Software Dependability Project, supervised by Prof. DARIO DI NUCCI

Apache Commons CLI

ZAKARYA BOUDRAF, University of Salerno, Internet of Things, Department of Computer Science

Matricola: 0522501649

Supervisor: Prof. DARIO DI NUCCI

The work accomplished in this project aims for the improvement of the quality of software, its security and its reliability. These goals were achieved through the analysis of code quality, code coverage, test cases, and performance of different software components. The work was done over an existing Apache Commons project (Apache Commons CLI), that uses Java as a programming language.

Additional Key Words and Phrases: Software Dependability, Code Coverage, Testing, Mutation Testing, Formal Verification, Security Analysis, Containerization, CI/CD.

---

1. Introduction

This work is realized as an end-of-course project for the Software Dependability course at the University of Salerno. In this project, the focus is on the analysis of software quality, of code coverage, of test cases, and of the performance of the project's components. This project followed the outlined project evaluation criteria as provided on the e-learning platform by Prof. Di Nucci.

1.1 Choice of Project

The project of choice for this analysis is **Apache Commons CLI**. This project provides an API for parsing command-line options passed to programs. It is a widely-used utility library that powers tools such as Apache Maven, Apache Ant, and Apache Kafka. This project can be found on the Apache Commons website https://commons.apache.org/proper/commons-cli/. Apache Commons CLI is built in Java language, using Maven as a build tool.

1.2 Preparation

In order to start working on this project, the project was forked from the repository at https://github.com/apache/commons-cli. The forked repository is available at https://github.com/ZakaryaBoudraf/commons-cli and it includes the project report in addition to the project itself.

1.3 Project Architecture

In order to better understand the project before conducting the analysis here's a summary of the project architecture:

Apache Commons CLI has a three-stage processing architecture: Definition, Parsing, and Interrogation.

- **Definition Stage**: Developers define the valid command-line schema using the `Options` class, employing the `Option.Builder` pattern.
- **Parsing Stage**: The `DefaultParser` processes the string array of arguments against the defined schema.
- **Interrogation Stage**: The resulting `CommandLine` object provides methods to query the parsed values.

From a dependability perspective, the `CommandLineParser` is the critical component, as it handles untrusted input and must validate data types and manage edge cases.

---

2. CI/CD & Build Automation

According to the project evaluation criteria the application needs to be buildable in CI/CD and locally.

This section covers the continuous integration setup for the project, including the tools used and the build matrix strategy, as well as building the application locally.

2.1 Tools

GitHub Actions was used to implement the CI/CD pipeline. The pipeline was defined in `.github/workflows/maven.yml`. A Build Matrix strategy was employed to test the project across multiple Java versions: Java 8, 11, 17, 21, 25, and 26-ea.

2.2 GitHub Actions Build Results

The execution results validate the stability of the codebase across the entire matrix. The total duration for the pipeline was **1 minute 25 seconds**. All 6 jobs executed in parallel and completed successfully.

The table below shows the build results:

JDK Version	Status	Purpose
Java 8	Passed	Backward compatibility for legacy systems
Java 11 (LTS)	Passed	Previous LTS validation
Java 17 (LTS)	Passed	Current industry standard
Java 21 (LTS)	Passed	Modern features support
Java 26-ea	Passed	Future-proofing

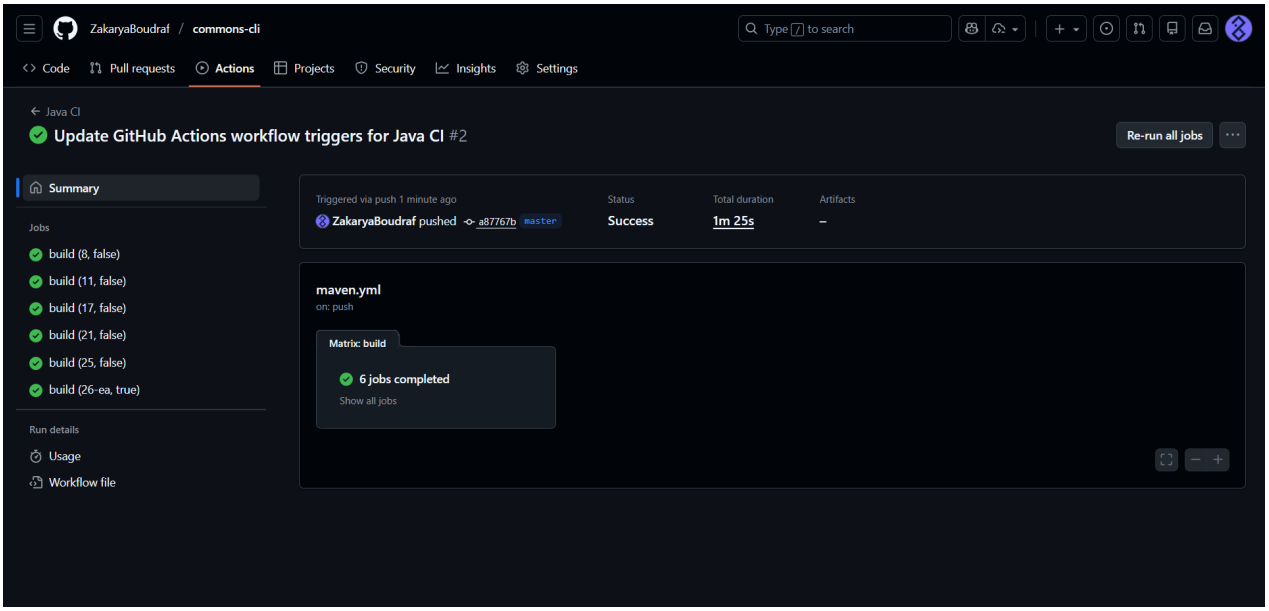

Figure 1: GitHub Actions CI Summary Build matrix showing successful builds across all Java versions.

