

Technical Demo: Code Coverage

A technical demonstration with Java Code Coverage on the open-source repository ZXing

Jonas Sävås, jonassav@kth.se

November 16, 2023

1 Introduction

This report is a technical demonstration for the master's thesis in order to receive a topic with a deadline of one week. The task consists of selecting an open-source Java repository with more than 500 commits and run the three coverage tools used in the paper "Coverage-Based Debloating for Java Bytecode" by César Soto-Valero et. al.[2] The coverage tools they use in the paper are JaCoCo, Jcov and Yajta. JaCoCo is a powerful tool with a lot of support with CI and build tool integration with for instance Gradle and Maven. Jcov is another powerful code coverage tool maintained by Oracle that requires more hands-on instrumentation of the classes to gather the metrics, and Yajta is a tool developed by the authors of the report to mainly gather third party library coverage in order to determine unused code within these libraries in an effort to debloat Java programs. The code coverage results will be gathered on the open-source project ZXing which is an android application to scan barcodes and retrieve the encoded data. ZXing fulfills the two requirements of being written in Java and has almost 4 000 commits. [3]

2 Background: Code Coverage

Code coverage is a metric in software development that reports what parts of the code have been covered during execution as well as more intricate details such as what execution traces were covered within for instance nested if-statements, often referred to as branch coverage. Code coverage mainly targets the testing phase of software development to provide insight into the robustness of the test suite by reporting what parts of the code was covered during its execution.

Since code coverage provides insight into what parts of a program is used during execution, the given metrics could also be used in order to remove unused parts of the code in an effort to reduce its memory footprint in a process called *debloating*. Removing unnessecary code is precisely what César Soto-Valero et al. does in their paper "Coverage-Based Debloating for Java Bytecode". In the paper they use both JaCoCo and Jcov to determine the code coverage for the application as well as their developed tool Yajta to determine the coverage of third party libraries in order to remove unused parts while maintaining full functionality.[2]

2.1 Tools for code coverage

There are various different tools to gather code coverage metrics for different programming languages. Such tools naturally differ in how they achieve such metrics but also in what types of coverage they can provide. In this case we are only investigating coverage tools for Java for which JaCoCo, Jcov and Clover are among the most popular.

JaCoCo and Jcov are coverage tools that Soto-Valero et.al mentions to be designed only to cover the project's code while the bytecode instrumentation leads to slightly incorrect coverage reports with for instance implicit methods among a few

other. They explain that this phenomenon is partly caused by compiler optimizations where JCov can handle compiler-generated methods. As previously mentioned, Yajta is a tool developed by Soto-Valero et al. in order to also gather the metrics for third party libraries which mainly serves the purpose of debloating the system under test.[2]

3 Method

3.1 Finding a suitable repository

There were two requirements that the open-source repository needed to fulfill in the task. The project needed to be written in Java and have more than 500 commits. Ensuring these requirements was quite simple and a multitude of repositories were investigated. Before choosing the final repository the code coverage tools JaCoCo, JCov and Yajta were examined to see if any benefits would come from the project specifications such as having a certain build tool. Yajta mentions that it can be depended upon in a Maven project and JaCoCo has a Maven plugin with various guides and tutorials. Although experience with Maven specifically was lacking a project with Maven as its build tool was chosen and the decision was made to run the coverage tools on the repository for the android barcode scanner ZXing.[3]

3.2 Building the instructed tools

Building the code coverage tools was something that was thought to be quite simple. The tools that required a JAR-file to be built was JCov and Yajta. The process started with Yajta which could be built with Maven defined by the pom.xml file. However, the build process induced some local test errors in each attempt which never completed the desired files and the resulting JAR-file could only be acquired by disabling the unit-tests in the build process.

As JCov was mentioned to be one of the most popular tools for Java the process for building the tool proved more cumbersome than initially anticipated. JCov needed the specific dependencies to be downloaded manually and it was not until the time limit for gathering the results was almost up until a detailed article on the matter was found.

