Software Dependability Report

Luciano Bercini
University of Salerno
l.bercini@studenti.unisa.it

Abstract

This report presents an applied study on improving software dependability through the enhancement of an existing open-source project. The project forked is a component for reading and writing comma-separated value files. By forking a CSV-handling library, we demonstrate the integration of software-dependability practices. The aim is to assess and improve key dependability attributes, including maintainability, reliability, safety and testability, while maintaining functional correctness. The report documents the process, challenges, and results of adapting industry-grade techniques to a real-world codebase.

1 Project Setup

The starting point for this project was the open-source library `commons-csv`, maintained by the Apache Software Foundation. This library provides a simple and consistent API for reading and writing CSV files in Java.

To begin the practical work, we forked the official repository:

- **Original repository:** https://github.com/apache/commons-csv
- **The forked repository:** https://github.com/Luciano-Bercini-Unisa/commons-csv

The forked repository was then cloned locally, and initial efforts were made to ensure that the codebase could be built and tested cleanly. This step is critical to establish a stable baseline before introducing further changes or enhancements.

1.1 Clean Build Verification

A full clean build was performed using the standard Maven build process. The following outcomes were verified:

- The project compiled successfully without errors or warnings.
- All unit tests executed correctly and passed on the local development environment.

This confirms that the cloned project was stable and suitable for further dependability-enhancing modifications such as CI/CD pipeline integration, test coverage analysis, and code refactoring.

1.2 GitHub CI/CD Integration

To improve automation and ensure code quality, a basic Continuous Integration (CI) pipeline was configured using GitHub Actions. This was achieved by adding a `ci.yml` file to the repository's `.github/workflows` directory.

The pipeline is triggered on every push to the repository. It performs the following tasks automatically:

- Compiles the source code using Maven.
- Executes the test suite to verify correctness.

This setup helps to catch potential issues early in the development process and ensures that changes do not break the build or introduce test failures. The simplicity of the workflow also makes it easily maintainable and extensible for future enhancements, such as integration with code coverage or static analysis tools.

2 SonarCloud Software Quality Analysis

To enhance software quality, SonarCloud was integrated into the development process to perform static code analysis.

2.1 Categorization of SonarCloud Issues

The SonarCloud analysis of the forked `commons-csv` repository identified a total of 782 open issues. These were categorized both by severity and by issue type, allowing for a prioritized and targeted improvement process.

2.2 By Type

- **Code Smells (778)**: The vast majority of findings fall into this category. These include maintainability-related issues such as high method complexity, missing assertions, and duplicated logic. These were the primary focus of the refactoring efforts.
- **Bugs (4)**: A small number of issues were classified as functional bugs. These included logic flaws and exception handling problems that could potentially lead to runtime errors. All identified bugs were reviewed and addressed.
- **Vulnerabilities (0)**: SonarCloud did not detect security-related issues in this analysis.

2.3 By Severity

The following table summarizes the issues by severity level:

Severity Level	Number of Issues
Blocker	17
High	25
Medium	146
Low	72
Info	522

This data reflects that most of the issues were low criticality, focusing primarily on maintainability and code cleanliness.

2.4 Refactoring

Based on the analysis, the following improvements were implemented:

- **Test Case Refactoring:** Test methods containing a high number of assertions were split into smaller and more focused test cases, improving readability and fault location.

- **Test Case Complexity Reduction:** Several test cases were simplified to reduce complexity and improve understandability. This was a particularly difficult task due to the overcomplicated code in the Lexer class.
- **Brain Methods:** Methods identified as brain methods, due to excessive logic or nested control structures, were refactored to improve modularity and reduce cognitive load.
- **Missing Assertions:** Test cases that relied solely on exception throwing for validation were updated to include explicit assertions, improving test precision and clarity.
- **Lambda Refactoring:** Lambda expressions invoking multiple methods capable of throwing the same checked exception were rewritten to disambiguate exception sources, reducing ambiguity and improving debuggability.
- **Deprecated API Usage:** Several usages of deprecated methods were identified and removed or replaced with recommended alternatives.

These changes addressed a number of test code smells, including:

- **Assertion Roulette:** Tests with multiple assertions lacking descriptive failure messages were broken down to clearly identify failure causes.
- **Eager Test:** Tests verifying multiple behaviors simultaneously were split into separate units for better isolation.
- **Indirect Testing:** Refactoring was applied so that each test directly evaluates the behavior of the unit under test, rather than relying on transitive effects.
- **General Fixture:** In cases where tests used setup data irrelevant to some assertions, setups were minimized to avoid unnecessary complexity.
- **Conditional Test Logic:** Excessive branching was reduced in the test code to ensure straightforward execution paths.