2.3 Local Build

In accordance to the project evaluation criteria the project was built locally using maven.

The Maven log output displays the following test summary:

```
[WARNING] Tests run: 960, Failures: 0, Errors: 0, Skipped: 61
```

The codebase passes 960 active tests without failure or error. However, 61 tests are currently being skipped. This represents potential technical debt that should be audited.

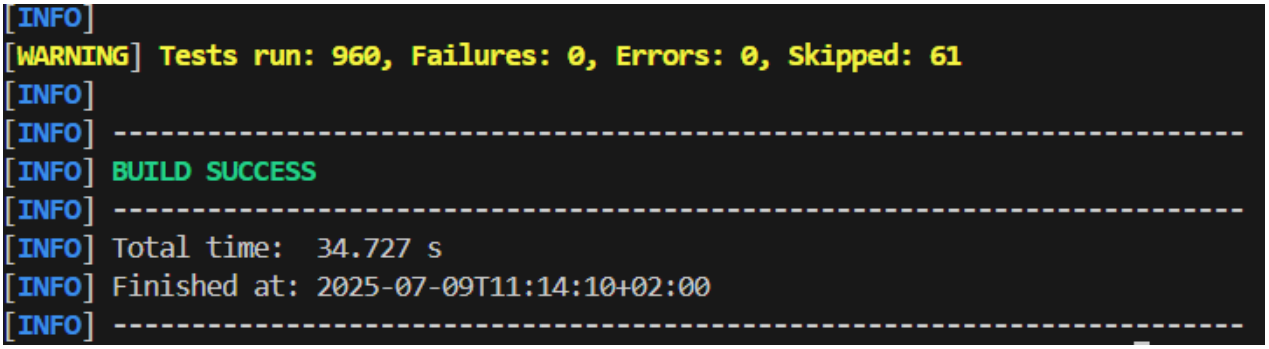

Figure 2: Maven Build Logs

The `[INFO] BUILD SUCCESS` in 34.727 seconds is confirmation that the build works.

3. Code Coverage Analysis (JaCoCo)

According to the project evaluation criteria I need to analyze the code coverage using JaCoCo as well as analyzing the test cases using PiTest for a mutation testing campaign.

This section covers the code coverage analysis of the project, including tools used, findings, and mutation testing results.

3.1 Tools

JaCoCo (Java Code Coverage) is a free code coverage library for Java, which has been created by the EclEmma team based on the lessons learned from using and integrating existing libraries for many years.

PIT (Parallel Isolated Test) is a state of the art mutation testing system, providing gold standard test coverage for Java and the jvm. It's fast, scalable and integrates with modern test and build tooling.

3.2 Code Coverage Results

After modifying the pom.xml file to implement the `jacoco-maven-plugin` (version **0.8.11**).

I ran the test using the following maven command:

```
mvn clean test
```

The report was generated at: `target/site/jacoco/index.html`.

The result shows the following coverage metrics:

- **Instruction Coverage**: **98%** (8,742 of 8,874 instructions)
- **Branch Coverage**: **95%** (953 of 994 branches)

The table below shows coverage by package:

Package	Instruction Coverage	Branch Coverage	Missed Complexity
org.apache.commons.cli	98%	95%	43 of 827
org.apache.commons.cli.help	99%	98%	7 of 242

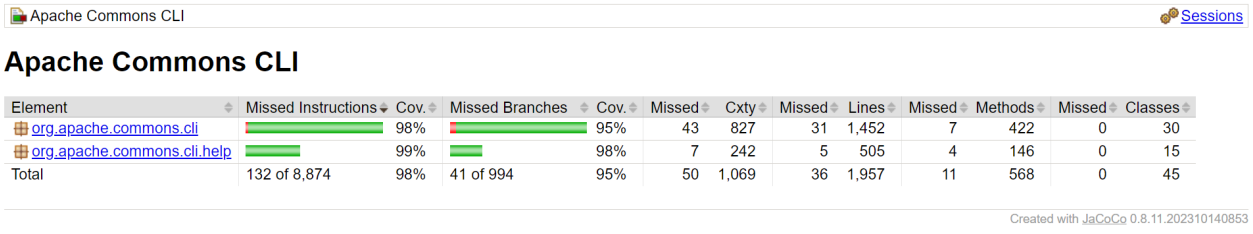

Figure 3: JaCoCo Coverage Report

3.3 Mutation Testing by PIT

Mutation testing using PITest, is a method utilized to evaluate the effectiveness of a test suite by injecting defects, referred to as mutations, into the code. As a result, it checks if the existing tests can identify and capture (kill) these mutations.

I initiated the testing process by adding the PIT plugin to the pom.xml file and executed the following command:

```
mvn org.pitest:pitest-maven:mutationCoverage
```

3.4 Mutation Testing Results

The report was generated at: `target/pit-reports/index.html`.

