Software Dependability project
Apache Commons Codec™
Teacher: Prof. Dario di Nucci

Student: L. Falanga, l.falanga@studenti.unisa.it[a]

[a]*MAT. [0522502019] UNISA, Università degli Studi di Salerno, Dipartimento di Informatica (Dipartimento d'eccellenza)*

January, 2025

Abstract

The software dependability analysis project conducted through the **Apache Commons Codec™**, an open-source collection of components that provides algorithms for encoding and decoding data.
The primary goal was to evaluate the behavior of the library under various conditions, identifying potential vulnerabilities or malfunctions.
The analysis focused on evaluating the robustness of key functions, their ability to handle invalid inputs, system errors, and stress conditions, code coverage, quality, test cases, and performance.
In particular it is created in order to consolidate Base64 encoder development efforts across projects.
Apache Commons Codec link

Figure 1: Apache Commons Codec ™ logo

1. Introduction

1.1. Building the Project

The project was forked from the GitHub repository and built locally.
My GitHub repo link

I forked from an existing repository and then managed using IntelliJ IDEA, a robust integrated development environment (IDE).
Maven was utilized for build management and dependency handling.
After forking, the repository was synchronized with GitHub to facilitate the implementation of CI/CD (Continuous Integration and Continuous Deployment) pipelines.
This setup allowed for the application of dependability engineering techniques, ensuring enhanced stability, performance, and fault tolerance throughout the development lifecycle.

2. SonarCloud Analysis

The project's quality and security were evaluated through SonarCloud, a platform that scans the code and performs static analysis to identify bugs, code smells, and potential vulnerabilities.
The analysis yielded the following outcomes:

- 2 bug
- Over 1400 code smells
- 0 vulnerabilities
- 31 security hotspots
- 1.5% code duplication

So, firstly I solved 1/2 bug because unnecessary and unboxing should be avoided.

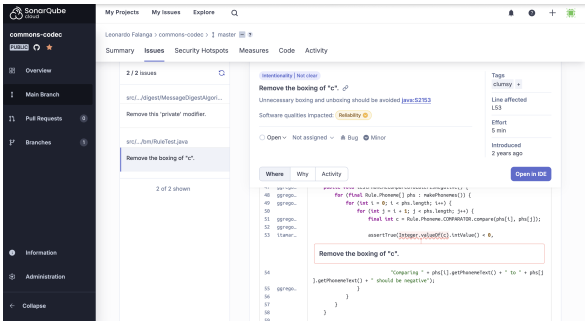

Figure 2: First bug solved

For the second bug instead "Remove this 'private'

modifier": JUnit5 test classes and methods should not be silently ignored.
JUnit5 is more tolerant regarding the visibilities of Test classes and methods than JUnit4, which required everything to be public.
JUnit5 supports default package, public and protected visibility, even if it is recommended to use the default package visibility, which improves the readability of code. But solving this problem causes a fatal error.
The message indicates that in the JUnit 5 test class, I have one or more classes or methods defined with the private access modifier.
This is a problem because JUnit 5 automatically ignores these entities, making the tests unusable without any warning, which can compromise software reliability.
But changing this option causes problems.
1. If the method or class is designed to be used only internally and contains references or logic closely related to the context in which it is defined, remove private might:

- Expose the method to a wider level of visibility, allowing external classes to access it in an undesirable way.
- Alter the behavior of the code if other methods or frameworks begin to invoke it improperly.

or

- Some frameworks or libraries (such as JUnit) may be configured to look for private methods specifically. In these changing the access modifier may make the method not compatible with framework conventions.
  or other possible errors.
  So I considered this bug a **false positive**

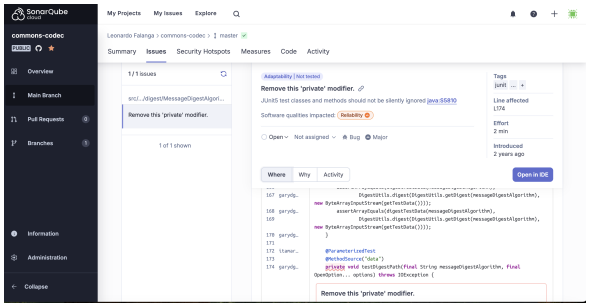

Figure 3: Second bug

2.1. Code smell

The refactoring of the code was carried out by addressing the identified code smells (from 1418 to 1283. This involved various actions, such as adding or removing comments, declaring variables, introducing default break statements in different classes and test classes, eliminating deprecated code, and modifying certain access modifiers.
These improvements aimed at enhancing code readability, maintainability, and overall quality.
The details of the changes can be found in the commit history, visible under the GitHub Actions tab and SonarCloud's "fixed" filter.