All the specific code changes are available and documented in the commit history of the GitHub repository at:

https://github.com/Luciano-Bercini-Unisa/commons-csv

These enhancements contribute to improved test maintainability, easier debugging, and increased confidence in the codebase's behavior. The following table summarizes the issues by severity level after the refactoring:

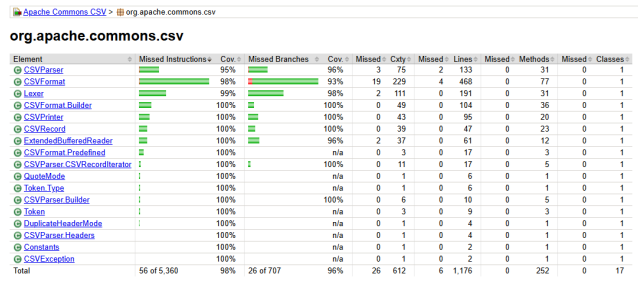
Severity Level	Number of Issues
Blocker	0
High	0
Medium	20
Low	2
Info	11

The main bulk of issues that were not fixed through refactoring, with the rationale for why they were skipped, are the following:

- **This block of commented-out lines of code should be removed:** These comments referred to psql lines of code that created the resources needed for the tests.
- **Complete the task associated to this TODO comment:** Expanding the project was not a task of interest.

3 Code Coverage Analysis with JaCoCo

To assess the quality and thoroughness of the test suite, we employed JaCoCo (Java Code Coverage) as our primary tool. JaCoCo was integrated into the build process to measure the extent to which the unit tests exercise the project's codebase. The coverage report was generated in HTML format, providing detailed insights into each class and method. The report can be found under target/site/jacoco/index.html.



Apache Commons CSV > org.apache.commons.csv

org.apache.commons.csv

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CSVParser		95%		96%	3	75	2	133	0	31	0	1
CSVFormat		98%		93%	19	229	4	468	0	77	0	1
Lexer		99%		98%	2	111	0	191	0	31	0	1
CSVFormat.Builder		100%		100%	0	49	0	104	0	36	0	1
CSVPrinter		100%		100%	0	43	0	95	0	20	0	1
CSVRecord		100%		100%	0	39	0	47	0	23	0	1
ExtendedBufferedReader		100%		96%	2	37	0	61	0	12	0	1
CSVFormat.Predefined		100%		n/a	0	3	0	17	0	3	0	1
CSVParser.CSVRecordIterator		100%		100%	0	11	0	17	0	5	0	1
QuoteMode		100%		n/a	0	1	0	6	0	1	0	1
Token.Type		100%		n/a	0	1	0	6	0	1	0	1
CSVParser.Builder		100%		100%	0	6	0	10	0	5	0	1
Token		100%		n/a	0	3	0	9	0	3	0	1
DuplicateHeaderMode		100%		n/a	0	1	0	4	0	1	0	1
CSVParser.Headers		100%		n/a	0	1	0	4	0	1	0	1
Constants		100%		n/a	0	1	0	2	0	1	0	1
CSVException		100%		n/a	0	1	0	2	0	1	0	1
Total	56 of 5,360	98%	26 of 707	96%	26	612	6	1,176	0	252	0	17

Figure 1: JaCoCo Code Coverage Analysis

The analysis focused on the `org.apache.commons.csv` package, which consists of 17 classes. JaCoCo reported a 98% instruction coverage and 96% branch coverage for the entire module, indicating a high level of test completeness. Of 5,362 bytecode instructions, only 56 were missed, and of 707 branches, 26 were not covered.

Significant classes such as CSVParser, CSVFormat, and Lexer show coverage above 95%. Even classes with complex logic such as CSVParser (95% instruction, 96% branch) and CSVFormat (98% instruction, 93% branch) were well tested.

The granular feedback from JaCoCo, including missed lines, methods, and classes, helped to identify areas for potential improvement. For instance, while most classes are fully covered, a few methods within CSVParser and CSVFormat remain partially untested. This provides actionable insights for enhancing test coverage further.

In conclusion, the JaCoCo report confirms that the Apache Commons CSV test suite is robust, with very few uncovered paths, affirming the reliability and maintainability of the codebase.