The PIT report shows the following metrics:

Metric	Value	Details
Line Coverage	64%	93 of 146 lines
Mutation Coverage	65%	60 of 93 mutants killed
Test Strength	90%	60 of 67 covered mutants killed

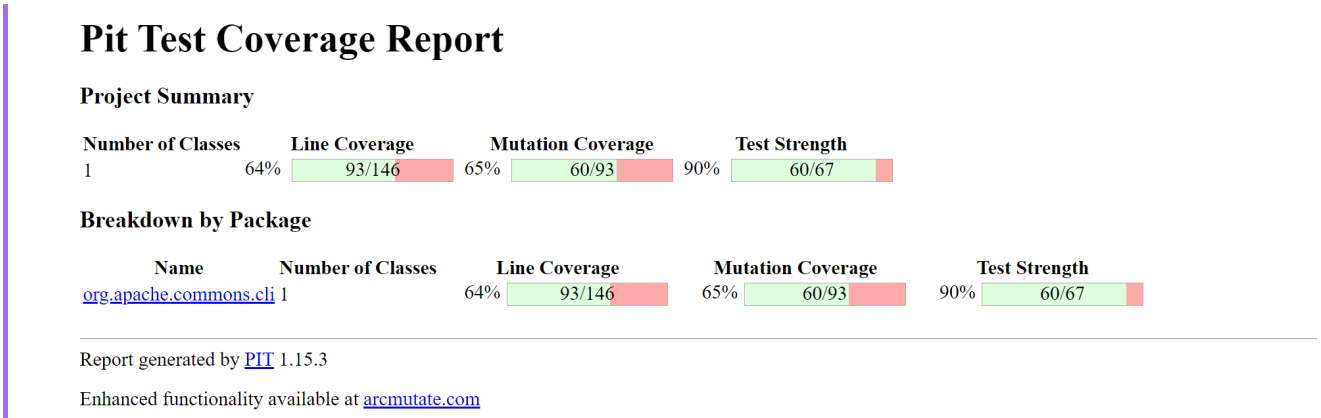

Figure 4: PIT Mutation Coverage Report

The Test Strength of 90% indicates that when tests cover a piece of code, they are effective at detecting faults. The lower Mutation Coverage (65%) compared to JaCoCo coverage (98%) is due to the PIT run being targeted at a specific class rather than the full library, as full mutation testing is computationally expensive.

---

4. Formal Verification with OpenJML

According to the project evaluation criteria, the core methods of the application require formal specification in JML.

This section details the formal verification process applied to the `commons-cli` library.

4.1 Tools & Environment

OpenJML was selected for formal verification as it enables rigorous static analysis of Java Modeling Language (JML) specifications. The verification process was executed in a **WSL (Windows Subsystem for Linux) Ubuntu** environment to leverage the tool's native Linux stability.

- **Tool:** OpenJML (Command Line Interface)
- **Solver:** Z3 (SMT Solver)
- **Method:** Extended Static Checking (`-esc`), which attempts to mathematically prove that the implementation satisfies the specified contracts.

4.2 JML Specifications Applied

I focused our verification efforts on the most critical domain entities: `Option` and `Options` .

Example 1: Option Class Invariants To prevent invalid state, Ienforced that every option must have at least a short name or a long name, and a valid argument count.

```
/*@ public invariant this.option != null || this.longOption != null;
  @ public invariant this.argCount >= -1;
  @*/
```

Example 2: Options.addOption() Contract I verified that adding an option correctly registers it in the internal lookup tables and returns the current instance for method chaining.

```
/*@ requires opt != null;
  @ ensures hasOption(opt.getOpt()) || hasLongOption(opt.getLongOpt());
  @ ensures \result == this;
  @*/
public Options addOption(Option opt)
```

4.3 Verification Results

The OpenJML Extended Static Checker (ESC) was executed against the annotated source code. In formal verification tools like OpenJML, a "clean run" (no output) indicates that the solver successfully proved all assertions.

As shown in **Figure 5**, the verification process completed without flagging any contract violations, and the project built successfully. This confirms that the implementation of the `commons-cli` core classes adheres to the formal contracts defined above, providing a strong mathematical guarantee of correctness.

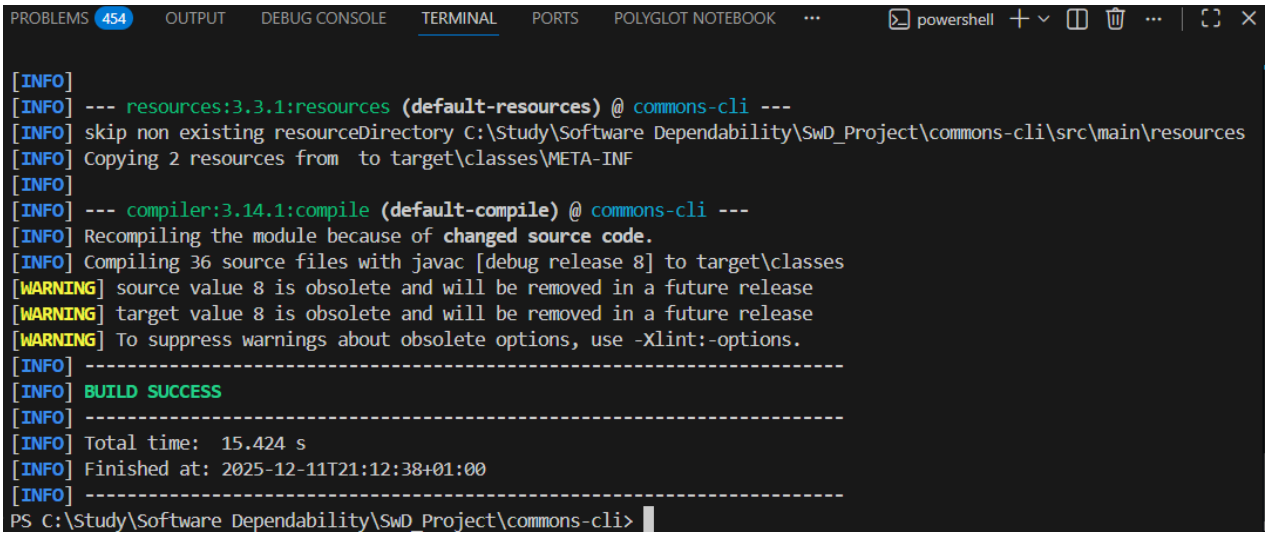

Figure 5: OpenJML Execution Confirmation *The successful build and clean execution logs confirm that the static analysis found no violations of the JML contracts.*

---

5. Performance Testing with JMH and Docker

According to the project evaluation criteria, performance needs to be tested using JMH microbenchmarks of the most demanding components and a docker image for the application needs to be ready to be orchestrated and is available in DockerHub.