The article "Calculating E2E no-JVM test code coverage with JCov" by Michał Wróbel provides detailed explanations for the difficulties of building JCov and outlines a short guide. Wróbel explains that the official build instructions is out of date which is a probable cause for the failed attempts when building it.[1] The article also provides a link to a repository with a prebuilt JCov JAR which was used in attempting to gather the coverage results.

3.3 Code Coverage: Running the tools

Running the tools started smoothly where the CI integration that JaCoCo provides with Maven made it very simple to gather the code coverage by instructing the

Maven build to generate a JaCoCo report. The process of gathering the results for JCoV and Yajta was significantly more involved. Yajta had instructions on how to include the dependency within a Maven project where many errors were encountered and further effort was instead done manually. Reading the tutorials in the Yajta repository and trying to instrument the classes went without any success. A similar turn of events happened with JCoV as the tutorials on their repository as well as the lack of information online was insufficient in gathering the coverage results for these two tools for a beginner in manual code instrumentation.

Knowing that only one out of the three instructed coverage reports were gathered within the required time-frame the process resorted to more simple methods of gathering code coverage to be compared with JaCoCo. The IntelliJ IDE has a built in code coverage tool that worked with minor configurations and instantly gathered a similar HTML report structure as JaCoCo previously had done. Efforts to gather other coverage reports with Clover went without success while Cobertura did generate both an HTML and XML report it did not correctly instrument the code which resulted in a very poor coverage metric that was far from correct.

With the time constraints and the immense trouble of both building and running the coverage tools the method resorted to comparing the coverage results from IntelliJ and JaCoCo in the upcoming sections.

4 Results

The complete results can be found in the GitHub repository in the results directory with this link where the coverage for all the classes with the different tools can be investigated by opening the respective index.html locally.

4.1 JaCoCo

The results from JaCoCo includes detailed information about which lines were covered and also includes branch coverage by default where inspecting the code reveals a yellow marked line to indicate missed branches. Figure 1 details a summary of the results where a total of 94% of the instructions were executed during the tests while 81% of the branches were covered.

ZXing Core

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed
com.google.zxing.datamatrix.encoder		89%		81%	165
com.google.zxing.oned		93%		83%	194
com.google.zxing.client.result		85%		68%	230
com.google.zxing		71%		46%	114
com.google.zxing.common		90%		80%	146
com.google.zxing.common.detector		59%		52%	51
com.google.zxing.pdf417.decoder		90%		84%	103
com.google.zxing.qrcode.encoder		93%		87%	68
com.google.zxing.datamatrix.decoder		90%		73%	86
com.google.zxing.pdf417.encoder		98%		84%	54
com.google.zxing.qrcode.decoder		96%		80%	56
com.google.zxing.multi		65%		71%	15
com.google.zxing.pdf417		99%		59%	28
com.google.zxing.pdf417.decoder.ec		86%		68%	35
com.google.zxing.aztec.encoder		94%		96%	17
com.google.zxing.datamatrix		83%		72%	24
com.google.zxing.maxicode.decoder		97%		80%	12
com.google.zxing.oned.rss.expanded		96%		93%	23
com.google.zxing.oned.rss		94%		88%	29
com.google.zxing.oned.rss.expanded.decoders		96%		87%	38
com.google.zxing.common.reedsolomon		94%		87%	19
com.google.zxing.qrcode		85%		77%	21
com.google.zxing.aztec		81%		67%	12
com.google.zxing.qrcode.detector		97%		93%	34
com.google.zxing.multi.qrcode.detector		92%		73%	20
com.google.zxing.multi.qrcode		84%		65%	8
com.google.zxing.aztec.decoder		97%		88%	12
com.google.zxing.aztec.detector		98%		94%	7
com.google.zxing.pdf417.detector		97%		95%	5
com.google.zxing.maxicode		94%		80%	4
com.google.zxing.datamatrix.detector		99%		98%	1
Total	7,976 of 134,269	94%	1,728 of 9,168	81%	1,631

Figure 1: This figure shows a snippet of the summary from the JaCoCo coverage report. There are a few missing metrics in the figure which the full view provides which are all related to instruction-, class- and branch coverage. We can see that there is a total of 94% instruction coverage and 81% branch coverage for a total of 270 measured classes.

