Apache Commons CSV

Francesco Paolo D'Antuono matr. 0522501767
Ivan Capobianco matr. 0522501694

ACM Reference Format:
Francesco Paolo D'Antuono matr. 0522501767 and Ivan Capobianco matr. 0522501694. 2024. Apache Commons CSV. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Apache Commons CSV library provides a simple interface for reading and writing CSV (Comma Separated Values) files of various types. This library makes it easier for Java Developers to work with CSV Data. It is available on GitHub at the following link: https://github.com/apache/commons-csv.

Apache Commons CSV key features are:

- **Parsing CSV Files**: The library offers a collection of methods for parsing CSV files into Java objects.
- **Writing CSV Files**: Apache Commons CSV also allows you to write data from Java objects into a specified CSV format. In this way you can export data from a Java application to a CSV file.
- **Flexible Configuration**: This library allows you to configure its behavior using configuration objects that let you specify custom delimiters, quoters, handling of empty lines, and more.
- **Support for CSV RFC 4180**: The library follows the RFC 4180 standard for CSV format.
- **Unicode Support**: Handling Unicode correctly is crucial for being able to work with multilingual data.
- **Open Source and Widely Used**: Apache Commons CSV is an open source project supported by the Apache Community. Also many examples and tutorials are available.

The Commons CSV project size is 37300 LOC (lines of code) with 46 classes. The repository is organized in a single module for implementing the CSVParser and two modules for testing classes.

In the project are also provided configuration files to build, compile, test and package the project with GitHub Actions.

The GitHub repository that we created and includes our modifications to the Apache Commons CSV library is availabe at the following link: https://github.com/CpDant/commons-csv.

2 SOFTWARE QUALITY ANALYSIS

We conducted a software quality analysis by scanning the project's GitHub repository with SonarCloud tool.

To make it work, we basically set the CI/CD with GitHub actions so that when particular action in GitHub is executed (like Push, Pull Requests and so on), GitHub performs an automatic building of the project.

Our analysis uncovered 6 bugs and 658 instances of code smells in the project. These bugs were classified as 3 critical and 3 major problems and they all referred to the reliability of the code.

One of the major issues identified, was the possibility of the method "getHeaderMapRaw()" in the class "CSVRecord.java" to throw a "NullPointerException" when invoked. This happened because its return value could be null and a reference to null should never be dereferenced/accessed. At best, such an exception will cause abrupt program termination. At worst, it could expose debugging information that would be useful to an attacker, or it could allow an attacker to bypass security measures. This specific problem was found twice in the project. We analyzed this issue and we discovered that this was a false positive because the issue was managed by another method that prevents this code fragment from returning null.

Another major issue identified was in the class "ExtendedBufferedReader.java". This issue referred to the value returned by the method "read()" in "readLine()" not being used or stored. By analyzing the class and the specific method, we realized that this use of the method was intentional and did not cause any problem. Also in this case this was a false positive because the method "readLine()" had to call the method "read()" just to continue the reading of the file and was not interested in its return value.

Critical issues were all of the same type and they were found in the class "CSVParserTest.java". In some test methods, multiple lambda expressions were chained together. When verifying that code raises an exception, a good practice is to avoid having multiple method calls inside the tested code, to be explicit about what is exactly tested. When two of the methods can raise the same checked exception, not respecting this good practice is a bug, since it is not possible to know what is really tested. So we made sure that only one method can raise the expected checked exception in the tested code. Indeed, here the analyses were correct and we fixed this bug by unchaining the lambda expression calls. We noticed that in this case, the same type of code had not been used, which had been used in the other test cases. So we rewrote the test case to follow the other test cases templates.

We analyzed some of the 658 code smells with a rationale that included only the most important that concerned the maintainability of the code. In one of these, SonarCloud expected to find in a method of a test class "PerformanceTest.java" (/perf package) at least one assertion. A test case without assertions ensures only that no exceptions are thrown. Beyond basic runnability, it ensures nothing about the behavior of the code under test. In this case, it was a false positive because the purpose of the test case was only to execute a Performance test, so there are no assertions in it, but

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the output showed us the time that the method took to do its tasks, so it works as intended.