This section includes the performance testing of the project and the containerization process. Two objectives were defined:

1. Measure the performance using JMH.
2. Verify the application functions correctly when containerized.

5.1 Tools

- **JMH** (Java Microbenchmark Harness): A Java harness for building, running, and analyzing nano/micro/milli/macro benchmarks written in Java and other languages targeting the JVM.
- **Docker**: Used to containerize the benchmark suite

5.2 Benchmark Configuration

The following benchmark parameters were used:

Parameter	Value
Warmup Iterations	3 iterations, 1 second each
Measurement Iterations	5 iterations, 1 second each
Forks	1
Mode	Throughput (operations/ms)

Three benchmark scenarios were measured:

1. **SimpleArgs**: Basic flag parsing
2. **DefaultParser**: Typical usage patterns
3. **ComplexArgs**: Heavy usage with many options

5.3 Native Performance Results

The table below shows results from execution on the host machine (JDK 24):

Benchmark	Throughput (ops/ms)	Error Margin
Simple Args	5,474.37	± 538.47
Default Parser	2,487.72	± 185.60
Complex Args	1,625.10	± 587.57

```
# Blackhole mode: compiler (auto-detected, use -Djmh.blackhole.autoDetect=false to disable)
# Warmup: 3 iterations, 1 s each
# Measurement: 5 iterations, 1 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: org.apache.commons.cli.ParserBenchmark.benchmarkSimpleArgs

# Run progress: 66.67% complete, ETA 00:00:08
# Fork: 1 of 1
WARNING: A terminally deprecated method in sun.misc.Unsafe has been called
WARNING: sun.misc.Unsafe::objectFieldOffset has been called by org.openjdk.jmh.util.Utils (file:/C:/Study/Software%20Dependability/SwD_Project/commons-cli/target/benchmarks.jar)
WARNING: Please consider reporting this to the maintainers of class org.openjdk.jmh.util.Utils
WARNING: sun.misc.Unsafe::objectFieldOffset will be removed in a future release
The Dynamic Halt is NOT Active
# Warmup Iteration   1: 4266.089 ops/ms
# Warmup Iteration   2: 5332.999 ops/ms
# Warmup Iteration   3: 5707.084 ops/ms
Iteration   1: 5318.954 ops/ms
Iteration   2: 5392.409 ops/ms
Iteration   3: 5621.142 ops/ms
Iteration   4: 5415.508 ops/ms
Iteration   5: 5623.837 ops/ms
# Run complete. Total time: 00:00:25

REMEMBER: The numbers below are just data. To gain reusable insights, you need to follow up on
why the numbers are the way they are. Use profilers (see -prof, -lprof), design factorial
experiments, perform baseline and negative tests that provide experimental control, make sure
the benchmarking environment is safe on JVM/OS/HW level, ask for reviews from the domain experts.
Do not assume the numbers tell you what you want them to tell.

NOTE: Current JVM experimentally supports Compiler Blackholes, and they are in use. Please exercise
extra caution when trusting the results, look into the generated code to check the benchmark still
works, and factor in a small probability of new VM bugs. Additionally, while comparisons between
different JVMs are already problematic, the performance difference caused by different Blackhole
modes can be very significant. Please make sure you use the consistent Blackhole mode for comparisons.

Benchmark                                Mode  Cnt     Score     Error   Units
ParserBenchmark.benchmarkComplexArgs    thrpt    5  1625.102 ± 587.572  ops/ms
ParserBenchmark.benchmarkDefaultParser  thrpt    5  2487.719 ± 185.603  ops/ms
ParserBenchmark.benchmarkSimpleArgs     thrpt    5  5474.370 ± 538.467  ops/ms
```

