Report on Apache Commons Imaging Open Source repository

[Explanation of Apache Commons Imaging]

Oscar Arbman
Lovisa Sjöberg
Ziyuan Jia
Hugo Larsson Wilhelmsson
Erik Diep
Group 23

OpenSource name: Apache Commons Imaging
OpenSource URL: `https://commons.apache.org/proper/commons-imaging/`
OpenSource Github: `https://github.com/apache/commons-imaging`
Project Group's repository URL:
`https://github.com/lojjsan/dd2480-g23-commons-imaging/tree/master`
Date: February 21, 2025

1 Onboarding experience

The project has no dependencies except for a very low Java version (1.7 and later), when the current latest Java version is 25. If using Maven, adding the software was easy as a dependency in the configuration file, and nothing needs to be installed to use it. If not using Maven, one has to download the binary file, extract it, verify its integrity, and then add it to the project's classpath. The steps were clear but how to execute them (i.e. how to extract a .tar file and verify it against a SHA512-file) were not which added some complexity to the onboarding.

How to set it up using Maven was unclear as one had to build the dependency by themselves, really, using the keywords found on Maven's own site. Setting it up using the file only required some googling to extract the file and to verify it against the SHA512-file. The documentation regarding the library is very limited regarding both setup and java documentation.

Building with Maven gave no errors and was very easy to use. Some more configuration might have to be done if adding the .jar-file directly into the project instead of using Maven. Tests run without problem using Maven for the library. It compiles and tests without fault.

We plan to use the project, overall it has a coverage of 74% and contained many functions of high complexity with far from 100% branch coverage which we could analyse and improve.

2 Complexity

2.1 What are your results for five complex functions?

BasicCParser::nextToken()
The complexity analysis tool Clover measured the function **nextToken()**'s complexity as a value of 28. For the manual count: There are 29 decision points, 8 exit points, which results in a complexity of 23 (29-8+2). This difference might be because Clover measures complexity in another way by incrementing complexity by number of decision points and not deducting on number of exit points. But this is not really clear in their documentation unfortunately.

PcxWriter::writeImage()
For function PcxWriter::writeImage() has a cyclomatic complexity of 33 while possessing 113 lines of codes, this result also aligns with the report generated by Lizard and is confirmed by Ziyuan and Oscar.

TiffImageParser::getImageInfo()
The formula for calculating cyclomatic complexity for a function is $M = \pi - s + 2$. When calculating the cyclomatic complexity manually, I get π to be 39 and s to be 3. This results in the cyclomatic complexity: $M = 39 - 3 + 2 = 38$. This is very close to the cyclomatic complexity 39, which the complexity analysis tool, Clover, measured it to be. The reason for not being the exact same is unclear. Either have I calculated something wrong, or does the analysis tool calculate something as π that I do not. Otherwise, the results are clear both by the manually calculated cyclomatic complexity as well as the analysis tools calculation.

TiffImageParser::getRasterData()
The formula for calculating cyclomatic complexity for a function is $M = \pi - s + 2$. When calculating the cyclomatic complexity manually, I get π to be 33 and s to be 17. This results in the cyclomatic complexity: $M = 33 - 17 + 2 = 18$, while the jacoco report said 35, this makes me assume that the jacoco coverage report does not take into account exit points since, $33 + 2 = 35$.

TiffImageParser::getBufferedImage()
According to our calculation of the cyclomatic complexity for a function. When calculating the cyclomatic complexity manually, I get π to be 28 and s to be 12. This results in the cyclomatic complexity: $M = 25 - 12 + 2 = 15$, while the clover report said 28. This difference might be the same cause explained above in nextToken() section.

2.2 Are the functions just complex, or also long?

PcxWriter::writeImage()
This high CC function isn't very long in terms of LOC(line of code).

TiffImageParser::getImageInfo()
The function is not only complex, it is also long and contains many if statements and performs multiple operations. The function consists of 261 lines of code.

BasicCParser::nextToken()
The function contains many conditional statements, both inside and outside the for-loop. This added to its complexity with nestled conditioning and many different cases. Some cases handled string building operations, others exceptions and some performed no operations. Although there are many instances, almost half of the conditional statements, that if evaluated to true results in an exit point - thus reducing the complexity. The function is not too long, with 59 lines of code, which also reduces the complexity. What makes the function complex is mostly the multiple cases and statements rather than the logic behind them.