The same type of issue was found in "JiraCsv213Test.java" and "JiraCsv264Test.java". These were also false positives because in these test cases there are no assertions, but these are specific bugs that are resolved from Apache, so this had to remain unchanged.

Also in classes "CSVRecordTest.java", "CSVParserTest.java" and "CSVFormatTest.java" SonarCloud found this type of issue, but, in this case, it wasn't a false positive and to resolve it, we added some assertions to control some things in these test cases. We noticed that there is the possibility that they forgot to complete the tests because the same type of test is done before them, but with a different target.

Another issue was found in class "PerformanceTest.java" (/csv package). This was a false positive because it isn't a test class with @Test annotation, but it's a class that is tested by running a command line command. So there is no need to add some tests in it.

In classes "Lexer.java", "ExtendedBufferedReader.java", "CSVParser.java" and "CSVFormat.java", the issues that were found by SonarCloud concerned in a too high Cognitive Complexity. Cognitive Complexity is a measure of how hard it is to understand the control flow of a unit of code. Code with high cognitive complexity is hard to read, understand, test, and modify. Most of the issues causing high cognitive complexity were due to have deeply nested code, resulting in big methods in which the control flow was not easily understandable. In these cases we had to refactor the code and make it simpler by extracting complex conditions in new methods.

In the class "CSVFormat.java" SonarCloud found 10 code smells regarding the naming scheme used for defining some constant values. SonarCloud suggested that the names should be defined to comply with a specific regular expression. Following a naming convention is always a good practice, and allows for better understanding of the code. In this case we decided not to change the names of these constant values because these are parts of the API exposed by the library to the end user. Indeed, modifying these names would constitute a breaking change in the API and should be better discussed and decided upon by the library's main contributors.

A large number of code smells classified as low severity by SonarCloud were about the use of the "public" modifier for test classes and methods. We solved these issues by removing the public modifier.

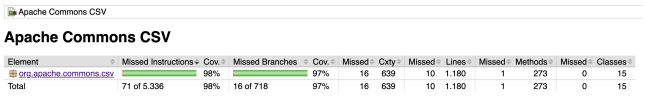
In the class "CSVPrinterTest.java" we found test cases with the same name, except for a sequential number at the end that were testing the same method but with different inputs. We rewrote these tests in a single test, using the parameterized test features of JUnit 5.

At the end of the software quality analysis process, we solved all of the 6 bugs and also 538 code smells.

3 CODE COVERAGE ANALYSIS

Code coverage offers valuable insights into how well testing is performed by gauging the proportion of the source code that is executed.

In our project, code coverage has been analyzed using the tool JaCoCo that produces comprehensive reports that aid developers in pinpointing areas that need additional testing. As shown in fig. 1, the code coverage for the only package in the library is 98%.

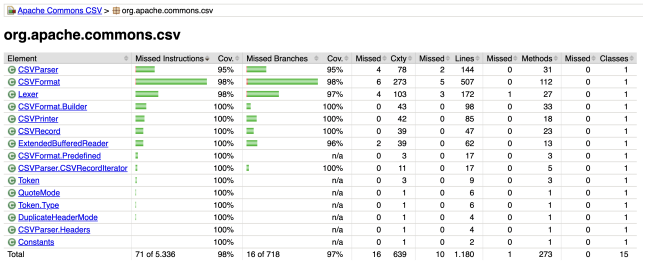

Apache Commons CSV

Apache Commons CSV

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.csv		98%		97%	16	639	10	1,180	1	273	0	15
Total	71 of 5,336	98%	16 of 718	97%	16	639	10	1,180	1	273	0	15

Figure 1: JaCoCo report - package coverage

Whereas, in fig. 2 we can notice the code coverage for each class in the package. The coverage for each class is very high so we can say that there are enough tests for the classes of this project.