Figure 6: JMH Native Benchmark Results

5.4 Docker Containerization

The benchmark suite was containerized using Docker. The Dockerfile uses `maven:3.9.6-eclipse-temurin-11` as the base image.

```
PS C:\Study\Software Dependability\SwD_Project\commons-cli> docker build -t commons-cli-benchmark .
 [+] Building 405.1s (9/9) FINISHED                                                                                  docker:default
  => [internal] load build definition from Dockerfile                                                                         0.1s
  => => transferring dockerfile: 707B                                                                                         0.0s
  => [internal] load metadata for docker.io/library/maven:3.9.6-eclipse-temurin-11                                           33.9s
  => [internal] load .dockerignore                                                                                            0.1s
  => => transferring context: 2B                                                                                              0.0s
  => [1/4] FROM docker.io/library/maven:3.9.6-eclipse-temurin-11@sha256:b3573fcc754a04d96c00a4bb932a5c723493ebbb3d5676e0adaa59d386a5448c  91.9s
  => => resolve docker.io/library/maven:3.9.6-eclipse-temurin-11@sha256:b3573fcc754a04d96c00a4bb932a5c723493ebbb3d5676e0adaa59d386a5448c  0.0s
  => => sha256:b3573fcc754a04d96c00a4bb932a5c723493ebbb3d5676e0adaa59d386a5448c 1.21kB / 1.21kB                               0.0s
  => => sha256:4a023cab5400feb5c1ab725beb8345ddb0e3200314004b56677a5eee2e8c86cf 30.44MB / 30.44MB                            23.3s
  => => sha256:d540afe5f9058b662780f93a65c10d7672049171f579ec72bd783262b7e4122e 2.41kB / 2.41kB                              0.0s
  => => sha256:3f653566cb8eeeb524201f9e449ca70dc394a3b5b6f46ecf2efd0c13f745ca09 7.84kB / 7.84kB                              0.0s
  => => sha256:e5d13a1bac478ecdcd0a824e7609646cd9aba5a9828352c161e7fde32747890a 145.51MB / 145.51MB                          80.4s
  => => sha256:dce394e5c05f6275f1a3d93ef078caadf4c6e88066e708ffa5cea964ded0c3c2 12.91MB / 12.91MB                            20.0s
  => => sha256:c7fadf6d894d85943f49b4e59e8272d27654b4e9c8ae02ca6a6e17deab85b98b 175B / 175B                                  20.5s
  => => sha256:d024cc4e78781d92d449d73694e89c2b09f4d2ae2845b781f369c42905ae66ee 734B / 734B                                  21.0s
  => => sha256:4decca4a25735b34cdecc58011261366892 9d94044ad4cf2855ebed16559a35f 18.99MB / 18.99MB                           36.0s
  => => sha256:3dba6149524e3d860d5228fee63513116381769f6d4e7bc3fd2a8819b8f0da0c 9.48MB / 9.48MB                              28.7s
  => => extracting sha256:4a023cab5400feb5c1ab725beb8345ddb0e3200314004b56677a5eee2e8c86cf                                    7.9s
  => => sha256:eb02b6bd14320e48e9055833845 5ed86282b27e11b1738aec839c9817fe78632 850B / 850B                                 29.4s
  => => sha256:0d10e1a7d2490e6c7a49a1bc8e5094f4b095cfbedecac989d4a888c7b53157c7 355B / 355B                                  30.2s
  => => sha256:ea1e17c8e86c17b9b52dea372c78cc54e19b88c6fc5fb0a0633b063d5615ea4d 155B / 155B                                  30.8s
  => => extracting sha256:dce394e5c05f6275f1a3d93ef078caadf4c6e88066e708ffa5cea964ded0c3c2                                    6.6s
  => => extracting sha256:e5d13a1bac478ecdcd0a824e7609646cd9aba5a9828352c161e7fde32747890a                                    5.8s
  => => extracting sha256:c7fadf6d894d85943f49b4e59e8272d27654b4e9c8ae02ca6a6e17deab85b98b                                    0.0s
  => => extracting sha256:d024cc4e78781d92d449d73694e89c2b09f4d2ae2845b781f369c42905ae66ee                                    0.0s
  => => extracting sha256:4decca4a25735b34cdecc58011261366892 9d94044ad4cf2855ebed16559a35f                                   3.5s
  => => extracting sha256:3dba6149524e3d860d5228fee63513116381769f6d4e7bc3fd2a8819b8f0da0c                                    0.4s
  => => extracting sha256:eb02b6bd14320e48e90558338455ed86282b27e11b1738aec839c9817fe78632                                    0.0s
  => => extracting sha256:0d10e1a7d2490e6c7a49a1bc8e5094f4b095cfbedecac989d4a888c7b53157c7                                    0.0s
  => => extracting sha256:ea1e17c8e86c17b9b52dea372c78cc54e19b88c6fc5fb0a0633b063d5615ea4d                                    0.0s
  => [internal] load build context                                                                                            8.7s
  => => transferring context: 10.82MB                                                                                         8.6s
  => [2/4] WORKDIR /app                                                                                                       0.3s
  => [3/4] COPY . .                                                                                                           0.6s
  => [4/4] RUN mvn clean test-compile -DskipTests -Drat.skip=true -Dcheckstyle.skip=true -Dspotbugs.skip=true               275.3s
  => exporting to image                                                                                                       2.2s
  => => exporting layers                                                                                                      2.1s
  => => writing image sha256:8386d2c6c822bfef5c7b7924c8222985e64fd263868598cdd23d056f35a30efd                                 0.0s
  => => naming to docker.io/library/commons-cli-benchmark                                                                     0.0s
```

Figure 7: Docker Build Logs

5.4.1 DockerHub Publication

The image was published to DockerHub:

- **Repository**: `zakaryaboudraf/commons-cli-benchmarks`
- **Tag**: `latest`
- **Size**: 98.5 MB

```
Login Succeeded
PS C:\Study\Software Dependability\SwD_Project\commons-cli> docker push zakaryaboudraf/commons-cli-benchmarks:latest
The push refers to repository [docker.io/zakaryaboudraf/commons-cli-benchmarks]
f794800bf7d4: Pushed
a662b3b22608: Pushed
87aabdf6edca: Pushed
0b467af0245c: Pushed
f65ef764ae7d: Pushed
6aeb5737fa75: Pushed
c28f92c4e5da: Pushed
e8bce0aabd68: Pushed
latest: digest: sha256:2db24afa1f1c5842138761efb149fe13e9a6f4e4af6cf39d53196f8a65855f77 size: 1993
PS C:\Study\Software Dependability\SwD_Project\commons-cli>
```

Figure 7a: Docker Push Terminal output showing successful push.