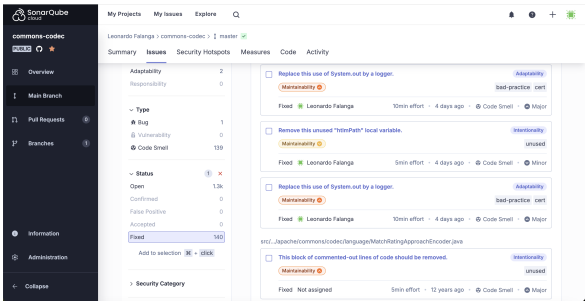

Figure 4: Code smell solved

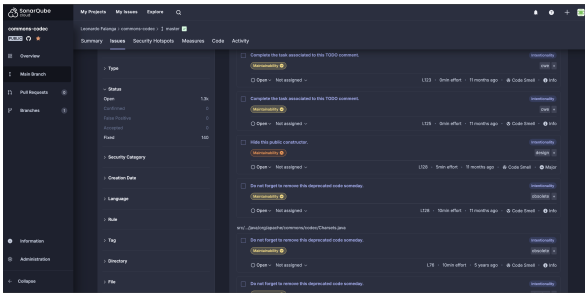

Figure 5: Total issues solved

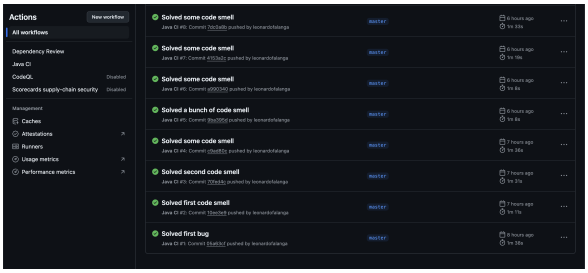

Figure 6: GitHub actions

2.2. SonarCloud bug

During the Christmas vacations, I encountered a bug with SonarCloud's "Severity" filters.
I proceeded to check with my colleagues who experienced the same error and wrote to technical support.

UPDATE: As of today 07/11/2025, updating my

report I found that they have fixed the bug.
So here are the issues fixed with the severity filter.

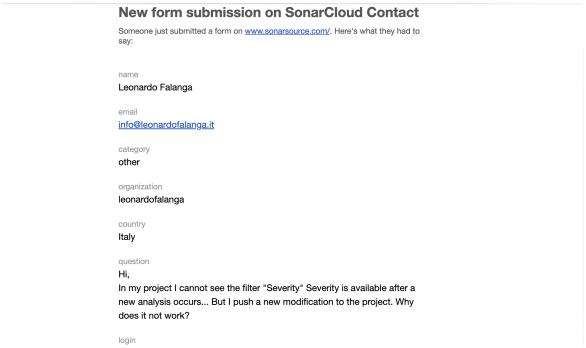

Figure 7: Sonarcloud bug report

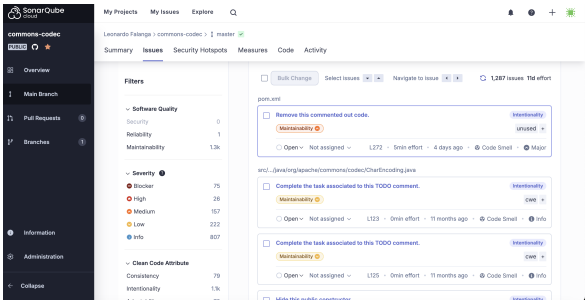

Figure 8: Severity filter

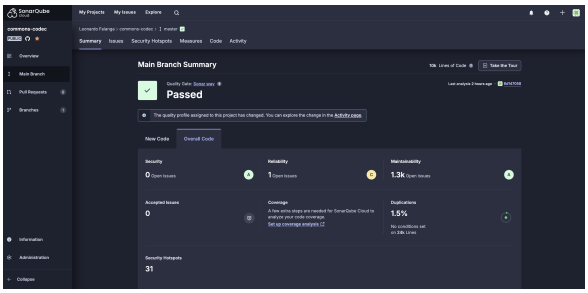

Figure 9: Quality Gate

3. Docker

Docker is a powerful platform that simplifies the process of building, shipping, and running applications in a consistent environment.
By leveraging containerization, Docker enables developers to isolate their applications from the underlying system infrastructure, ensuring that the software runs seamlessly across different environments.
This separation not only accelerates the software delivery lifecycle but also enhances reliability and scalability.
To package and run my Maven-based project within a Docker container, I followed a process commonly referred to as "Dockerizing" the application.
This docker image is available on: Docker image