4 Mutation Testing through PiTest

To assess the quality and effectiveness of the existing test suite, mutation testing was performed using **PIT** (Pitest), a mutation testing framework for Java. Mutation testing introduces small changes (mutants) to the bytecode and checks whether the existing tests detect them (i.e., fail when they should). If a test fails due to a mutant, the mutant is said to be "killed"; if it passes, the mutant "survives", indicating a potential gap in the test coverage.

4.1 Execution and Setup

PIT was executed on the `org.apache.commons.csv` package using the Maven plugin. The analysis focused on measuring the following metrics:

- **Line Coverage**: The percentage of lines of code executed by the test suite.

- **Mutation Coverage**: The percentage of mutants killed by the test suite.
- **Test Strength**: The ratio of killed mutants to all mutants that were actually covered by the tests (i.e. a refined form of mutation coverage).

4.2 Results

Pit Test Coverage Report

Package Summary

org.apache.commons.csv

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
7	99%	1149/1155	95%	679/715	95%	679/712

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
CSVFormat.java	99%	587/591	93%	300/321	94%	300/318
CSVParser.java	99%	158/160	100%	98/98	100%	98/98
CSVPrinter.java	100%	96/96	100%	58/58	100%	58/58
CSVRecord.java	100%	47/47	95%	41/43	95%	41/43
ExtendedBufferedReader.java	100%	61/61	94%	48/51	94%	48/51
Lexer.java	100%	191/191	93%	132/142	93%	132/142
Token.java	100%	9/9	100%	2/2	100%	2/2

Report generated by PIT 1.15.2

Figure 2: PIT Test Coverage Report Summary

The PIT report showed high coverage and strength across the entire package:

- **Line Coverage:** 99% (1149/1155 lines)
- **Mutation Coverage:** 95% (679/715 mutants killed)
- **Test Strength:** 95% overall

All seven classes in the package achieved near-complete line and mutation coverage. In particular:

- `CSVPrinter`, `CSVRecord`, and `Token` reached 100% mutation coverage.
- `CSVFormat` and `Lexer` had slightly lower mutation coverage but still exceeded 90%.
- Test strength remained consistently above 93% across all classes.

4.3 Conclusion

These results indicate a robust and effective test suite capable of detecting the vast majority of potential faults. The high mutation coverage and test strength demonstrate strong test thoroughness and reliability. The PIT analysis reinforces the findings from JaCoCo and complements the structural testing with a deeper semantic validation.

5 Automated Test Generation with Randoop

To complement the manual and mutation-based testing strategies, the Randoop tool was used to automatically generate test cases. Randoop (Random Tester for Object-Oriented Programs) generates sequences of method calls that serve as unit tests for Java classes. It is particularly useful for identifying edge-case behaviors and improving regression coverage.

5.1 Execution and Setup

Randoop was configured to target three classes (specified in the file RandoopClassList.txt) within the `org.apache.commons.csv` package. The tool dynamically explored the public API of these classes and attempted to generate both regression and error-revealing test cases.

Key metrics from the execution:

- **Total public members analyzed:** 182
- **Test sequences generated:** 2,372
- **Regression tests generated:** 1,304
- **Error-revealing tests:** None found
- **Method executions:** 588,938 normal, 490 exceptional

5.2 Results and Analysis

Randoop successfully generated over 2,300 test sequences and extracted 1,304 regression tests. These tests were saved as JUnit test cases. These regression tests are valuable for confirming that code behavior remains consistent across future changes.

No error-revealing tests were generated, which suggests that no exceptions or observable failures occurred during execution that Randoop could exploit. However, during execution, 14 test failures were observed — mostly related to nondeterministic behavior or reliance on external state.

5.3 Flaky Tests

Randoop identified multiple **flaky tests**, i.e. tests that produce inconsistent results across runs. These are usually caused by:

- Use of non-local state (e.g., reading from files, I/O streams)
- Nondeterministic method behavior
- Dependency on external resources

The following methods were flagged as likely sources of nondeterminism:

- `CSVFormat.parse(Reader)`
- `CSVParser.parse(Reader, CSVFormat)`
- `CSVFormat.getHeaderComments()`
- `CSVFormat.isNullStringSet()`
- `CSVFormat.getDelimiterString()`

More examples involved file and stream handling from Apache Commons IO's builder classes.