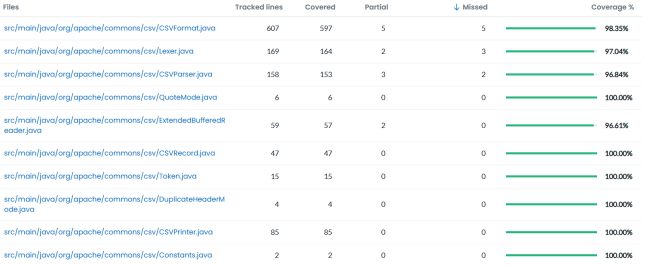

Apache Commons CSV > org.apache.commons.csv

org.apache.commons.csv

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CSVParser		95%		95%	4	78	2	144	0	31	0	1
CSVFormat		98%		98%	6	273	5	507	0	112	0	1
Lexer		98%		97%	4	103	3	172	1	27	0	1
CSVFormat.Builder		100%		100%	0	43	0	98	0	33	0	1
CSVPrinter		100%		100%	0	42	0	85	0	18	0	1
CSVRecord		100%		100%	0	39	0	47	0	23	0	1
ExtendedBufferedReader		100%		96%	2	39	0	62	0	13	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
Token		100%		n/a	0	3	0	9	0	3	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
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
Total	71 of 5,336	98%	16 of 718	97%	16	639	10	1,180	1	273	0	15

Figure 2: JaCoCo report - classes coverage

Another important tool for code coverage that we used for our project is Codecov. This tool helped us to determine and visualize in a better way the coverage of our classes/package and where the code isn't covered at all or partially covered. It gave us this type of information everytime we built our project in CI/CD. So, when we performed a push to the repository on GitHub, Codecov computed the code coverage of our project. In fig. 3 there is an example of what can be seen from the interface provided by Codecov.



Files	Tracked lines	Covered	Partial	↓ Missed	Coverage %
src/main/java/org/apache/commons/csv/CSVFormat.java	607	597	5	5	98.35%
src/main/java/org/apache/commons/csv/Lexer.java	169	164	2	3	97.04%
src/main/java/org/apache/commons/csv/CSVParser.java	158	153	3	2	96.84%
src/main/java/org/apache/commons/csv/QuoteMode.java	6	6	0	0	100.00%
src/main/java/org/apache/commons/csv/ExtendedBufferedReader.java	59	57	2	0	96.61%
src/main/java/org/apache/commons/csv/CSVRecord.java	47	47	0	0	100.00%
src/main/java/org/apache/commons/csv/Token.java	15	15	0	0	100.00%
src/main/java/org/apache/commons/csv/DuplicateHeaderMode.java	4	4	0	0	100.00%
src/main/java/org/apache/commons/csv/CSVPrinter.java	85	85	0	0	100.00%
src/main/java/org/apache/commons/csv/Constants.java	2	2	0	0	100.00%

Figure 3: Codecov report - classes coverage

4 MUTATION TESTING

To conduct the Mutation Testing we used PiTest which is a state of the art mutation testing system. PiTest allowed us to specify which mutators to use to perform the testing. During our test we used the default configuration which means all the stable mutators were used. The results we obtained are described in fig. 4 and fig. 5.

During the execution of the mutation tests, 10 tests resulted in a time-out. By analyzing the report produced by PiTest, we were able to determine that all of these 10 tests timed-out because the specific mutation applied created an infinite loop. It is worth mentioning though that PiTest does not provide any way to know

which specific test case resulted in a time-out. For this reason it is not always straightforward to determine if a time-out occurred because of an infinite loop or for other reasons. PiTest allows us to configure the time-out factor and the time-out constant, but incrementing these values would result in a longer execution time for the mutation testing, while still not providing a clear indication for the reason of the time-out.

We configured PiTest to use 8 threads instead of the default 1. This reduced the execution time of the test by 90% on average. We chose 8 because it was the value that best fitted our local machines on which we conducted the tests.

