



UNIVERSITY OF SALERNO

Computer Science Department

SOFTWARE DEPENDABILITY PROJECT REPORT

Apache Commons Text

CALABRESE CARMINE - 0522501853

2023-2024 Academic Year

Contents

1	Introduction	1
1.1	Structure of the Report	1
1.2	Apache Commons Text	2
1.3	Project Evaluation Criteria	2
2	Library Analysis	3
2.1	Introduction	3
2.2	SonarCloud Analysis	3
3	Refactoring	5
3.1	Bug Fixing	5
3.2	Code Smells Fixing	5
3.2.1	Assertions in Test Cases	6
3.2.2	Duplicated String Literals	6
3.2.3	Naming Conventions for Constants	6
3.2.4	Empty Methods	6
3.2.5	Reducing Cognitive Complexity	6
4	Testing	8
4.1	Coverage Analysis	8
4.2	Performance Testing	9

4.2.1	Metrics	9
4.2.2	Results Before Cognitive Complexity Fix	9
4.2.3	Results After Cognitive Complexity Fix	10
4.2.4	Conclusions	11
4.3	Automated Test Generation	11
4.3.1	Randoop	12
4.3.2	GitHub Copilot	12
4.4	Mutation Testing	12
5	Containerization	13
5.1	Containerization Process	13
5.2	Available Methods	13
6	Vulnerabilities	15
6.1	FindSecBugs	15
6.1.1	Dodgy Code	15
6.1.2	Security	16
6.2	Dependency Check	16
7	Conclusions	17
7.1	Evaluation Criteria	17
7.1.1	Buildability	17
7.1.2	Code Quality	17
7.1.3	Coverage	18
7.1.4	Mutation Testings	18
7.1.5	Vulnerability Assessment	18
7.1.6	Containerization	18
7.1.7	Performance	19
7.1.8	Automation	19
7.1.9	Security	19

CHAPTER 1

Introduction

1.1 Structure of the Report

This project involves an analysis of the Apache Commons Text library. The report is structured as follows:

- **Chapter 1 - Introduction:** A brief overview of the project's objectives and an introduction to the library.
- **Chapter 2 - Library Analysis:** Details of the CI/CD pipeline and the use of analytics tools like SonarCloud.
- **Chapter 3 - Refactoring:** A summary of all the identified bugs and code smells, along with the methodologies used to address them.
- **Chapter 4 - Testing:** An overview of code coverage, the test cases generated to maintain coverage standards, and performance testing, including a quick look at the metrics.
- **Chapter 5 - Containerization:** An explanation of the library's containerization process and the creation of a test web application.

- **Chapter 6 - Software Vulnerabilities:** An analysis of the vulnerabilities identified during the study.
- **Chapter 7 - Conclusions:** A concise summary of the results obtained in the preceding chapters.

1.2 Apache Commons Text

The *Apache Commons Text* library is part of the Apache Commons project, an initiative by the Apache Software Foundation to provide reusable Java components. This library is specifically designed for algorithms and utilities that process text data.

1.3 Project Evaluation Criteria

The project is evaluated based on the following criteria:

- **Buildability:** The library can be built both locally and through the CI/CD pipeline implemented with GitHub Actions.
- **SonarCloud:** SonarCloud is used to identify bugs and code smells in the source code (Chapter 2).
- **Coverage:** Code coverage analysis is performed using Jacoco (Chapter 4).
- **Containerization:** A Docker image has been created to test the library's main methods via a web application (Chapter 5).
- **Performance:** The library's performance is evaluated using JMH benchmarks (Chapter 4).
- **Automation:** Test cases are automatically generated to meet code coverage requirements (Chapter 4).
- **Security:** A study of the library's security, including its classes and dependencies, is conducted (Chapter 6).

