3.2 Code Smells

Regarding code smells, SonarCloud found 2.4k code smells. In the table shown below, the code smells have been categorized before and after the refactoring.

Table 1. Code Smells categorization.

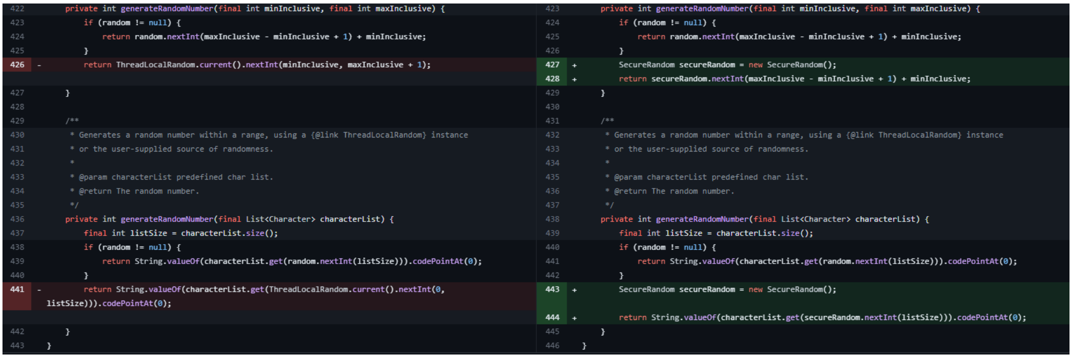
Category	Number of code smells before refactoring	Number of code smells after refactoring
Blocker Code Smells	12	1
Critical Code Smells	37	22
Major Code Smells	222	89
Minor Code Smells	967	12
Info Code Smells	1.2k	31

The most relevant code smells that we have not removed concerned the label assigned by SonarCloud "Refactor this method to reduce its Cognitive Complexity", "Refactor the code in order to not assign to this loop counter", "Refactor the code of the lambda to have only one invocation possibly throwing a runtime exception" and "Refactor this method to reduce the number of assertions".
We could not refactor the tag "Refactor this method to reduce its Cognitive Complexity" because it would cause some tests to fail during the build phase of the project.

We could not refactor the tag "Refactor the code in order to not assign to this loop counter", the tag "Refactor the code of the lambda to have only one invocation possibly throwing a runtime exception" and the tag "Refactor this method to reduce the number of assertions" because they would cause a build failure.

3.3 *Security Hot-spots*

A Security Hotspot is a specific term used by SonarCloud to indicate a potential vulnerability or security risk area within the source code of an application. SonarCloud identifies these areas using static code analysis and specific security rules. When SonarCloud detects a Security Hotspot, it doesn't treat it as a direct error or warning, but as a notice of a potential security issue that requires attention. A Security Hotspot indicates that a portion of code may have security risks or potential vulnerabilities, but it requires further assessment and analysis by developers.
SonarCloud found in the project 2 security hot-spots for the RandomStringGenerator class. The potential issue was tagged with "Make sure that using this pseudo-random number generator is safe here" which was fixed by using the SecureRandom class instead of the Random class. The figure below shows the refactoring of the two methods.



```
    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);

    }

    /**
     * Generates a random number within a range, using a {@link ThreadLocalRandom} instance
     * or the user-supplied source of randomness.
     *
     * @param characterList predefined char list.
     * @return The random number.
     */
    private int generateRandomNumber(final List<Character> characterList) {
        final int listSize = characterList.size();
        if (random != null) {
            return String.valueOf(characterList.get(random.nextInt(listSize))).codePointAt(0);
        }
-       return String.valueOf(characterList.get(ThreadLocalRandom.current().nextInt(0,
listSize))).codePointAt(0);

    }
}
```