TiffImageParser::getRasterData()
This high CC function was 66LOC so of decent size.

TiffImageParser::getBufferedImage()
cc 27 was 114 LOC.

2.3 What is the purpose of the functions?

PcxWriter::writeImage()
The purpose of this function is to write pcx format images which needs to determine appropriate bit depth and color plane before writing to a file, so the high CC is justified.

TiffImageParser::getImageInfo()
The purpose of the function is to extract metadata information from a TIFF image. It is doing this by reading details like resolution, width and height. The metadata that is extracted is then encapsuled in an object, *ImageInfo*.

BasicCParser::nextToken()
The nextToken()-function is part of the BasicCParser.java-class which basically reads input from a PushbackInputStream. The PushbackInputStream makes it possible for the stream to unread the read byte. As part of the parser, the nextToken() function parses the input stream of bytes into tokens, and returns them as tokens. In the context of image parsing, the function is used to read files with the image file format XPM.

TiffImageParser::getRasterData()
The purpose of getRasterData is to extract raster image data from a TIFF file dirctory and return it as a TIFFRaster object. Also ensureing correct formating.

TiffImageParser::getBufferedImage()
The purpose of the function is to read and convert into a java Buffered Image. *ImageInfo*.

2.4 Are exceptions taken into account in the given measurements?

PcxWriter::writeImage()
Try-catch structure wasn't used with this function, but if it was used, I will treat it as multiple if-else expressions where every single line in the try structure that could trigger the exception is a single if-else pair, with else being the catch part.

TiffImageParser::getImageInfo()
Yes, exceptions are taken inte account in the measurements. The function for example throws Image-Exception when there are missing critical TIFF tags. The function also handles IOException related to reading the TIFF file.

BasicCParser::nextToken()
Regarding the coverage, Clover evaluates the statements which include exceptions to catch whether they are thrown or not. These are often situated in conditional statements which Clover tracks as either fully taken (both true and false), partially (either) or not at all. The manual coverage tool also logged when exceptions were taken.

TiffImageParser::getRasterData()
Yes, exceptions are taken inte account in the measurements. The function for example throws ImageException when there are missing critical TIFF tags.

2.5 Is the documentation clear w.r.t. all the possible outcomes?

There is no documentation, the link for their Javadoc and API doc leads to a Not Found page.

Regarding the nextToken()-function, there is no documentation in the source code other than a comment stating that the code author does not know if the parser is complete. For writeImage() and getImageInfo() there is no documentation neither on the website or in the source code. Other then a few small single line comments describing one line of code there is no documentation for getRasterData().

3 Coverage

3.1 Tools

Clover worked well, it generates an extensive report which contains man measurements and statistics. It was convenient to be able to view what tests hit some lines of code in the functions and which statements were executed fully, partially or not at all. The Clover documentation provided explanations on how the total coverage (which was a measurement showed in the generated report) was calculated, but did not explicitly show the branch coverage. Thus Clover does not present coverage for the branches but rather only the total coverage - it is not clear in the documentation how to either view or calculate the branch coverage through their measurements.

Integrating Clover with the environment was easy if using Maven, as it only needed one plugin in the configuration file to be able to do so. Then generating the report was done through:

```
mvn clean clover:setup test clover:aggregate clover:clover
```

3.2 Your own coverage tool

We implemented a coverage measurement tool and an example of it can be found here.

Our tool supports manually insertion of branch checkpoints in the source code, which can also measures all sorts constructs if configured correctly.

3.3 Evaluation of our own tool

1. How detailed is your coverage measurement?

   Our coverage measurement tool produced a report inside Maven test logs which describes which branch have been covered during testing and which have not, with the line number of that branch in the source file and full calling stack available. The overall coverage is presented as a percentage for each configured function.

2. What are the limitations of your own tool?

   Every branch needs to be assigned an ID and located manually, sometimes this requires adding an 'else' statement in the source code, and this could interfere with the results in the automated coverage measurement tool like JaCoCo.

3. Are the results of your tool consistent with existing coverage tools?

Yes, our own measurement results are quite consistent with Clover's coverage report, exactly describing which branch has been executed during testing and which has not. We also confirmed the results using another coverage measurement tool: JaCoCo and the later produced the same results.