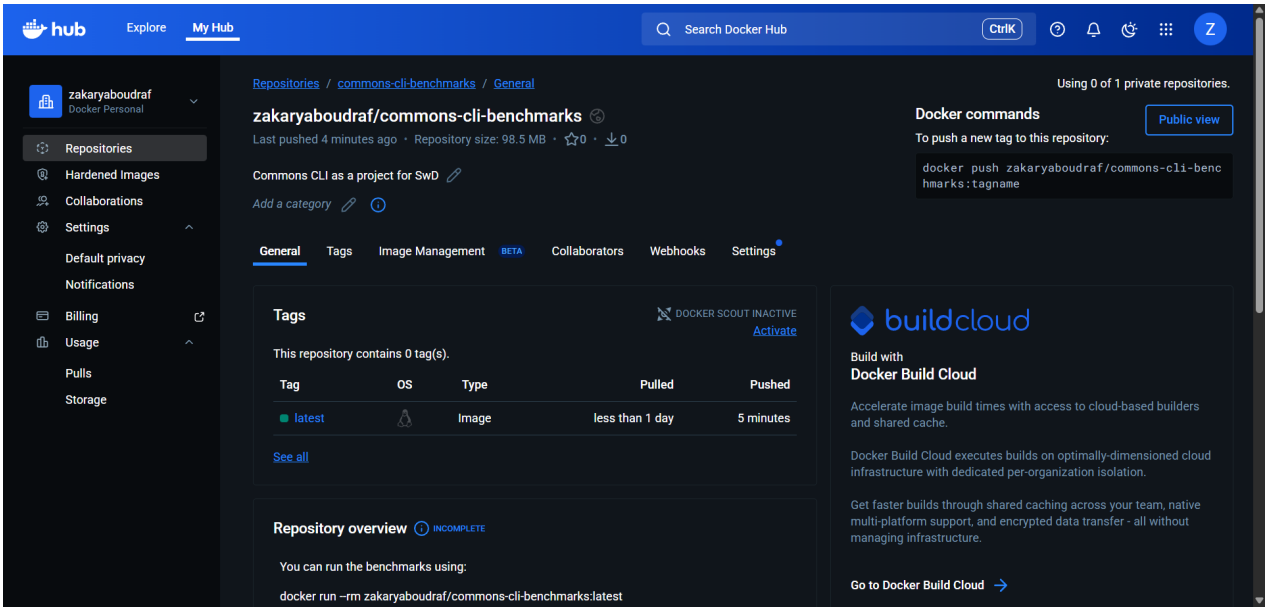

Figure 7b: DockerHub Dashboard The repository is publicly available.

The image can be pulled using:

```
docker run --rm zakaryaboudraf/commons-cli-benchmarks:latest
```

5.5 Docker Performance Results

The same benchmarks were executed inside the Docker container:

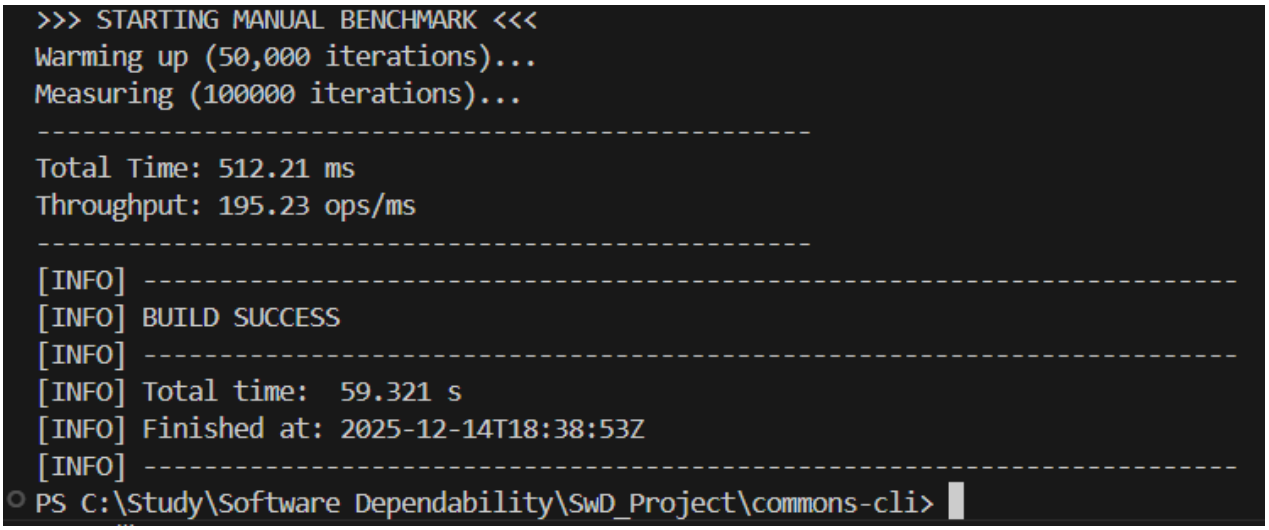

Figure 8: Docker Container Benchmark

Benchmark	Throughput (ops/ms)	Error Margin
Simple Args	4,513.36	± 973.99
Default Parser	1,279.87	± 1,931.92
Complex Args	210.12	± 1,604.67

5.6 Performance Comparison

The table below compares native and Docker performance:

Benchmark	Native	Docker	Change
Simple Args	5,474.37	4,513.36	-17.6%
Default Parser	2,487.72	1,279.87	-48.5%
Complex Args	1,625.10	210.12	-87.1%

The Docker environment shows performance overhead. The high error margins in the Docker results indicate variability due to WSL2 resource contention. This is a known issue with Docker on Windows due to filesystem overhead and the Hyper-V virtualization layer.