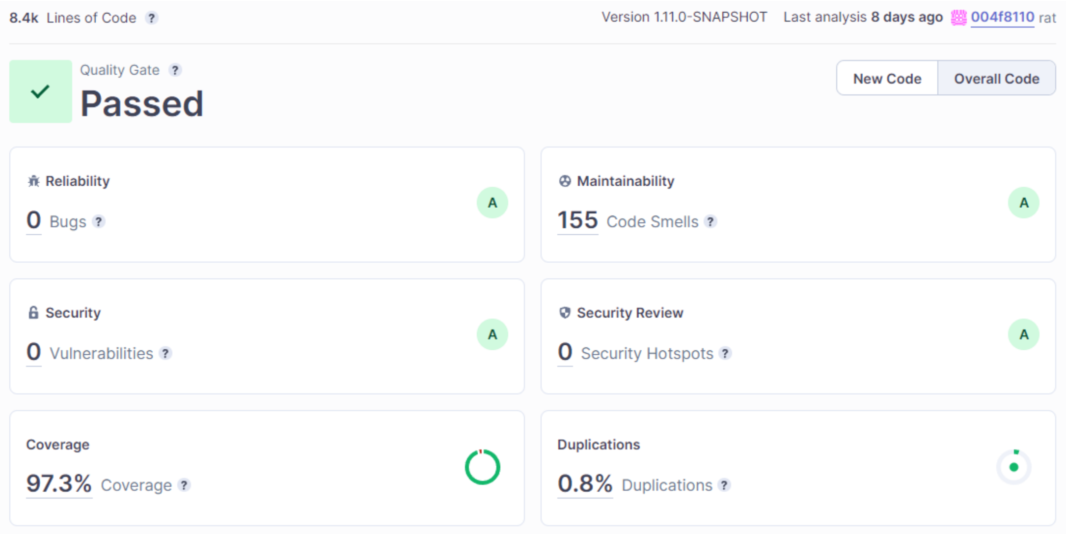
```
    private int generateRandomNumber(final int minInclusive, final int maxInclusive) {
        if (random != null) {
            return random.nextInt(maxInclusive - minInclusive + 1) + minInclusive;
        }
+       SecureRandom secureRandom = new SecureRandom();
+       return secureRandom.nextInt(maxInclusive - minInclusive + 1) + minInclusive;
    }

    /**
     * Generates a random number within a range, using a {@link ThreadLocalRandom} instance
     * or the user-supplied source of randomness.
     *
     * @param characterList predefined char list.
     * @return The random number.
     */
    private int generateRandomNumber(final List<Character> characterList) {
        final int listSize = characterList.size();
        if (random != null) {
            return String.valueOf(characterList.get(random.nextInt(listSize))).codePointAt(0);
        }
+       SecureRandom secureRandom = new SecureRandom();

+       return String.valueOf(characterList.get(secureRandom.nextInt(listSize))).codePointAt(0);
    }
}
```

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.

3.4 *Final result in SonarCloud*

After all the refactoring made in this step we got this situation:



8.4k Lines of Code ?

Version 1.11.0-SNAPSHOT Last analysis **8 days ago** 004f8110 rat

Quality Gate ?

✓ **Passed**

New Code | Overall Code

Reliability

0 Bugs ? A

Maintainability

155 Code Smells ? A

Security

0 Vulnerabilities ? A

Security Review

0 Security Hotspots ? A

Coverage

97.3% Coverage ?

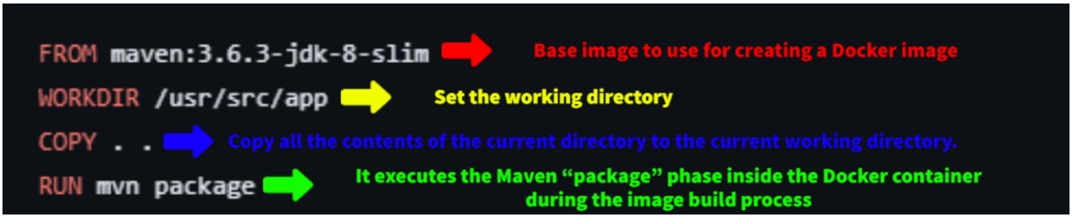
Duplications

0.8% Duplications ?

Figure 10. SonarCloud Analysis after refactoring.

4. Docker image and containerization

For the containerization of the project, the open-source platform Docker was used. The process of using Docker starts with defining a file called Dockerfile. A Dockerfile defines the step-by-step process of building a Docker image. The instructions in the Dockerfile specify the actions to take to configure the environment in which the application will run within the Docker container.



```
FROM maven:3.6.3-jdk-8-slim
WORKDIR /usr/src/app
COPY . .
RUN mvn package
```