4.2 IntelliJ Built in Code Coverage

The results from the built in code coverage tool within the IDE IntelliJ provides more simplistic results. Here the metrics that are available are line coverage which is also separated into class- and method coverage where we can see a total line coverage of 82.8% in figure 2.

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	88.3% (273/309)	83.2% (1663/1999)	82.8% (12819/15475)

Coverage Breakdown


Package 	Class, %	Method, %	Line, %
com.google.zxing	90.5% (19/21)	66.7% (80/120)	70.9% (319/450)
com.google.zxing.aztec	100% (3/3)	86.7% (13/15)	85.9% (79/92)
com.google.zxing.aztec.decoder	100% (4/4)	100% (18/18)	93% (198/213)
com.google.zxing.aztec.detector	100% (3/3)	93.1% (27/29)	97.8% (227/232)
com.google.zxing.aztec.encoder	100% (8/8)	87.3% (55/63)	96.3% (394/409)
com.google.zxing.client.j2se	0% (0/13)	0% (0/65)	0% (0/591)
com.google.zxing.client.result	100% (37/37)	85.8% (199/232)	85.2% (1090/1279)
com.google.zxing.common	100% (16/16)	92.3% (155/168)	90.6% (958/1057)
com.google.zxing.common.detector	66.7% (2/3)	71.4% (10/14)	61.5% (131/213)
com.google.zxing.common.reedsolomon	100% (5/5)	100% (35/35)	93.8% (274/292)
com.google.zxing.datamatrix	100% (2/2)	75% (9/12)	85.5% (124/145)
com.google.zxing.datamatrix.decoder	100% (9/9)	94.4% (51/54)	80.6% (482/598)
com.google.zxing.datamatrix.detector	100% (1/1)	100% (11/11)	99.4% (180/181)
com.google.zxing.datamatrix.encoder	100% (19/19)	92.1% (151/164)	86.4% (1001/1159)
com.google.zxing.maxicode	100% (1/1)	66.7% (4/6)	89.3% (25/28)
com.google.zxing.maxicode.decoder	100% (3/3)	88.2% (15/17)	82.9% (107/129)
com.google.zxing.multi	50% (1/2)	54.5% (6/11)	63.3% (62/98)
com.google.zxing.multi.qrcode	100% (2/2)	100% (7/7)	82.5% (52/63)
com.google.zxing.multi.qrcode.detector	100% (3/3)	100% (9/9)	88.4% (99/112)
com.google.zxing.oned	97% (32/33)	91.5% (162/177)	92.1% (1930/2095)
com.google.zxing.oned.rss	100% (6/6)	88.1% (37/42)	92.7% (342/369)
com.google.zxing.oned.rss.expanded	100% (4/4)	84.6% (33/39)	94.6% (424/448)

Figure 2: This figure shows a snippet of the summary of the coverage report provided by IntelliJ. There are less metrics in the report and only the class-, method- and line coverage are considered. We see that the line coverage for all classes is reported to be 82.8%.

4.3 Coverage Differences

While the results from IntelliJ provides quite simple metrics the detailed class reports do not indicate any particular differences in their covered lines between IntelliJ and JaCoCo upon further inspection. As previously mentioned, JaCoCo does provide more coverage details and while the covered and non-covered lines are clearly marked in both tools it does provide the extra insight of the number of uncovered branches which are marked in yellow.

```

208.     ResultPoint x = null;
209.     //go down left
210.     ◆ for (int i = 1; x == null && i < maxSize; i++) {
211.         x = getBlackPointOnSegment(right, up + i, right - i, up);
212.     }
213.
214.     ◆ if (x == null) {
215.         throw NotFoundException.getNotFountInstance();
216.     }
217.
218.     ResultPoint y = null;
219.     //go up left
220.     ◆ for (int i = 1; y == null && i < maxSize; i++) {
221.         y = getBlackPointOnSegment(right, down - i, right - i, down);
222.     }
223.
224.     ◆ if (y == null) {
225.         throw NotFoundException.getNotFountInstance();
226.     }
227.
228.     return centerEdges(y, z, x, t);
229.
230.     } else {
231.         throw NotFoundException.getNotFountInstance();
232.     }
233. }