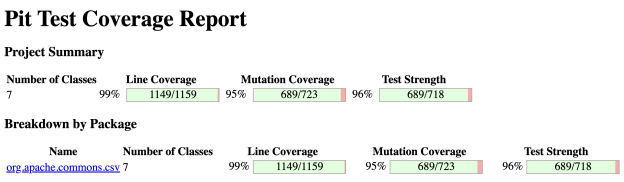


Figure 4: PiTest report - package coverage

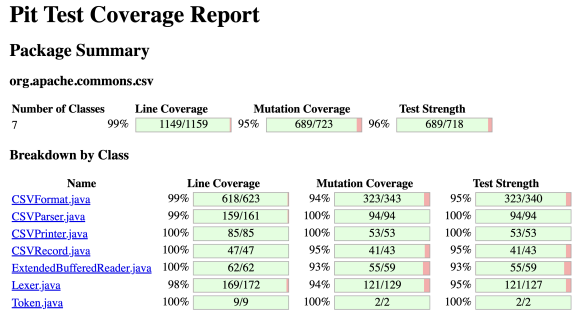


Figure 5: PiTest report - classes coverage

5 PERFORMANCE TESTING

For this type of testing, we used JMH (Java MicroBenchmark Harness) which is an annotation-based framework that enables control over the execution of test cases. In our project, we noticed that this framework was already implemented, together with another important tool that is CSV Tools. The aforementioned tool provides utility classes for simply reading from, and writing to CSV files, hence, it is used for simulating the behavior of our code on a temporary CSV.

To make it work, we had to change the location of the dependencies and the plugins for benchmark in the pom.xml, because they were in a particular place which made them not visible for the maven compilation.

In the first execution of our benchmark, we tested the average time of execution of each test case, with the annotation @BenchmarkMode(Mode.AverageTime), and we used only one @Fork because it was enough for our type of test cases.

We noticed that it took too many iterations than needed to do the tests in a way that the results would be stable, so we reduced the number of @Warmup and @Measurement iterations to reduce the time of execution of our benchmark.

In the second execution we had the final results (in ms/ops) that we put in table 1

Benchmark	Avg Time	Min Time	Max Time
parseCommonsCSV	1523.675	1337.792	1765.603
parseGenJavaCSV	1225.577	1124.541	1295.039
parseJavaCSV	1234.754	1073.294	1498.246
parseSkifeCSV	1462.125	1307.927	1628.038
parseSuperCSV	1381.573	1293.049	1526.181
read	157.744	151.640	166.647
scan	1140.819	1012.635	1293.638
split	750.084	688.006	851.519

Table 1: Benchmark Results

the results were very satisfactory so we left the classes as they were.

6 AUTOMATIC TEST CASE GENERATION

To automatically produce test cases, we used two important tools: EvoSuite and Randoop. After we used these tools we had to compile our project with this particular command "mvn package -Drat.skip=true" because these tools generated some classes that weren't under the src root, so we were forced to skip it.

6.1 EvoSuite

Evosuite was used for implementing basic unit tests using evolutionary algorithms. To set it up, we followed Emanuele Iannone's instructions on this GitHub repository https://github.com/emaiannone /tools-tutorial/tree/master/evosuite, because we couldn't make it work with Maven. So, we used the jar file "evosuite-1.2.0.jar" in our project and we had to run some commands on the command-line (we recommend using Ubuntu), which we found in the GitHub repository that we mentioned before. In particular, the first command that we used is "java -jar libraries/evosuite-1.2.0.jar -class org.apache.commons.csv.ExtendedBufferedReader -projectCP target/classes" and we had to use it with Java 8 (with other versions it did not work, the command for switching Java versions that we used is the following "sudo update-alternatives –config java").

After this, a test suite for ExtendedBufferedReader.java class (which is an extension of BufferedReader.java that overrides some methods of his father) was generated with default configurations and 32 test cases. We tested this class, because we noticed that it didn't have many important tests but at the same time it was a very important class for our project.

This test suite wasn't under the src root folder, so we had to compile it manually with this command "javac $(find ./evosuite-tests -name "*.java") -cp target/classes: libraries/evosuite-1.2.0.jar:target /dependency/junit-jupiter-5.10.0.jar:target/dependency /hamcrest-2.2.jar" but before this, we had to switch the Java version to Java 17 (with Java 8 it didn't work). After this command, we had the .class files that we needed in order to run our test suite.

Finally, we ran our test suite with this command "java –add-opens java.base/java.net=ALL-UNNAMED –add-opens java.desktop /java.awt=ALL-UNNAMED -cp evosuite-tests:target/ classes: libraries /evosuite-1.2.0.jar: target/dependency/ junit-platform-consol e-standalone-1.9.2.jar org.junit.platform.console.ConsoleLauncher –scan-class-path" and all the tests passed. Note that the command attributes "–add-opens java.base/java.net=ALL-UNNAMED" and "–add-opens java.desktop/java.awt=ALL-UNNAMED" were used to skip the classes that weren't accessible by Evosuite.