Base image to use for creating a Docker image

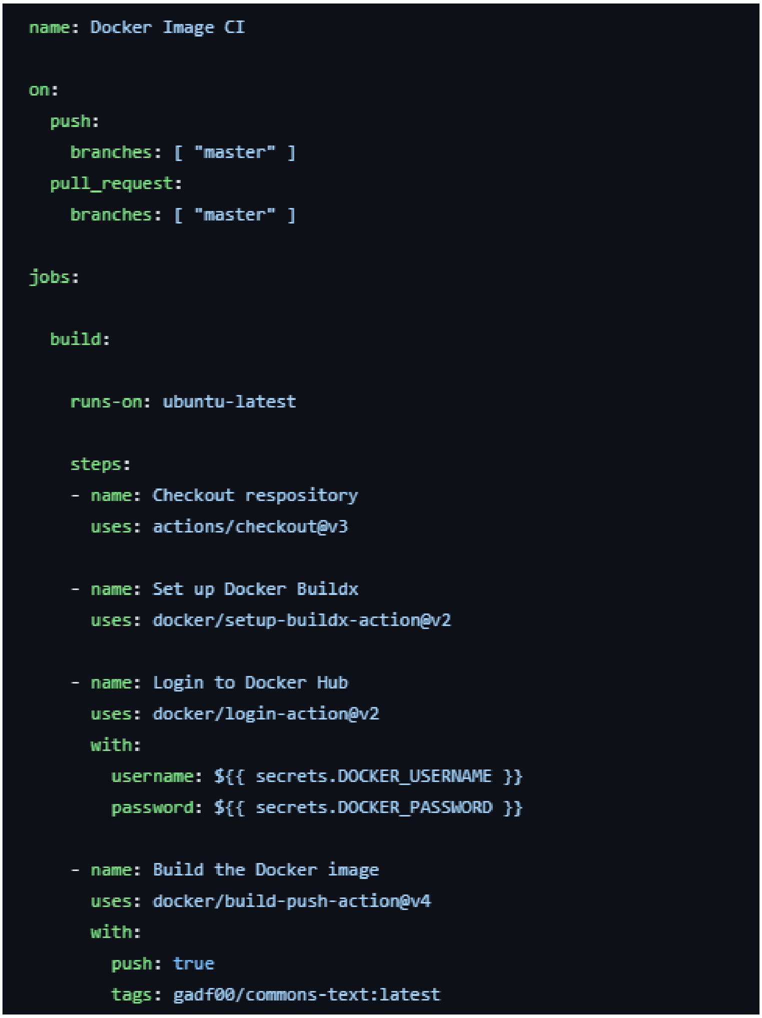
Set the working directory

Copy all the contents of the current directory to the current working directory.

It executes the Maven “package” phase inside the Docker container during the image build process

Figure 11. Dockerfile for Docker image

After that another workflow was created in GitHub Action so that, at each commit,it uses the previously created Dockerfile and it generates a new Docker image and uploads it on Docker Hub.

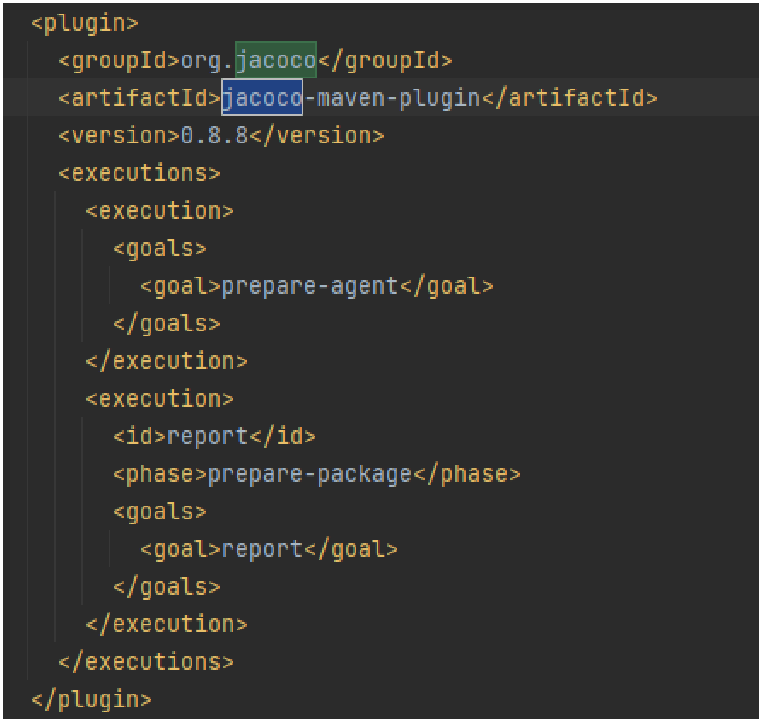


```
name: Docker Image CI

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
        tags: gadf00/commons-text:latest
```

Figure 12. GitHub Action for the creation of the Docker Image.