```

Figure 3: This figure shows a snippet of the code coverage report by JaCoCo in the MinimalEncode.java class. The covered and uncovered lines are marked in green and red respectively while the branch indications are marked in yellow. Hovering over the yellow lines shows further information about the taken branches.

```

208     ResultPoint x = null;
209     //go down left
210     for (int i = 1; x == null && i < maxSize; i++) {
211         x = getBlackPointOnSegment(right, up + i, right - i, up);
212     }
213
214     if (x == null) {
215         throw NotFoundException.getNotFountInstance();
216     }
217
218     ResultPoint y = null;
219     //go up left
220     for (int i = 1; y == null && i < maxSize; i++) {
221         y = getBlackPointOnSegment(right, down - i, right - i, down);
222     }
223
224     if (y == null) {
225         throw NotFoundException.getNotFountInstance();
226     }
227
228     return centerEdges(y, z, x, t);
229
230     } else {
231         throw NotFoundException.getNotFountInstance();
232     }
233 }

```

Figure 4: This figure shows a snippet of the code coverage report by IntelliJ in the MinimalEncode.java class. The covered lines are marked in green while the uncovered lines are red which in turn shows the absence of branch coverage in the report.

Figure 3 and 4 shows the difference between JaCoCo and IntelliJ where the main difference is the absence of branch coverage in IntelliJ. In this case we can easily identify the execution but in the case of nested branches the branch coverage could prove significant when designing further test-cases or to indicate its robustness. Another detail about the coverage reports is that IntelliJ included more classes with a total of 309 whereas JaCoCo only reported coverage for 270 classes.

5 Discussion

The results indicated that JaCoCo and IntelliJ provided very similar information about the fully measured classes where both tools seemed to report the execution of the same instructions in all investigated cases. One of the main differences between the two was the number of classes that were reported by the tools. JaCoCo reported coverage for 270 classes while IntelliJ included 309 classes. While IntelliJ does not disclose how their coverage tool works the results point towards it being able to report coverage for classes that are called by the classes under test. On the other hand, JaCoCo generated more detailed coverage for the classes within the project which led to higher overall line/instruction coverage since IntelliJ reported untested classes that were called during testing.

Besides the inclusion of more classes, the biggest difference between the tools is JaCoCo's inclusion of branch coverage where IntelliJ simply reports if a line has been executed or not. An example of this can be seen in the result section in figures 3 and 4 which shows the same class in the respective report. In terms of usability in a live project, JaCoCo could prove to be much more beneficial in providing information about the robustness of the test suite as well as how to potentially improve it. However, IntelliJ's simpler report does come without its potential use-cases. While searching for suitable repositories the majority did not use any sort of coverage tool for their project. As there were a lot of issues when both trying to build and run the more manual tools like the instructed JCov or Yajta the simple tools like the built-in IntelliJ coverage could be very useful for providing a quick sanity check or basic measure of code coverage as it was very simple to use and a quick coverage report could be generated without any particular setup.

The time restriction to get a hold of a topic for the master's thesis is quite unfortunate in combination with studying full time which did not allow me to spend enough time on learning how to manually instrument the code with the two remaining tools. The results from Yajta and JCov would have been really interesting to investigate as they provide even more information and even include coverage of third-party libraries. However, the JCov official instructions lacked the necessary information for a beginner within the topic to both build and more crucially run the coverage tool within the time frame. However, having more time without interfering deadlines the issues would most definitely have been resolved.

References

- [1] Michał Wróbel. “Calculating E2E no-JVM test code coverage with JCov”. In: (2019). URL: <http://blog.mwrobel.eu/dynamic-java-e2e-code-coverage/>.
- [2] César Soto-Valero et al. “Coverage-Based Debloating for Java Bytecode”. In: *ACM Trans. Softw. Eng. Methodol.* 32.2 (Apr. 2023). ISSN: 1049-331X. DOI: 10.1145/3546948. URL: <https://doi.org/10.1145/3546948>.
- [3] “ZXing”. In: (2023). URL: <https://github.com/zxing/zxing>.