4 Coverage improvement

PcxWriter::writeImage()
The original coverage for the writeImage function was 77.1% and Ziyuan Jia identified the missing branches using the Clover coverage report. After improvements in the test cases, the latest coverage rate for the PcxWriter class increased to 97.8%. The improved test cases can be viewed here, it contains 5 more test cases with different parameter for increased coverage.

BasicCParser::nextToken()
Clover initially generated a total coverage of 67% and the manual instrumentation resulted in 64% branch coverage. Since the manual instrumentation utilizes manual logging, i.e. not automated, human errors in the code might be the reason for the coverage being lower. Additionally it could be because the manual tool measures branch coverage and Clover measures total coverage. Since Clover tracks which conditional statements are evaluated fully (both true and false), one could utilize that to create unit tests that runs through that code. 7 tests were written to cover edge cases and then generated a total coverage (according to Clover) of 98% (from the initial 67%). The test 7 cases added can be viewed here.

TiffImageParser::getBufferedImage()
The clover coverage tool initially reported the total coveraged was 80.7% and the manual instrumentation resulted in 75% coverage. Since the manual instrumentation utilizes manual logging, i.e. not automated, human errors in the code might be the reason for the coverage being lower. The possibility of human error is probably the largest limitation to this "coverage tool". After the added test cases, the coverage increased 89.8% for clover and 90.9% for the manual tool.

5 Refactoring

PcxWriter::writeImage()
The writeImage() has several components, first it generates a Palette from the input parameters, then it tries to determine the appropriate bitDepth and (color)planes accroding to the PCX format specifications, while trying to respect the requested bitDepth and planes settings passed to the class. This has created a huge if-else-if statement which drastically increased code complexity. After that it generates the required file header and write the header, pixels and additional color palette information consequently. Since the main code complexity is introduced by the bitDepth and planes determine process, the refactor plan is to isolate this process as a seperate function determineParameters(). The current if-else-if expressions are considered to be accurate and easily readable, thus they won't be changed to another construct (like switch-case). The write-to-file part of the function is also considered to be feature-unique and atomic, so they won't be separated as well.

The refactoring was carried out by Ziyuan Jia and can be viewed here. This has successfully reduced the cyclomatic complexity of writeImage() from 33 to 12, which is a 63% decrease.

BasicCParser::nextToken()
Since nextToken() contains independent code, refactoring should not be too difficult. For example, if we are parsing an identifier - we would never go into conditional if-statements that regard logic of parsing a string. That logic could be refactored into smaller methods that processes a string and an identifier separately. This would reduce the amount of conditional statements checked, and thus reduce cyclomatic complexity.

The gigantic logical *or* condition statement could also be refactored with a regular expression to reduce the number of expressions needed to be evaluated. Regex makes it possible for only one expression needing to be evaluated instead of 8.

TiffImageParser::getImageInfo()
3There are multiple ways to refactor the code to get functions with lower cyclomatic complexity. One way is to simply split the last part of the function into a new one. The code to move to the new function could in this case include the last two swith case statements. The first switch case is used to determine the image's color type based on the variable *photoInterp*. Thereafter, the code checks for the compression field value in the metadata of a TIFF image and if no compression is found, the default value is used. The last switch case is used to determine the compression algorithm based on the value of the variable compression.

When refactoring in this way, the complexity in getImageInfo is reduced by the factor 20. Therefore, the cyclomatic complexity in getImageInfo is reduced by $1 - (38 - 20)/38 \approx 0,526$. The cyclomatic complexity is therefore reduced by 52,6 %.

TiffImageParser::getRasterData() & TiffImageParser::getBufferedImage
These two functions share a long sequence of if statements for validating an "Rectangle" object sub image. An easy refactoring would be to remove this sequence of if statements and make its own functions where we validate an image without creating any side effects. This was performed for these two functions and the new "validateSubImage(...)" had a complexity of 9, and the complexity of getRasterData(...) was lowered from 35 to 27, and the complexity of getBufferedImage(...) was lowered from 27 to 19. According to measurements given by jacoco report. A potential drawback from this refactoring could be lowered readbility, since the code is abstracted away, and you cant inplace follow the code line by line, without going to the new function.