5. Code Coverage

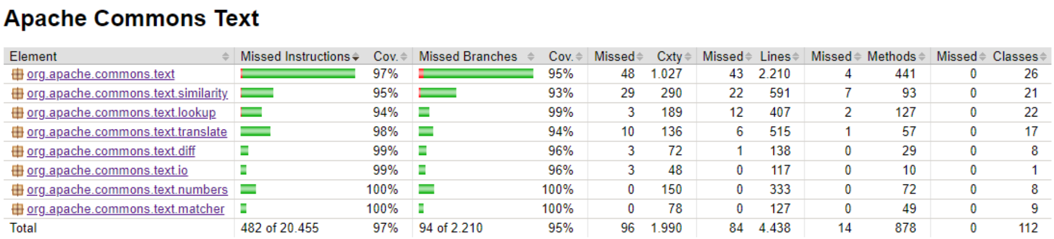
The JaCoCo plugin has been integrated into the project which allowed us to calculate the coverage of the code based on the tests performed. Thanks to the integration of this plugin, with each build,the tests of the project are automatically launched and the respective code coverage is thus calculated. In the pom.xml we added the following plugin to integrate JaCoCo in the project.



```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.8</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Figure 13. JaCoCo's integration.

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



Apache Commons Text

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.text		97%		95%	48	1.027	43	2.210	4	441	0	26
org.apache.commons.text.similarity		95%		93%	29	290	22	591	7	93	0	21
org.apache.commons.text.lookup		94%		99%	3	189	12	407	2	127	0	22
org.apache.commons.text.translate		98%		94%	10	136	6	515	1	57	0	17
org.apache.commons.text.diff		99%		96%	3	72	1	138	0	29	0	8
org.apache.commons.text.io		99%		96%	3	48	0	117	0	10	0	1
org.apache.commons.text.numbers		100%		100%	0	150	0	333	0	72	0	8
org.apache.commons.text.matcher		100%		100%	0	78	0	127	0	49	0	9
Total	482 of 20.455	97%	94 of 2.210	95%	96	1.990	84	4.438	14	878	0	112

Figure 14. JaCoCo's report.

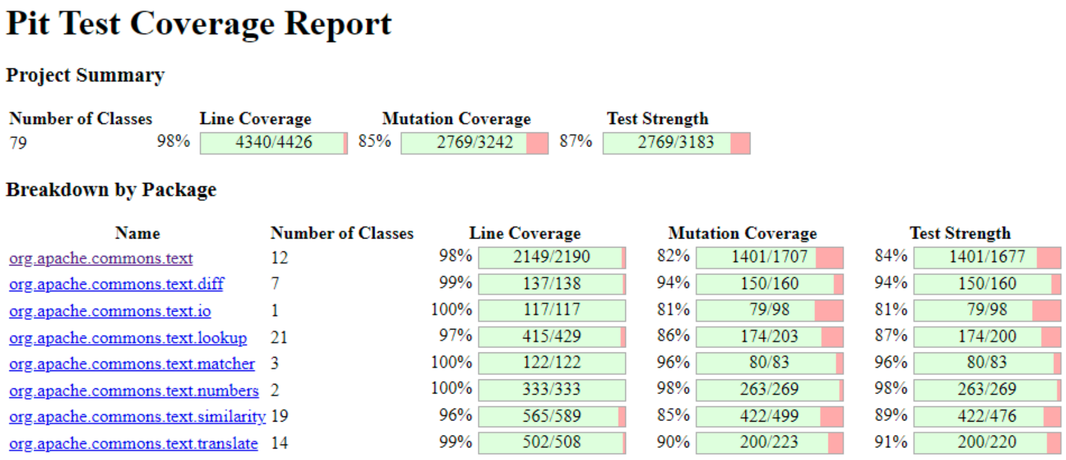
As we can see from the results shown in the figure, the test classes already present in the project guarantee a branch coverage of 95% with only 94 missed branches out of 2.210 total branches and a statement coverage of 97% with only 482 missed instructions out of 20.455 total instructions. A complete report can be found in **jacoco** directory in the project's repository.

6. Mutation Testing using PITest

Mutation testing is a technique used to assess the effectiveness of automated tests. PITest is a popular tool for performing mutation testing in Java projects.
Here's how mutation testing with PiTest works:

- PiTest generates mutants from the project's source code. A mutant is a modified version of the original source code, where a mutation is introduced, such as a changed arithmetic operator, an inverted logical operator, a removed condition, and so on.
- For each generated mutant, PiTest executes the set of automated tests on the mutant and checks if the tests can detect the mutation. If the tests fail for a mutant, it means the mutation has been detected, and the mutant is considered "killed." If the tests pass for a mutant, it means the mutation has not been detected, and the mutant is considered "survived."