CHAPTER 2

Library Analysis

2.1 Introduction

To begin the library analysis, a series of steps were undertaken to prepare a local copy of the Apache Commons Text library for modification and testing:

1. **GitHub Repository Fork:** The repository was forked and cloned onto the local machine. IntelliJ IDEA was used as the primary IDE for modification and analysis.
2. **Study of the CI/CD Pipeline:** The existing GitHub Actions workflow was analyzed to understand the continuous integration and deployment pipeline, ensuring it supported a single Java version (Java 8) for execution.
3. **Pipeline Testing:** Both the local and GitHub-hosted pipelines were tested to confirm their correct functionality across different environments and machines.

2.2 SonarCloud Analysis

SonarCloud was utilized to analyze the project and identify bugs and code smells in the source code. The initial scan produced the following results:

- **Total Bugs:** 4
- **Total Code Smells:** 1.4k

Each issue identified by SonarCloud was assigned a severity level to prioritize their resolution. In this analysis, only issues with high severity were addressed. A total of 46 high-severity issues were identified and categorized as follows:

- **Bugs:** 1
- **Code Smells:** 44

These fixes involved applying best practices in coding and ensuring compliance with the library's functional and performance requirements. The process aimed to enhance the library's overall quality while maintaining compatibility with existing features.

CHAPTER 3

Refactoring

After analyzing the library with SonarCloud, the primary task was to address the high-severity issues by understanding the root causes and implementing appropriate solutions.

3.1 Bug Fixing

The only bug identified during the analysis was related to improper array access. The issue occurred when an array was forced to have a minimum size of $n + 1$ elements, and the n -th element was returned. This could potentially lead to an *ArrayIndexOutOfBoundsException*. To resolve this, a simple control was added to ensure safe access to the array.

3.2 Code Smells Fixing

Addressing code smells was the most time-intensive task, as they accounted for the majority of the issues. The 44 code smells identified were categorized as follows:

- **Refactor this method to reduce its Cognitive Complexity (19).**

- **Tests should include assertions (12).**
- **Duplicated string literals (5).**
- **Constants should comply with naming conventions (3).**
- **Methods should not be empty (5).**

3.2.1 Assertions in Test Cases

All test cases flagged for missing assertions were marked as false positives. These tests were designed to validate the successful execution of methods. Some test cases ensured that exceptions did not occur, and their success criteria were inherently tied to the absence of exceptions. Therefore, explicit assertions were unnecessary.

3.2.2 Duplicated String Literals

For the duplicated string literals, global constants were created to replace the repeated strings, ensuring consistency and reducing redundancy.

3.2.3 Naming Conventions for Constants

All flagged constants adhered to the naming conventions enforced by the project's checkstyle rules. Consequently, these issues were marked as false positives.

3.2.4 Empty Methods

The empty methods were constructors used for JavaBean initialization. To clarify their purpose, comments were added explaining why these methods were intentionally left empty.

3.2.5 Reducing Cognitive Complexity

The cognitive complexity of 19 methods was addressed. The approach involved splitting complex methods into smaller, more manageable ones, adhering to the principle of single responsibility whenever possible. By the end of the process:

- 16 methods met the target complexity value of 15 or less.
- 3 methods could not meet the target due to specific constraints:
 - **readQuote(...)**: This method relied on a single loop with multiple *continue* statements. The loop was the primary contributor to the high cognitive complexity, making it impossible to reduce the value to 15.
 - **StringSubstitutor(...)**: This method had an initial cognitive complexity value of 99. Refactoring it to an acceptable level would have required significant man-hours, which were reallocated to higher-priority tasks.
 - **findDetailedResults(...)**: Despite efforts, the minimum achievable complexity value was 18. Further reduction was not feasible without compromising functionality or clarity.

CHAPTER 4

Testing

Apache Commons-Text already had a comprehensive test suite, which was executed every time the pipeline ran. The results indicate a total of 1,399 tests, with no failures or errors but five skipped tests.

4.1 Coverage Analysis

A Jacoco analysis was integrated into the pipeline with predefined minimum coverage thresholds. If the application fails to meet these thresholds, the build process is aborted, and new test cases must be added. Below are the results of the coverage analysis (Figure 4.1).

One challenge encountered during this phase was handling the JMH-generated class files within the target folder. Jacoco treated these as untested classes, which affected the coverage metrics. To address this, the original threshold values were adjusted.