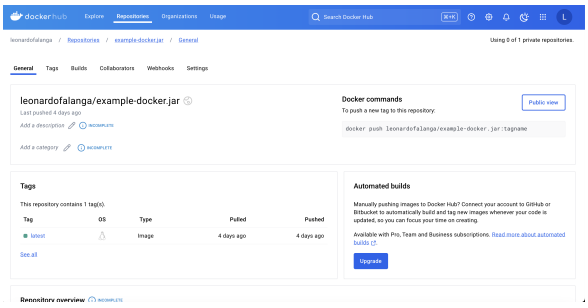

Figure 10: Docker

In ExampleDocker, I set up a web application accessible via an HTML interface served on port 8080. The interface allows users to encode text by interacting with the server.

Here's a detailed description of the setup:

1. HTML Page:
   - The index.html file provides a user-friendly interface where users can input text to encode.

   The page contains:
   - A textarea for input.
   - An "Encode" button to submit the text.
   - A section to display the encoded result.
2. Backend Integration:
   - When the "Encode" button is pressed, the page sends the input text to the server at the endpoint http://localhost:8080/encode using a POST request with text/plain content type.
3. Server Response:
   - The backend processes the input and returns the encoded result.
   - The JavaScript in the HTML updates the interface to display the encoded result in a bold, blue style.

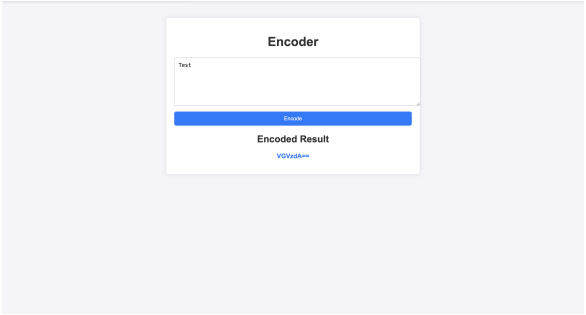

Figure 11: Web application

4. Code coverage analysis

JaCoCo is a powerful code coverage tool specifically designed for Java applications.
In essence, It's a tool that measures how much of my Java code is actually executed when I run my tests.
This metric provides invaluable insights into the quality and comprehensiveness of your test suite.

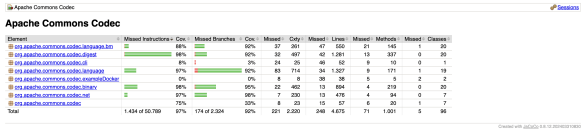

Figure 12: JaCoCo

4.1. Interpretation of results

High Coverage: Most of the packages (e.g., org.apache.commons.codec.language, org.apache.commons.codec.digest) show high coverage, indicating that a large part of the code has been tested.
Critical Points: The org.apache.commons.codec.cli package shows very low coverage, suggesting that it may need to be improved with new testing.

5. Mutation testing

PiTest is a good a popular open-source tool for performing mutation testing in Java projects.
Mutation testing is a powerful technique to assess the effectiveness of your test suite.
I took one package... a snapshot of the mutation testing results for the org.apache.commons.codec.net package is here:

Figure 13: PITest - Codec package coverage

- Mutation Coverage: 87% (252/290). This means that out of 290 mutations introduced by PiTest, your tests were able to detect 252 of them. A higher mutation coverage score generally indicates a more robust and effective test suite.
- Test Strength: 94% (252/268). This metric combines line coverage with mutation coverage to provide an overall assessment of your test suite's strength. A high test strength score suggests that your tests are not only covering a significant portion of the code but also effectively detecting mutations.