- PiTest calculates mutation coverage, which represents the percentage of killed mutants out of the total mutants generated. High mutation coverage indicates that the tests have a good ability to detect mutations in the source code.

In the pom.xml we added the dependency of PITest and we ran the command **mvn test-compile org.pitest:pitest-maven:mutationCoverage** to start the analysis.
At the end of the analysis, we obtained the following result:



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

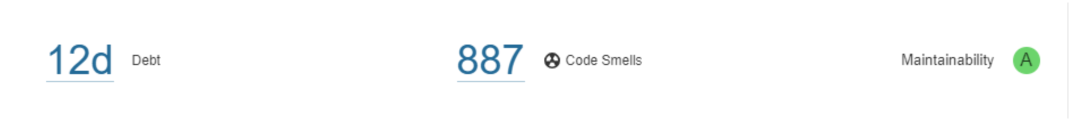
Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.apache.commons.text	12	98%	2149/2190	82%	1401/1707	84%	1401/1677
org.apache.commons.text.diff	7	99%	137/138	94%	150/160	94%	150/160
org.apache.commons.text.io	1	100%	117/117	81%	79/98	81%	79/98
org.apache.commons.text.lookup	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 15. PITest report.

As we can see from the figure, PITest created a total of 3242 mutations in the project. During mutation testing, the tests in the project were able to detect and kill 2769 mutations. This equates to 85% of the mutations created by PITest. Another important metric of the report generated by PITest is Test Strength. Test Strength refers to the level of coverage and robustness of the test set, i.e., it indicates how well the test set is able to detect any defects or errors in the design we are testing. In our case, the total test strength of the project is 87%. A complete report can be found in **pit-reports** directory in the project's repository.

7. SonarQube Analysis using EcoCode plugin

We used SonarQube, which is similar to SonarCloud, with the addition of a plugin called EcoCode to assess the energy efficiency of the project and identify possible optimisations to reduce its consumption. After we configured SonarQube and added the EcoCode plugin to it, we started an analysis of the project with the command **mvn sonar:sonar**. The results showed an increase of code smells compared to the SonarCloud analysis performed previously. In addition to the number of code smells, the debt is also indicated, representing the amount of additional work required to resolve code quality issues in a software project.


12d Debt

887 Code Smells

Maintainability A