Table 4.1: Jacoco Coverage Analysis Results

Package	Instruction Coverage (%)	Branch Coverage (%)
<code>commons.text</code>	96	96
<code>commons.similarity</code>	96	89
<code>commons.translate</code>	98	94
<code>commons.lookup</code>	98	97
<code>commons.diff</code>	98	91
<code>commons.io</code>	99	93
<code>commons.numbers</code>	100	100
<code>commons.mather</code>	100	100

4.2 Performance Testing

Performance testing was conducted using JMH (Java Microbenchmark Harness) for a method with high cognitive complexity. The chosen method, `wrap(...)`, wraps long strings by inserting newlines at appropriate points. The test cases covered a range of string lengths to ensure a thorough evaluation.

4.2.1 Metrics

Three key metrics were used to assess the performance of the `wrap(...)` method:

- **Throughput (thrpt):** Measures the number of operations per unit of time.
- **Average Time (avgt):** Measures the average time taken for a single operation.
- **Cold Start (ss):** Measures execution time without warm-up iterations.

The measurements were taken on a nanosecond scale for better granularity. Each test included three warm-up iterations and five execution iterations.

4.2.2 Results Before Cognitive Complexity Fix

The results before reducing the cognitive complexity are presented below:

Table 4.2: Performance Results Before Cognitive Complexity Fix

Benchmark	Mode	Score	Error	Unit
Normal String	thrpt	0.003 ± 0.001		ops/ns
Long String	thrpt	0.002 ± 0.001		ops/ns
Really Long String	thrpt	0.002 ± 0.001		ops/ns
Normal String	avgt	357.037 ± 6.929		ns/op
Long String	avgt	430.752 ± 24.796		ns/op
Really Long String	avgt	460.371 ± 3.647		ns/op
Normal String	ss	60,540 ± 86,034.795		ns/op
Long String	ss	57,760 ± 76,895.633		ns/op
Really Long String	ss	36,520 ± 29,084.982		ns/op

Key observations:

- Longer strings require more time for the `wrap()` method to process.
- Throughput is consistent across string lengths.
- Cold start performance improves slightly with longer strings, though the JVM struggles without warm-up.

4.2.3 Results After Cognitive Complexity Fix

The `wrap(...)` method's cognitive complexity was reduced from 44 to 15. The updated results are as follows:

Key findings:

- Throughput remains mostly unchanged.
- Average time shows that very long strings are now processed more efficiently.
- Cold start performance exhibits notable improvement, likely due to reduced method complexity.

Table 4.3: Performance Results After Cognitive Complexity Fix

Benchmark	Mode	Score	Error	Unit
Normal String	thrpt	0.002 ± 0.001		ops/ns
Long String	thrpt	0.002 ± 0.002		ops/ns
Really Long String	thrpt	0.001 ± 0.001		ops/ns
Normal String	avgt	698.805 ± 80.937		ns/op
Long String	avgt	856.253 ± 21.450		ns/op
Really Long String	avgt	646.583 ± 654.257		ns/op
Normal String (Cold)	ss	58,740 ± 152,799.888		ns/op
Long String (Cold)	ss	46,360 ± 31,043.212		ns/op
Really Long String (Cold)	ss	38,200 ± 66,963.573		ns/op

4.2.4 Conclusions

While reducing cognitive complexity improves maintainability, it may not always enhance performance. In this case:

- Cold start performance improved due to reduced complexity.
- Average execution time increased for shorter strings but decreased for very long strings.
- The trade-off between maintainability and performance optimization must be carefully evaluated.

4.3 Automated Test Generation

Automated tools were used to generate test cases for selected classes. The chosen classes were:

- `DockerContainer.java`: Manages the server within the Docker image.
- `BenchmarkRunner.java`: Handles performance testing of the library.

4.3.1 Randoop

Randoop provided mixed results:

- For `BenchmarkRunner.java`, no test cases were generated.
- For `DockerContainer.java`, generated test cases improved coverage marginally but required significant manual adjustments.