You can see now the results for the individual classes here:

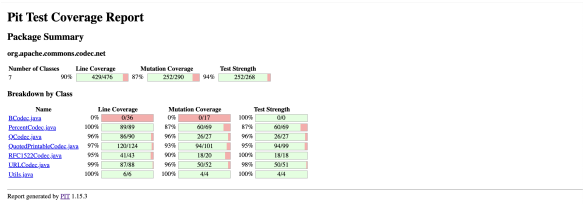

Figure 14: PITest - classes coverage

6. Automatic test case generation

6.1. Randoop

Randoop is a powerful tool that automatically generates unit tests for Java code. All the 1296 tests were successfully tested.

7. Performance testing

JMH (Java Microbenchmark Harness) is a specialized tool designed for creating and running microbenchmarks in Java.
It's ideal for measuring the performance of small code snippets at a very fine-grained level.
I created a new class BenchmarkRunner in which I added benchmark tests.

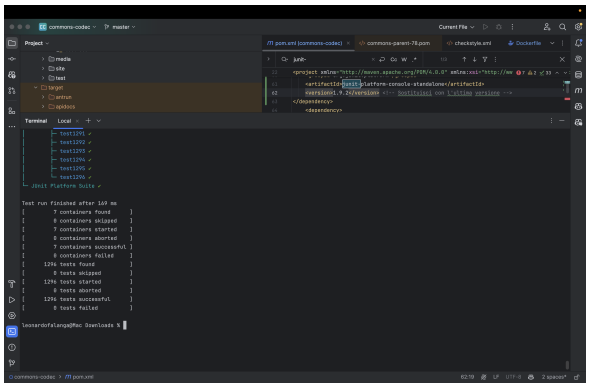

Figure 15: Randoop

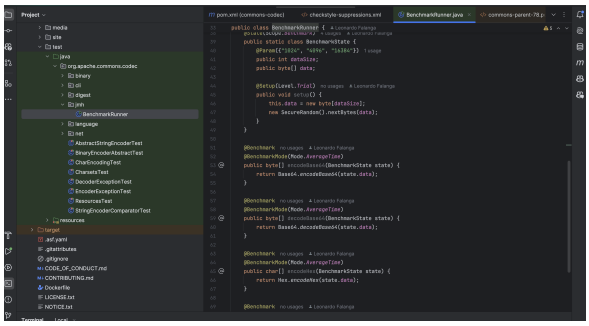

Figure 16: Benchmarks code

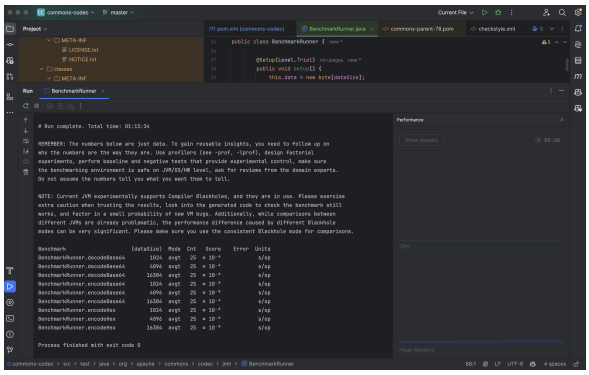

Figure 17: Benchmarks classes

It took about 1 hour and half:

BenchmarkState Class: This class is annotated with @State(Scope.Benchmark), indicating it holds data used across benchmarks.
It has a dataSize parameter that controls the size of the data to be encoded/decoded (1024, 4096, or 16384 bytes).
It also has a data byte array field to store the random data generated in the setup method. The setup method initializes the data array with random bytes using SecureRandom.

Benchmark Methods: The class defines four benchmark methods:

- encodeBase64: This method encodes the data using the Base64 encoder and returns the encoded bytes.
- decodeBase64: This method decodes the Base64 encoded data (assumed to be available) and returns the decoded bytes.
- encodeHex: This method encodes the data using the Hex encoder and returns the encoded character array.
- decodeHex: This method decodes the Hex encoded data (assumed to be available as a String) and returns the decoded bytes.

Benchmark annotations: Each benchmark method is annotated with @Benchmark to indicate it's a benchmark method.
The @BenchmarkMode(Mode.AverageTime) annotation specifies that the benchmark should measure the average execution time.