5.4 Impact of Randoop on Code Coverage

After integrating tests automatically generated by Randoop, we observed a modest but measurable improvement in test coverage, specifically for the `CSVFormat` class — the main focus of testing in this analysis.

After incorporating the Randoop regression tests and ensuring they were correctly executed by PIT, the coverage slightly increased in mutation test strength, indicating that some additional paths had been exercised that were previously untouched.

The following is the JaCoCo analysis of CSVFormat after integrating Randoop tests:

CSVFormat	99%		95%		15	229	2	468	0	77	0	1

Figure 3: JaCoCo Coverage Report After Randoop

The following is the PIT report of CSVFormat after integrating Randoop tests:

Name	Line Coverage		Mutation Coverage		Test Strength	
CSVFormat.java	99%	589/591	94%	303/321	95%	303/320

Figure 4: PIT Mutation Coverage Report After Randoop

This small increase is expected — most branches and behaviors in `CSVFormat` were already covered by manually written tests. However, Randoop helped to uncover a few new combinations that further strengthened the test suite.

5.5 Conclusion

Randoop was effective in generating a wide variety of regression tests for the target classes, offering an automated method to enhance test coverage. However, due to the presence of flaky tests, care must be taken when incorporating these tests into the main suite. Randoop's documentation provides guidance on avoiding nondeterministic test generation, such as by restricting the set of methods tested or mocking state-dependent behavior.

6 Performance Evaluation via Java Microbenchmark Harness

To evaluate the parsing performance of different CSV libraries in Java, we employed the Java Microbenchmark Harness (JMH). We configured and executed a series of benchmarks comparing several popular CSV parsers, including Apache Commons CSV, JavaCSV, OpenCSV, and SuperCSV. The benchmarks were designed to simulate realistic usage by reading a preloaded dataset (to eliminate I/O interference) and counting the numbers of record within it. We've measured the average parsing time for each library and approach.

6.1 Setup

The benchmarking class `CSVBenchmark` was created under the test sources, and the JMH was invoked. We used the following JMH configuration:

- **Mode:** AverageTime
- **Forks:** 1
- **Warmup Iterations:** 5
- **Measurement Iterations:** 5
- **Threads:** 1
- **Heap Settings:** `-Xms1024M -Xmx1024M`

All parsers processed the same GZIP-compressed CSV file (`worldcitiespop.txt.gz`).

6.2 Results

The average parsing times are reported in the following table:

Table 1: JMH Benchmark results on the `worldcitiespop` dataset.

Benchmark	Iterations	Time (ms/op)	± Error
Commons CSV	10	898.44	19.04
Java CSV	10	811.34	10.48
Open CSV	10	892.52	9.51
Super CSV	10	796.94	21.46
read (lines)	10	115.30	7.26
scan	10	1355.49	1.88
split	10	492.19	18.17

6.3 Discussion

Super CSV yielded the best performance among the dedicated CSV libraries, followed closely by Java CSV. Apache Commons CSV and Open CSV showed similar performance. As expected, low-level approaches such as line-by-line reading and splitting without full CSV compliance performed significantly faster but lack feature support like multi-line fields.

7 Security Analysis

The security has been analyzed using two tools: OWASP Dependency-Check and FindSecBugs.

7.1 OWASP Dependency-Check

OWASP Dependency-Check is a software composition analysis tool that scans project dependencies to detect publicly disclosed vulnerabilities listed in sources such as the National Vulnerability Database (NVD). It identifies components via Common Platform Enumeration (CPE) and evaluates them against known CVEs (Common Vulnerabilities and Exposures).

The scan was performed on the Apache Commons CSV project using version 6.5.3 of the Dependency-Check tool. The analysis flagged two third-party libraries with known vulnerabilities:

- `commons-codec-1.17.1.jar`
  Identified with CVE-2021-37533, this vulnerability allows a malicious server to redirect the FTP client to a different host, potentially exposing private network details. Severity: **Medium (6.5)**.
- `commons-io-2.18.0.jar`
  Also associated with CVE-2021-37533, due to a CPE misidentification with Apache Commons Net. Severity: **Medium (6.5)**.

7.2 FindSecBugs

FindSecBugs is a static analysis tool for identifying security issues in Java bytecode. It is an extension of SpotBugs focused specifically on security vulnerabilities.

The analysis was performed on the compiled output of the Commons CSV project. No security vulnerabilities were detected in the codebase. The analysis covered 20 classes across a single package (`org.apache.commons.csv`) and confirmed the absence of issues.