4.3.2 GitHub Copilot

GitHub Copilot proved more effective:

- Generated test cases that met minimum coverage standards.
- Required iterative prompts and manual corrections to achieve optimal results.
- Occasionally introduced unnecessary dependencies, which had to be manually resolved.

AI-assisted tools can significantly help testing but require oversight to ensure correctness and relevance.

4.4 Mutation Testing

Mutation Testing aims to evaluate how well the test suite behaves by changing some portions of the source code exercised by the test cases. The mutation testing campaign was conducted by using PiTest. The results of this testing shows that of all the 97 classes:

Table 4.4: Mutation Testing Results

# of classes	Line Coverage	Mutation Coverage	Test Strength
91	97% [5891/6071]	83% [3954/4722]	85% [3954/4631]

CHAPTER 5

Containerization

This chapter provides a brief description of the Docker image and the containerization process.

5.1 Containerization Process

The containerization process involves the following steps:

- A JAR file of the library is generated.
- A web application is implemented to test specific methods from the library.
- A Docker image is created and uploaded to DockerHub.

The resulting Docker image can be executed through the terminal. Running the image starts a server, enabling users to test selected methods via the web application interface (see Figure 5.1).

5.2 Available Methods

The web application allows users to test the following four methods:

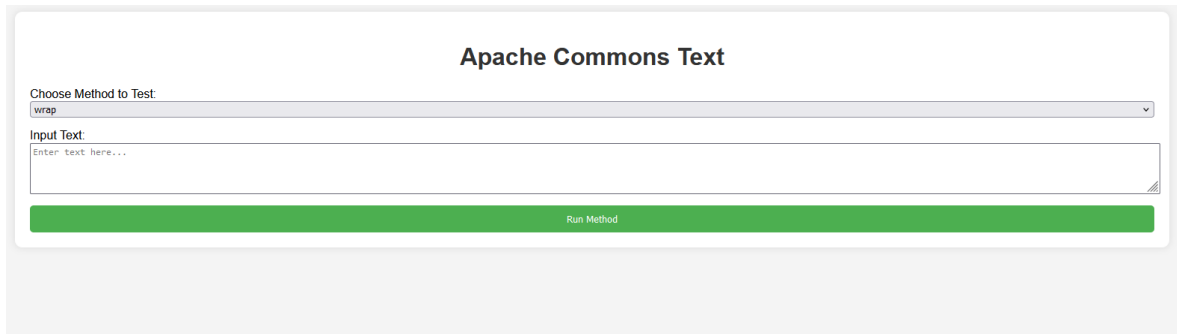
The screenshot shows a web application interface titled "Apache Commons Text". It features a dropdown menu labeled "Choose Method to Test" with "wrap" selected. Below this is a text input field labeled "Input Text:" with a placeholder "Enter text here...". At the bottom of the interface is a green button labeled "Run Method".

Figure 5.1: Web Application Interface

- **Wrap:** Wraps a long string into multiple lines, with a default line length of 20 characters.
- **Capitalize:** Retrieves the first character of each word and capitalizes it.
- **Initials:** Extracts the initial character from each word and returns it as a string.
- **Random:** Generates a random string of random characters, given a specified length.

These four methods were selected because the majority of the library's other methods are either private or require input objects that are incompatible with JavaScript.

CHAPTER 6

Vulnerabilities

A vulnerability scan was conducted using two different tools: FindSecBugs to identify vulnerabilities within the library's source code and OWASP Dependency-Check (DC) to analyze vulnerabilities in the dependencies used by the library.

6.1 FindSecBugs

The scan with FindSecBugs identified a total of 15 issues, categorized as follows:

- **Dodgy Code:** Code that does not impact the correct execution of methods but does not adhere to best practices (9 issues).
- **Security:** Code that could compromise the security of the library, such as potential injections or predictable logic (6 issues).

6.1.1 Dodgy Code

Among the 9 issues identified in this category, only one was fixed. This issue arose during the refactoring of methods with high cognitive complexity. It involved a local variable whose value was modified but never subsequently used in the code. This was corrected by removing the unnecessary variable.