8. Important POM errors

Since JMH creates a bunch of "dirty" files. , checkstyle in the pom caused me to fail actions on GitHub several times and I spent a week figuring out this problem. I used to run the command "mvn –errors –show-version –batch-mode –no-transfer-progress" but it gave me no error and that screwed me over.

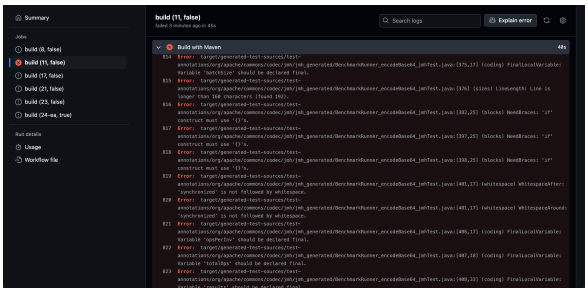

Figure 18: Christmas present :D

9. SW vulnerabilities

To conduct the static security analysis of the project, I used the tools **SpotBugs** and **OWASP DC** but in order to use these tools there was a problem.
The problem was that both tools were performable with Java 8, so I created a virtual machine with Ubuntu to perform the analysis.

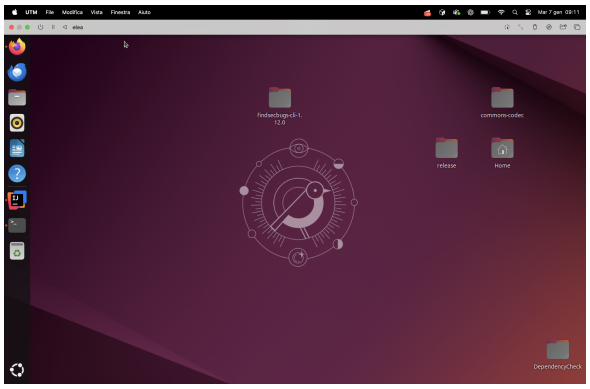

Figure 19: Ubuntu clear installation

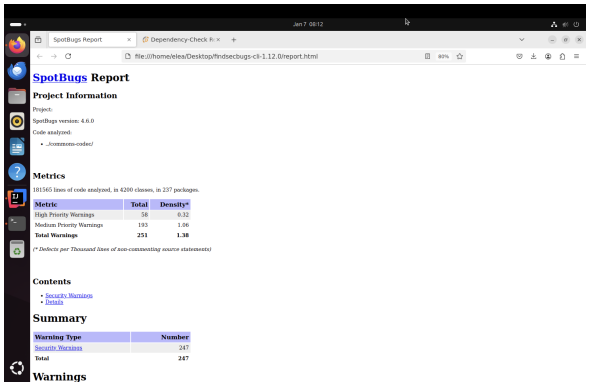

Figure 20: SpotBugs

9.1. SpotBugs

SpotBugs is a static code analysis tool designed to detect potential bugs in Java code.

SpotBugs version used: The version of SpotBugs used for the analysis is 4.6.0.
Code analyzed: Code in the commons-codec package was analyzed.
Metrics:
Lines of code: 181,565 lines of code were analyzed.
Classes: 4200 classes were analyzed.
Packages: 237 packages were analyzed.

Warnings:

1. High Priority Warnings: 58 high priority warnings were found, indicating potential serious bugs that could cause significant problems in the application.
2. Medium Priority Warnings: 193 medium priority warnings were found, which could indicate potential less serious but still noteworthy bugs.

- **Total Warnings**: A total of 251 warnings were found.

9.2. OWASP DC

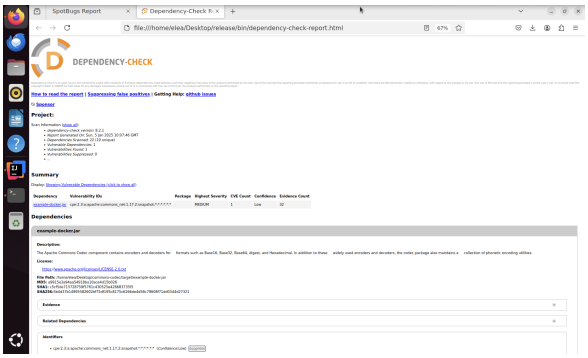

Figure 21: OWASP DC

With Dependency-Check I found just on vulnerability with medium severity, low confidence for to the .jar I created for docker. Amazing :).

Report written in LaTeX using Overleaf.