7.3 Conclusion

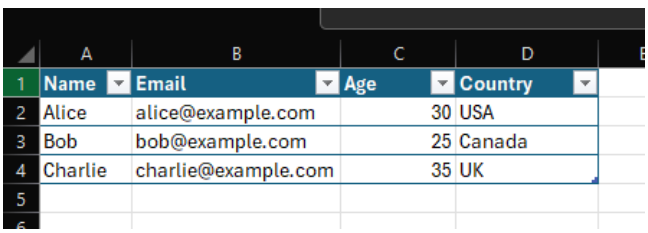
The project passed the static analysis performed by FindSecBugs with no findings. However, OWASP Dependency-Check reported two medium-severity vulnerabilities inherited from dependencies. The team should evaluate whether an upgrade or exclusion is needed and confirm the validity of the flagged issues.

8 Web Application Integration, Dockerization and Continuous Delivery

To fulfill the deployment and orchestration requirements, the Apache Commons CSV project was extended with Docker support and a web-based demonstration application. A custom Dockerfile was written to build a lightweight Docker image containing the compiled JAR.

```
FROM openjdk:8-jre-alpine
COPY target/commons-csv-1.13.0-SNAPSHOT.jar /app/app.jar
WORKDIR /app
ENTRYPOINT ["java", "-jar", "app.jar"]
```

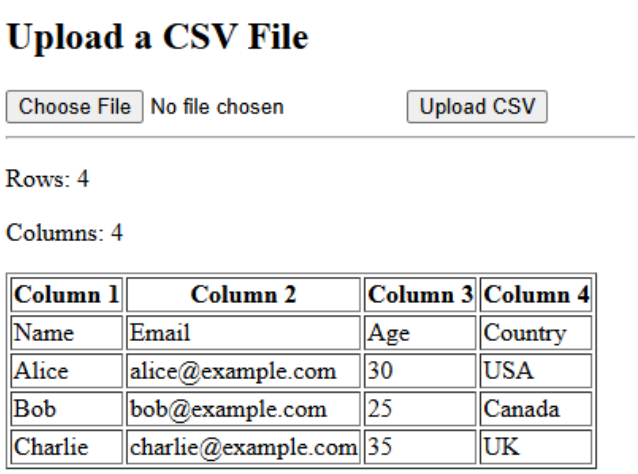
This Docker image encapsulates a minimal Javalin web application designed to demonstrate the functionality of the Commons CSV library. The app exposes an HTTP interface where users can upload a CSV file from their local machine. Upon upload, the file is parsed using Apache Commons CSV and the contents are displayed in the browser. A sample CSV might look like this:



	A	B	C	D
1	Name	Email	Age	Country
2	Alice	alice@example.com	30	USA
3	Bob	bob@example.com	25	Canada
4	Charlie	charlie@example.com	35	UK
5				

Figure 5: Sample CSV file.

Once uploaded, the output is shown:



Upload a CSV File

Choose File No file chosen Upload CSV

Rows: 4

Columns: 4

Column 1	Column 2	Column 3	Column 4
Name	Email	Age	Country
Alice	alice@example.com	30	USA
Bob	bob@example.com	25	Canada
Charlie	charlie@example.com	35	UK

Figure 6: Javalin App parsing the sample CSV file.

Internally, the file is parsed using the following:

```
CSVParser parser = CSVFormat.DEFAULT.parse(reader);
List<CSVRecord> csvRecords = parser.getRecords();
```

To automate the build and delivery of the Docker image, a GitHub Actions workflow named `docker.yml` was created. This workflow is triggered on every push to the `master` branch and performs the following tasks:

- Sets up a Java 8 environment.
- Builds the Maven project into a JAR.
- Logs in to DockerHub using secrets configured in the GitHub repository.
- Builds the Docker image.
- Pushes the image to DockerHub under the tag `latest`.

This setup ensures both Continuous Integration and Continuous Delivery:

- **CI**: The code is automatically compiled and tested on each push.
- **CD**: A new Docker image is automatically published to DockerHub, making the updated version immediately deployable.

Once built, the container can be run locally using: docker run -p 7070:7070 lucianoberciniunisa/commons-csv:latest

This launches the web server on `http://localhost:7070`, where the user can interact with the CSV parser without requiring any Java installation or manual setup. This complete pipeline—from source control to deployable image—demonstrates a full DevOps loop for a library project extended with a simple web frontend.