For the remaining 8 issues, no fixes were applied for the following reasons:

- Similar to the fixed issue, some problems involved local variables whose values were changed but never used. However, these cases occurred in classes generated by JMH, so no modifications were made to these autogenerated classes.
- Two issues involved catching `NullPointerException`, which is generally discouraged. However, in this case, the exceptions were intentionally caught by the original developers. Removing these exception handlers would require rewriting a new set of test cases. To preserve the integrity of the original test cases, no changes were made.

6.1.2 Security

The security-related issues require additional considerations. While the bugs identified are valid, they are deeply embedded in the library's design. Resolving them would necessitate significant changes across the entire library, creating a cascade of required modifications. These changes would involve considerable man-month effort, making them impractical within the scope of this project.

6.2 Dependency Check

A Dependency-Check scan was conducted, revealing no vulnerabilities in the dependencies used by the library. The `commons-text` library relies on five dependencies: `commons-math`, `commons-lang`, `jmh`, `jmh-annotations`, and `jopt-simple`. Of these, two dependencies were added during this project's analysis. None of the dependency versions used have any known vulnerabilities, as confirmed by the scan results.

CHAPTER 7

Conclusions

In this chapter, we summarize the findings and experiences from the analysis and development process, evaluating the project based on the evaluation criteria. Each criterion is addressed individually, reflecting on the outcomes and on the success of it, or not.

7.1 Evaluation Criteria

7.1.1 Buildability

The project after at the end of the analysis can be built both locally and with the CI/CD. Other changes were needed to be able to allow the build to pass the quality gate (that at the start didn't pass because of Reliability Issue).

7.1.2 Code Quality

The quality of the code was assessed using tools like SonarCloud and FindSecBugs. While several issues were identified, many were minor or related to autogenerated code. All the high severity issue were addressed and fixed.

7.1.3 Coverage

Coverage test passed successfully by modifying the original value, but this is mostly because jacoco recognize the deprecated class as part of the library and without a way to exclude this file there is no possibility to pass the coverage with the initial value imposed by the apache group.

7.1.4 Mutation Testings

A mutation testing campaign was conducted to assess the completeness and robustness of the test cases. The results were solid, indicating that the tests are effective at identifying issues and ensuring the reliability of the library's source code.

7.1.5 Vulnerability Assessment

Through tools like FindSecBugs and OWASP Dependency-Check, the library was rigorously analyzed for vulnerabilities. While no critical issues were found in the dependencies, the library's source code revealed areas where improvements could be made. Addressing these vulnerabilities, particularly those related to injection risks, would enhance the library's robustness.

7.1.6 Containerization

The containerization process revealed both successes and challenges. A Docker image was successfully created, along with a functioning web application for testing. However, an issue arose when HTTP requests to invoke the methods failed due to missing external dependencies inside the image. Several approaches were attempted to resolve this:

- A "fat" JAR was created to bundle all dependencies, but some dependencies could not be included.
- The entire folder containing the dependencies was added to the Docker image to ensure they were available. Despite this, the application still failed to recognize the dependencies.

The only way to successfully test the web application was by executing the main class within the `DockerContainer.java` file, which worked without errors.

7.1.7 Performance

The performance testing provided valuable insights into the trade-off between performance and explainability. While reducing cognitive complexity improved the maintainability and future modification of the methods, it inadvertently resulted in a decrease in performance. This highlights a common challenge in software development, where optimizing for one aspect can negatively impact another.

7.1.8 Automation

Automated testing tools, such as AI-driven tools and Randoop, were explored for their potential to assist in generating test cases. Although these tools do not always produce perfect test cases, they were helpful in kickstarting the creation of test classes and improving the overall test coverage. In particular, they provided a solid foundation to build upon and ensure that the system behaves as expected under various scenarios.

7.1.9 Security

The security analysis revealed several issues that need to be addressed. However, given the scope and time constraints of this university project, resolving these vulnerabilities would require significantly more effort. While some issues are more easily fixable, others involve deeper changes that would demand extensive modifications to the core library. Thus, addressing these security concerns is a challenge that may need to be pursued in future work, possibly requiring additional resources and time.