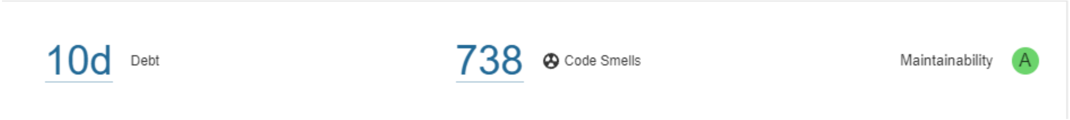
Figure 16. SonaQube with Ecocode Analysis.

As we can see from the figure, the code smells increased from 155 to 887 with a debt of 12 days. In the table below, the code smells tags and their possible refactoring are shown.

Table 2. Code Smells tags and refactoring rules.

Tag	Refactoring
Avoid using global variables	Avoid the use of global variables by replacing them with local variables
The variable is declared but not really used	Remove unusued variables
Using a switch statement instead of multiple if-else if possible	Replace cascading if with a switch
Initialize StringBuilder or StringBuffer with appropriate size	Use a size when you initialize a StringBuilder or StringBuffer
Use ++i instead of i++	Replace variable++ with ++variable
Use System.arraycopy to copy arrays	Use System.arraycopy to copy an array instead of using loops
Do not call a function when declaring a for-type loop	Create a variable and initialize it calling the function before the for-type loop
Avoid usage of static collections	If you want to use static collections make them final and create for example a singleton if needed containing the collections
Avoid getting the size of the collection in the loop	Use in the loop a variable initialized with the size of the collection instead
Avoid the use of Foreach with Arrays	Use a traditional for loop
Do not concatenate Strings in loop, use StringBuilder instead	Concatenating Strings with StringBuilder

We were only able to solve 156 code smells related to EcoCode. This is because most of the code smells belonged to the tags 'Avoid using global variables' and 'Using a switch statement instead of multiple if-else if possible'. For the former, global variables could not be removed because they were being used by multiple methods. For the latter, the addition of the switch led to the appearance of a Major code smell asking to remove the switch statement and add if/else instead.
The final result after the refactoring we performed is shown in the following figure.


10d Debt

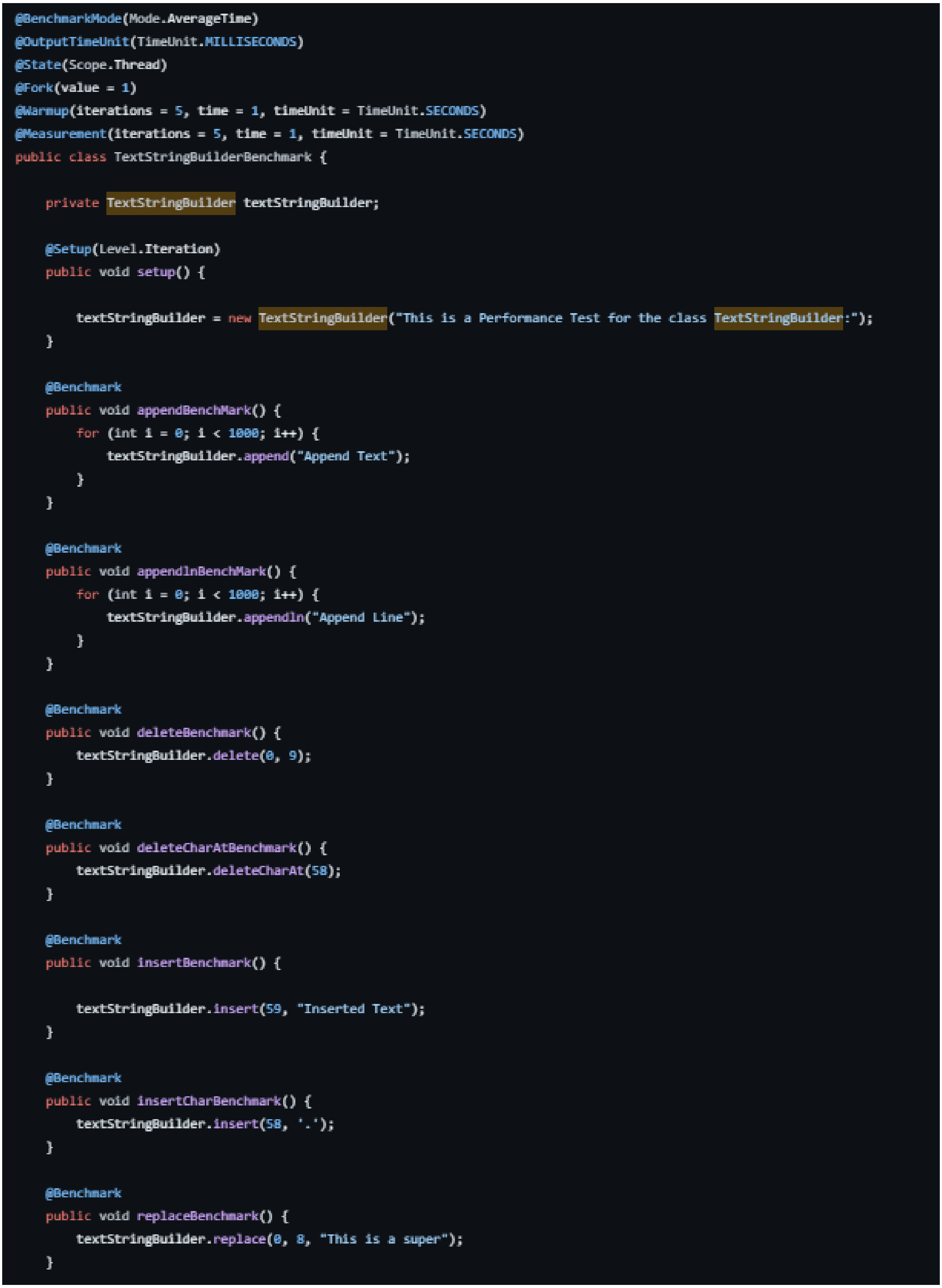
738 Code Smells

Maintainability A

Figure 17. SonaQube with Ecocode final analysis.

8. Performance Testing using Java Microbenchmark Harness

The Java Microbenchmark Harness (JHM) benchmarking framework was used to test the methods of the classes in the project.
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.