6.2 Randoop

We also used Randoop to implement basic unit tests using random generation and for the same reasons as the Evosuite installation, we followed Emanuele Iannone's instructions from this GitHub repository https://github.com/emaiannone/tools-tutorial/tree/master/ randoop. So, we used the jar file "randoop-all-4.3.2.jar" and the first command line that we used is "java -cp libraries/randoop-all-4.3.2.jar :target/classes randoop.main.Main gentests –testclass=org.apache .commons.csv.CSVParser –time-limit=20 –junit-output-dir=randoop-tests" but, also in this case, we had to use Java 8.

After this, a class named regressionTest.java was generated with default configurations and 108 test cases. So we had our test suite for CSVParser.java, but to run it, we first had to compile it (with Java 17), so we used this command "javac $(find randoop-tests -name "*.java") -cp target/classes :libraries/randoop-all-4.3.2.jar: target/dependency /junit-jupiter-5.10.0.jar: target/ dependency/hamcr est-2.2.jar".

Finally we could run the test suite with this command "java -cp randoop-tests: target/classes: libraries/randoop-all-4.3.2.jar: target/dependency /junit-platform-console-standalone-1.9.2.jar org. junit.platform.console.ConsoleLauncher –scan-class-path" and the tests were all green.

7 SOFTWARE VULNERABILITIES

To conduct the static security analysis of our chosen project, we used the tools SpotBugs and OWASP DC.

We installed SpotBugs as a plugin for the Intellij IDE and configured it to use the security rules. After performing the analysis, one security issue was discovered. This issue was related to the method 'parse' of the CSVParser.java class that takes an URL as a parameter. The possible security vulnerability is a Server-Side Request Forgery (SSRF). This issue occurs when a web server executes a request to a user supplied destination parameter that is not validated. Such vulnerability could allow an attacker to access internal services or to launch attacks from the web server on which the application is running. For example, a valid URL could use the 'file://' protocol to access local file systems and potentially other services. Some of the solutions proposed by SpotBugs were:

- Don't accept URL destinations from users
- White list URLs
- Validate that the beginning of the URL is part of a white list

Since Commons CSV is a library, we determined that none of the proposed solutions could be implemented directly in the project, since they would have a negative impact on the flexibility of the library. Indeed, we should not have any restrictions on the allowed URLs directly in the library, but decisions on what kind of URLs to accept should be taken by the users that use this library to implement their applications based on their needs.

To detect disclosed vulnerabilities of the dependencies of Commons CSV we used OWASP DC by installing it as a maven dependency. Commons CSV only has one dependency which is Commons IO and, by running the tool, no vulnerabilities were reported.

Another tool for reporting vulnerabilities of the dependencies used by Commons CSV is the Dependabot provided by GitHub. This bot can notify of known vulnerabilities in the dependencies of the project and automatically merge the necessary changes to update a dependency when a new version is available and is safe to do so.

8 CONCLUSIONS

The software quality analysis revealed various insights into the Apache Commons CSV library, including both positive aspects and areas for improvement.

The analysis of bugs and code smells highlighted a mix of valid issues and false positives. We successfully resolved the identified bugs and clarified the reasons why some bugs were false positives.

The code coverage analysis indicated a high level of coverage for both packages and individual classes. Even if a good code coverage does not necessarily indicate a good testing, it's still a very positive sign.

Mutation testing provided insights into the effectiveness of the test suite. Our decision to increase the number of threads for PiTest execution showcased a reduction in testing time.

Performance tests were already implemented in this project and they tested the most important classes, but we modified some parameters to improve the effectiveness. After that, our performance tests demonstrated satisfactory results for the library's benchmarked methods and also a reduction of the time they took to run.

Automatic test case generation tools EvoSuite and Randoop were employed to create additional test suites. These tools contributed to the testing strategy, generating tests for specific classes and methods. The validity of these test was confirmed by their successful execution.

For what concerns security, SpotBugs identified a potential SSRF vulnerability related to URL handling in the CSVParser class. We decided not to enforce URL restrictions directly in the library but recommended users consider security measures based on their specific use cases.

OWASP DC and Dependabot were utilized for dependency security analysis. No vulnerabilities were reported in the dependencies.

In conclusion, the Apache Commons CSV library emerges as a robust and well-maintained project, with a focus on code quality, security, and performance.