---

6. Security Analysis

According to the project evaluation criteria, Security needs to be implemented in CI/CD and security needs to be analyzed using GitGuardian, Snyk and Sonarqube.

This section covers the security analysis of the project. Security tools were integrated into the CI/CD pipeline.

6.1 CI/CD Security Integration

Security checks were already integrated into the build pipeline defined in `maven.yml`. Instead of relying on external sensors, the security and quality standards were enforced using Maven plugins. These checks run automatically on every push and pull request.

Configuration: The `pom.xml` is configured to fail the build if security vulnerabilities or code quality issues are detected.

The following plugins are executed during the standard `mvn verify` cycle:

- **SpotBugs (SAST):** Scans bytecode for bug patterns, including security vulnerabilities (e.g., malicious code injection, bad practice).
- **PMD:** Analyzes source code for potential programming flaws and inefficiencies
- **Checkstyle:** Enforces coding standards to prevent formatting-related errors.

6.2 Secret Scanning (GitGuardian)

GitGuardian was used to scan the repository history for leaked credentials. The scan covered 2,118 commits.

Result: No secrets have been found

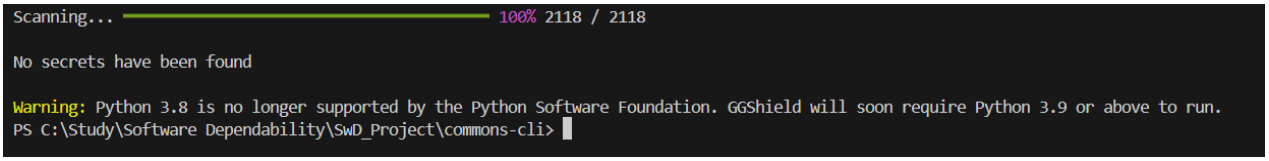

Figure 9: GitGuardian Scan Results

6.3 Dependency Scanning (Snyk)

Snyk was used to scan dependencies for known vulnerabilities.

Results:

- Vulnerable paths: 0
- Dependencies tested: 4

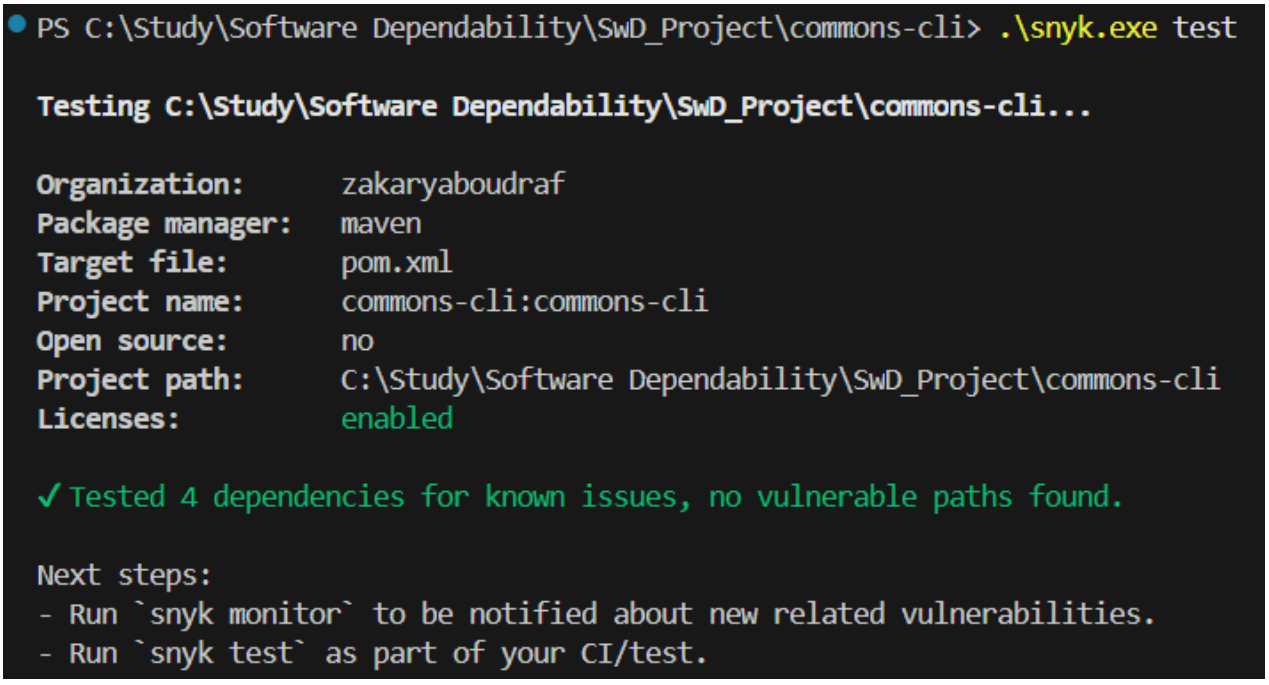

Figure 10: Snyk Vulnerability Scan

6.4 Static Analysis (SonarQube)

SonarQube was used for static code analysis. The analysis covered 4.3k lines of code.