```
@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("This is a Performance Test for the class TextStringBuilder:");
    }

    @Benchmark
    public void appendBenchMark() {
        for (int i = 0; i < 1000; i++) {
            textStringBuilder.append("Append Text");
        }
    }

    @Benchmark
    public void appendlnBenchMark() {
        for (int i = 0; i < 1000; i++) {
            textStringBuilder.appendln("Append Line");
        }
    }

    @Benchmark
    public void deleteBenchmark() {
        textStringBuilder.delete(0, 9);
    }

    @Benchmark
    public void deleteCharAtBenchmark() {
        textStringBuilder.deleteCharAt(58);
    }

    @Benchmark
    public void insertBenchmark() {

        textStringBuilder.insert(59, "Inserted Text");
    }

    @Benchmark
    public void insertCharBenchmark() {
        textStringBuilder.insert(58, '.');
    }

    @Benchmark
    public void replaceBenchmark() {
        textStringBuilder.replace(0, 8, "This is a super");
    }
```

Figure 18. TextStringBuilderBenchmark implementation.

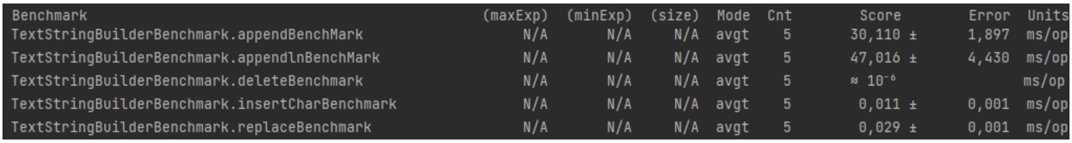
As we can see from the figure, the methods of the TextStringBuilder class that have been tested are the following:

- **append(String)** appends a String;
- **appendln(String)** appends a String followed by a new line;
- **delete(int,int)** delete the characters between the two specified indices;
- **deleteCharAt(int)** delete the character at the specific index;
- **insert(int,String)** insert a string at a specified index;
- **replace(int, int, String)** replace a portion of the string builder with another string;

The test class was configured as follows:

- **@BenchmarkMode(Mode.AverageTime)** this mode calculates the average time taken by the benchmarked code over multiple iterations;
- **@OutputTimeUnit(TimeUnit.MILLISECONDS)** it indicates that the measured time values will be reported in milliseconds;
- **@State(Scope.Thread)** it means that each thread executing the benchmark method will have its own instance of the state object;
- **@Fork(value=1)** to specify the number of times the benchmark should be forked or executed;
- **@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)** it means that the benchmark will undergo a warm-up phase before the actual measurements are taken:
  - **iterations = 5** this specifies the number of iterations to run during the warm-up phase. In this case, the benchmark will be executed 5 times;
  - **time = 1** this specifies the duration of each warm-up iteration. In this case, each iteration will run for 1 second;
  - **timeUnit = TimeUnit.SECONDS** this specifies the time unit for the warm-up duration. In this case, the duration is specified in seconds;
- **@Measurements(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)** it means that the benchmark will be executed for a certain number of iterations or a specific duration to obtain measurements:
  - **iterations = 5** this specifies the number of iterations to run for measurement. In this case, the benchmark will be executed 5 times;
  - **time = 1** this specifies the duration of the measurement phase. In this case, the benchmark will be executed for 1 second;
  - **timeUnit = TimeUnit.SECONDS** this specifies the time unit for the measurement duration. In this case, the duration is specified in seconds;

Subsequently, a main was written to execute all methods of the test class and produced the results shown in the figure.



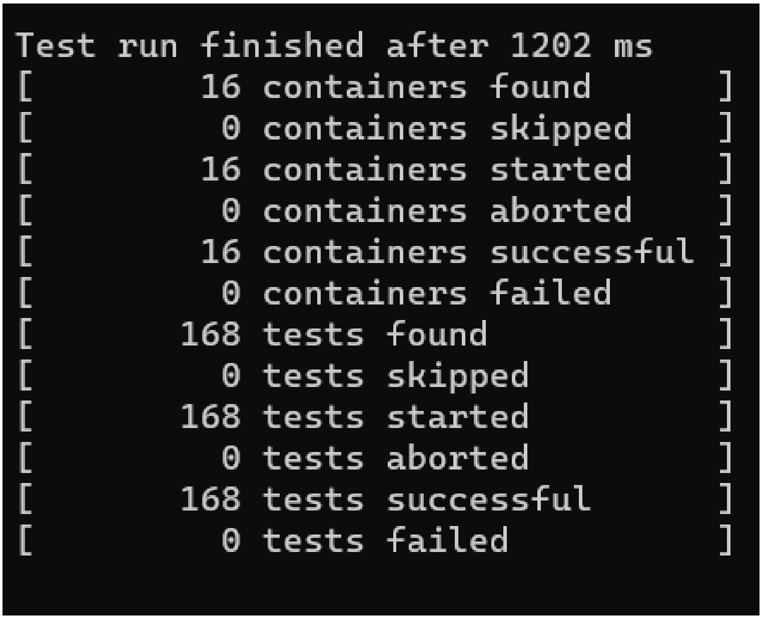
```
Benchmark                                    (maxExp)  (minExp)  (size)  Mode  Cnt   Score        Error  Units
TextStringBuilderBenchmark.appendBenchMark        N/A       N/A     N/A  avgt    5  30,110 ±     1,897  ms/op
TextStringBuilderBenchmark.appendlnBenchMark      N/A       N/A     N/A  avgt    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
```

Figure 19. JMH results.

9. Test Cases gereration using Evo Suite

Evo Suite is an open-source automated test generation tool for Java applications. It uses evolutionary algorithms to automatically generate unit tests with the goal of achieving high code coverage and detecting potential bugs and faults in the software. Evo Suite takes advantage of feedback from code

coverage analysis and fitness functions to guide the evolutionary process. It generates test cases that cover different execution paths and aim to maximize the coverage of the code under test.
Thanks to jacoco's report we have selected the classes with the lowest coverage of the project and we have used Evo Suite to generate new tests for these classes. For a total of 16 classes, 168 tests were generated by Evo Suite. Subsequently, using the Junit Platform Console, all the tests generated by Evo Suite were performed and they were all successful.
The tests created with Evo Suite are present in the evosuite-tests directory of commons-text.

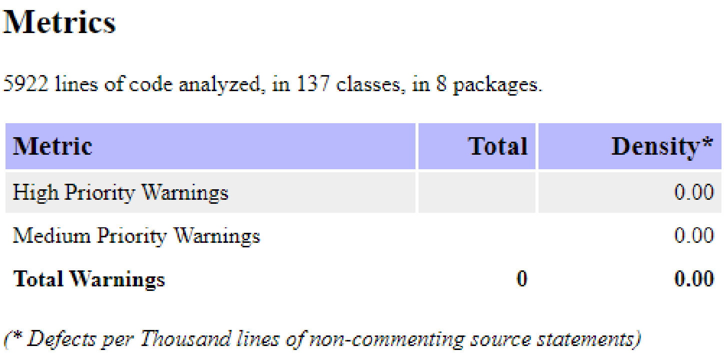