6 Self-assessment: Way of working

6.1 Principles Established

Checkpoints, principles and constraints were discussed and established amongst the team members in regards to the assignment's constraints and the course content's principles. We focus the work on what the graders will grade, for which the context of the assignment and project is built upon. We continued to use many of the tools we used in assignment one and two, such as Git (GitHub). We decided to work in Java. Before the first meeting, everyone were sent with homework to read through, understand and think about the assignment. On the first meeting we divided some work, and divided the functions. We also discussed how we wanted to work with issues, PRs and merge conflicts and decided to continue like we did on the second assignment. In the next meeting we discussed some of the issues we had encountered, and in the third meeting we worked on the last tasks. These meetings were multiple hours meetings where we worked a lot together on the code.

6.2 Foundation Established

As the tools were established, the foundation was laid and because we worked on different functions we could work quite easily in parallel. Problems encountered with issues were resolved both in creating new issues but also by communication via the agreed social platform (Discord) to avoid multiple members to work on the same bugs or missing features. After the first assignment everyone in the team was familiar to the other team members as well as we are quite similar of the way we are working, which has been improved after assignment two.

6.3 In Use

As the assignment are wrapping up and main functionalities are in process, the practices and tools have been used for real work. The issues and PRs are regularly reviewed, and are fixed by reviewers in case of merge conflicts or errors. Some issues were set up in the first meeting to guide the work forward, but later issues have been more to address newly arisen errors or misinterpretations of the assignment, which showcases the group's way of working adapting towards the assignment's goals. The functionality of issues and commenting on reviews have also been a great way of understanding other people's code and pointing out misunderstandings.

6.4 In Place

We have worked on the functions quite separately so we do not have a perfect understanding of all parts in the code. On the other hand, we reviewed each others results in for example PRs which gave broad understanding of the whole project to all group members. To split the work like we did, led to that everybody is familiar with all the procedures and gives everyone something to do on their freetime away from the meetings. Everybody also has the right to handle the repository for the code and everybody is part of the meetings to review work that has been done or needs to be done.

6.5 Working well

In the end of assignment three, the team has gotten into the new practices, but mishaps and misunderstandings of using tools can still occur, especially when merge conflict occur when multiple people are working on similar things. When problems occur we communicate via Discord and support each other so everyone can learn and contribute to the assignment.

6.6 Retired

The team has not retired its way of working. Not in this phase at all.

6.7 Conclusion

Still, members of the team need some practice, even if not as much as during assignment one or two. Therefore it seems that the team is more between the states of "In Place" and "Working Well" like during assignment one and two, but more close to "Working Well". Our goal is to reach the "Working Well" state during assignment four. The biggest challenge for the team is to work on similar things in the project at the same time, and then merge the work. We will continue to improve our skills within this area in the next assignment.

7 Statement of contributions

All members of the group used issues to track changes in the repository and tasks/subtasks needed to be done.

- Oscar Arbman
  Created skeleton code for the CoverageLogger, and also made it such that it is run after all of the maven tests has run. Did analyzation and improvement of getRasterData()-function, did 8 tests which improved the branch coverage from 72.4% to 80.7%. Implemented refactoring for the function, lowered complexity with 8 from 35 to 27. Aims for P+.

- Lovisa Sjöberg
  Did analyzation and improvement of nextToken()-function. Did 7 tests and improved the branch coverage from 67% to 98% (measured with Clover) and refactored it to reduce the complexity from 28 to 18 (measured with Clover), reduction of 35.7%. Aims for P+. Refactoring available on the patch

  `nextToken_patch.patch`

- Ziyuan Jia
  Did analyzation and improvement of PcxWriter::writeImage() function. Did 5 new tests and improved the existing test cases which brought branch coverage from 77% to 97.8% (measured with Clover) and refactored it to reduce the complexity from 33 to 12 (measured with Clover), a reduction of 63%. Aims for P+.

- Hugo Larsson Wilhelmsson
  Did analyzation of getImageInfo()-function and created two tests which improved branch coverage. Calculated complexity manually for the function, which gave almost the same results as Clover. Also implemented our own tool to measure branch coverage in the function. I described

how to refactor the function, which would result in the cyclomatic complexity to be reduced by 52,6 %. I also wrote on the report and the way of working. Aims for P.

- Erik Diep
  Did analyzation and improvement of TiffImageParser::getBufferedImage() function. Did 5 new tests and improved the existing test cases which brought branch coverage from 75% to 90.8% (measured with Clover) and refactored it to reduce the complexity from 28 to 17 (measured with Clover), a reduction of 40.3%. Aims for P+.