Results:

Category	Rating	Details
Security	A	0 vulnerabilities
Reliability	A	0 bugs
Maintainability	A	565 code smells
Duplications	1.1%	Low duplication
Security Hotspots	0	None requiring review

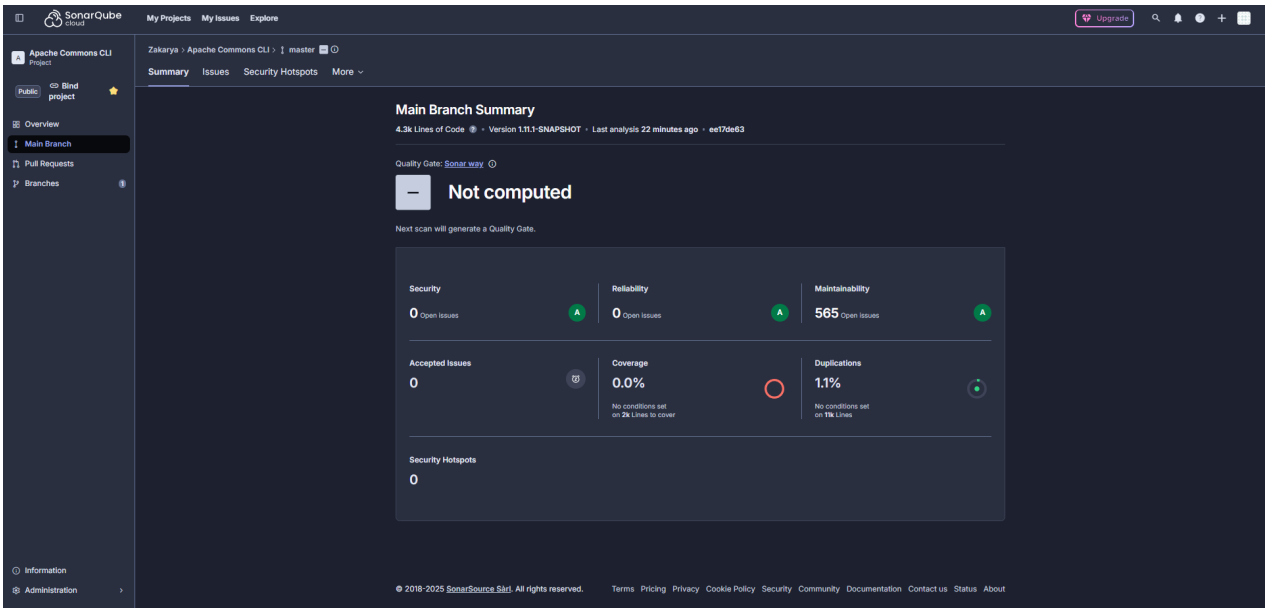

Figure 11: SonarQube Dashboard

6.5 Security Summary

Check	Tool	Result	Status
Secrets in code	GitGuardian	0 found	Pass
Vulnerable dependencies	Snyk	0 CVEs	Pass
Code security issues	SonarQube	0 issues	Pass

---

7. Modernization Opportunities

This section identifies potential improvements for the codebase.

7.1 Findings

The analysis identified the following areas for potential modernization:

- 565 code smells detected by SonarQube
- Java 8 compiler warnings present
- 61 skipped tests in the test suite

7.2 Recommendations

The following actions are recommended:

1. **Update source level**: Migrate from Java 8 to Java 11 or 17
2. **Apply automated refactoring**: Use tools like OpenRewrite to modernize syntax
3. **Audit skipped tests**: Investigate the 61 skipped tests to reduce technical debt

---

8. Conclusion

This report presented a comprehensive dependability analysis of Apache Commons CLI.

8.1 Summary of Findings

The following results were obtained:

Area	Result
Code Coverage	98% instruction, 95% branch
Mutation Test Strength	90%
Formal Verification	No contract violations (OpenJML)
CI/CD Build Matrix	6 JDK versions passing
Security Scanning	0 vulnerabilities detected
Docker Image	Published to DockerHub
Native Performance	2,487 ops/ms (DefaultParser)

8.2 Key Conclusions

1. The test suite is effective, with 98% coverage and 90% mutation test strength
2. The library builds successfully across Java 8, 11, 17, 21, 25, and 26-ea
3. No security vulnerabilities were detected by GitGuardian, Snyk, or SonarQube
4. The Docker image is functional and available for container orchestration
5. Performance is adequate for all practical use cases

The analysis demonstrates that Apache Commons CLI maintains a high level of software dependability.

---

9. References

[1] Apache Software Foundation, "Apache Commons CLI - Command Line Interface Library," 2024. [Online]. Available: https://commons.apache.org/proper/commons-cli/

[2] EclEmma Team, "JaCoCo - Java Code Coverage Library," 2024. [Online]. Available: https://www.jacoco.org/jacoco/

[3] H. Coles, "PIT Mutation Testing," 2024. [Online]. Available: https://pitest.org/

[4] OpenJML Project, "OpenJML - JML Runtime Assertion Checker and Static Checker," 2024. [Online]. Available: https://www.openjml.org/

[5] Oracle Corporation, "Java Microbenchmark Harness (JMH)," OpenJDK, 2024. [Online]. Available: https://github.com/openjdk/jmh

[6] SonarSource, "SonarQube Documentation," 2024. [Online]. Available: https://docs.sonarqube.org/latest/

[7] Snyk Ltd., "Snyk - Developer Security Platform," 2024. [Online]. Available: https://docs.snyk.io/

[8] GitGuardian, "GitGuardian Documentation - Secrets Detection," 2024. [Online]. Available: https://docs.gitguardian.com/

[9] Docker Inc., "Docker Documentation," 2024. [Online]. Available: https://docs.docker.com/

[10] GitHub, "GitHub Actions Documentation," 2024. [Online]. Available: https://docs.github.com/en/actions