```
Test run finished after 1202 ms
[        16 containers found      ]
[         0 containers skipped    ]
[        16 containers started    ]
[         0 containers aborted    ]
[        16 containers successful ]
[         0 containers failed     ]
[       168 tests found           ]
[         0 tests skipped         ]
[       168 tests started         ]
[         0 tests aborted         ]
[       168 tests successful      ]
[         0 tests failed          ]
```

Figure 20. Evosuite test analysis

10. FindSecBugs Analysis

FindSecBugs is an open-source tool used for static code analysis of Java source code to identify common security vulnerabilities and potential security issues. The main objective of FindSecBugs is to assist developers in identifying and addressing vulnerabilities in the code during the development phase, thereby reducing the risks of attacks and security breaches.
We ran an analysis with FindSecBugs and after analyzing 5922 lines of code, in 137 classes, it found 0 security warnings. A complete report can be found in **findsecbugs-report.html** in the project's repository.



Metrics

5922 lines of code analyzed, in 137 classes, in 8 packages.

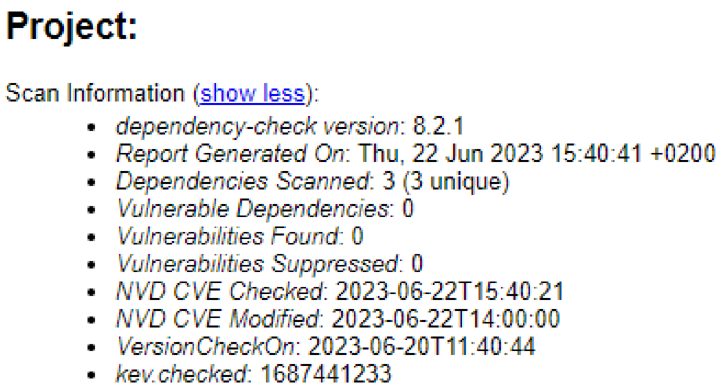
Metric	Total	Density*
High Priority Warnings		0.00
Medium Priority Warnings		0.00
Total Warnings	**0**	**0.00**

(Defects per Thousand lines of non-commenting source statements)*

Figure 21. FindSecBugs Analysis.

11. OWASP DC Analysis

OWASP Dependency Check is a popular open-source tool developed by the OWASP community. It is designed to identify known vulnerabilities in the dependencies used by your software projects. By analyzing the project's dependencies, including libraries and frameworks, Dependency Check helps ensure that you are using up-to-date and secure versions of these components. We ran an analysis with OWASP DC and on 3 dependencies analysed it found 0 vulnerabilities. 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 22. OWASP DC Analysis.

12. OWASP ZAP Analysis

OWASP ZAP (Zed Attack Proxy) is a widely used open-source web application security testing tool. It helps developers, security professionals, and organizations identify and mitigate security vulnerabilities in web applications. ZAP is designed to be easy to use and provides a range of powerful features for security testing.

We did not have the opportunity to use this tool because the project is a library and this tool can only be used only on web applications.

13. Conclusion

In conclusion, we can say that our goal has been achieved. In fact, we were able to analyze the project we chose with the various tools studied during the Software Dependability course and also proceeding to refactor the code when necessary.