

Report for assignment 3

Authors: Oscar George Judd, Elliot Marklund, Adam Mehdi, Xianqing Zeng & Mary Abeysekera

Date: February 21, 2025

Project

Name: commons-imaging

URL: <https://github.com/apache/commons-imaging>

About

Library for reading and writing images of different formats. Written completely in Java.

Onboarding experience

As a first step, the group members each looked at a few projects from the list of suggestions. After this, the group agreed on commons-imaging. As none of the other projects were deeply examined, we will not mention them. Though it can be said, that one of the reasons for picking commons-imaging, was that it compiled without errors on the first attempt (which was not the case for some other projects).

Did it build and run as documented?

1. How easily can you build the project? Briefly describe if everything worked as documented or not:
 - a. Did you have to install a lot of additional tools to build the software?
JDK and Maven are the only required tools to build the project
 - b. Were those tools well documented?
Yes they were well documented
 - c. Were other components installed automatically by the build script?
Dependencies specified in pom.xml are installed by Maven
 - d. Did the build conclude automatically without errors?
The build completed without errors but with a few warnings
 - e. How well do examples and tests run on your system(s)?
All tests completed and succeeded (except those that were skipped)
2. Do you plan to continue or choose another project?
Continue :).

Complexity

Complex functions:

Function 1:

- `getImageInfo(final ByteSource byteSource, final JpegImageParameters params)`
located at: `org/apache/commons/imaging/formats/jpeg/JpegImageParser` line 299

1. Using Clover, the calculated complexity is 92.

Manual calculation of complexity:

using formula $M = pi - s + 2$

pi = number of decision (if, while, &&, ||)

if = 30

while = 0

for = 4

&& = 37

|| = 4

switch-cases = $4+4+4+5+5+8 = 30$

pi = 107

s = number of exit points

throw: 2

normal exit: 1

s = 3

M = $107 - 3 + 2 = 106$

2. The function is over 300 lines long, which we consider to be rather long. So this particular function with high CC happens to be quite long
3. The function extracts metadata from a JPEG image. The function has many different cases accounting for whether different segments are present in the metadata of the file. A lot of different things are checked in the function, which explains the high complexity.
4. From our understanding, clover takes exceptions into account when calculating complexity. If exceptions are thought of as additional branches, each (possible) exception raises the complexity by 1.
5. The method is not explicitly documented.

Coverage:

Using Clover, the calculated coverage is 47.4%

Using my own calculation, the coverage is:

1001110100111101001111001001011001111100100000000000000000000000 ~36%

The ad-hoc tool (branch `JpegImageParserCoverage`) works by identifying different “branches” and in each of them, setting a bit in a bool array to true, if that part of the code is reached. At the beginning of the function, the array is read from a file, and the array is written back upon return/exception. This way, all branches visited can be overviewed after running the test suite.

A limitation of the tool is that it is hard coded into the function. It is only really usable to untested spots in the current state of the function. If the code would change, the “tool” would need to be adjusted as well.

Using Clover, lines of code that are not covered by tests were identified. One example is the jpeg metadata “density units”, for which only inches (field = 1) were tested.

A few test cases were created in order to increase coverage (branch JpegImageParserTestCleanUp). In particular, images where the “density units field” is set to values other than inches were created and used in tests. Using my own ad-hoc tool, this resulted in two additional branches being covered (39%) coverage. Using Clover, the coverage went up to 49.3%.

While the ad hoc coverage tool and clover present different values, the coverage indicated by both tools increased with the addition of new tests. Though leaving the code for the ad hoc tool yields misleading results from clover, which explains the existence of the branch “JpegImageParserTestCleanUp”. In the branch “JpegImageParserTest”, the new tests as well as the ad hoc tool are present.

Refactoring:

The particular method is quite long and has a high CC. This is however hard to avoid given the task that it performs. While it is possible to divide the method into multiple smaller methods, doing so would only reduce the readability of the code. One example of code that could be separated into a separate function, is the checking of density units (inches/centimeters/...). This would reduce the cyclomatic complexity by eliminating the switch-statement with 4 cases. In addition, it would reduce code repetition, as the density units is currently read in the same way in two different parts of the function.

Function 2:

- `guessFormat(final ByteSource byteSource)`
org/apache/commons/imaging/Imaging line 703
complexity: 36 coverage: 77.1% (clover)

Part 1: Complexity measurement

1. Using Clover the calculated complexity is 36.

Manual calculation of complexity:

using formula $M = pi - s + 2$

pi = number of decision (if, while, &&, ||)

if = 28

while = 0

for = 0

&& = 3

|| = 4

pi = 35

s = number of exit points

return = 21

throws = 4

s = 25

M = $35 - 25 + 2 = 12$

2. The `guessFormat` function is relatively long in terms of lines of code (110 LOC). The function has many conditional checks (if statements) to compare byte sequences against known "magic numbers" of different image formats. Each of these conditionals adds to the cyclomatic complexity (CC), making the function both long and complex.
3. The purpose of the `guessFormat` function is to determine the format of an image file by analyzing its "magic numbers" (signature bytes at the beginning of the file). The high CC is directly related to this purpose because the function must handle multiple formats, each requiring a different check.
4. The documentation provides a general overview but does not explicitly detail all possible outcomes from different branches.

Part 2: Coverage measurement & improvement

Existing coverage tool:

Using Clover, the calculated coverage is 77.1%

Command to create report: `"mvn clean clover:setup test clover:aggregate clover:clover"`.

DIY branch coverage:

Using my own coverage tool (found on branch [guessFormatCoverage](#)), the calculated coverage is 66.67%. The command `"mvn test"` runs the coverage test and prints the result to a separate file called `"branch_coverage_report.txt"`.

1. The branch coverage model implemented here is a basic approach that tracks whether each branch in the function has been executed. It provides a simple binary (executed/not executed) measure of


coverage. The model does not explicitly track ternary operators. If there were ternary expressions in the code, they would not be covered unless rewritten into if-else statements.

2. The branch check statements are manually placed in each branch (defined by if/else statements), which means that it would require manual updates whenever the code changes. The number of branches is also hardcoded. So if new branches are introduced or old ones are removed, the static initialization must be updated.
3. The coverage measured by my tool is not the exact same as that measured by the existing coverage tool used. A reason for this could be that the existing coverage tool measures line coverage as opposed to branch coverage which causes the overall coverage result to be slightly different.

Task 2: Coverage improvement

I created four new tests (found in **ImagingGuessFormatTest.java**) that test five branches which were previously untested. This results in the calculated coverage (using my own coverage tool) increasing to 81.82%.

Clover before and after

Class	Line #	Total Statements	% Filtered	Complexity	Uncovered Elements	TOTAL Coverage
Imaging	90	212	0%	108	98	71,3% 
<code>guessFormat(ByteSource) : ImageFormat</code>	703	75	0%	36	30	77,1%
<code>guessFormat(ByteSource) : ImageFormat</code>	714	108	0%	36	24	85,4%

Task 3: Refactoring plan

The high complexity is somewhat necessary because the function needs to handle multiple image formats with different "magic numbers." However, the way it is structured contributes to unnecessary complexity. To reduce the complexity, the function could be split up into smaller functions. This would both reduce the overall complexity but also shift some complexity to other functions. This could be done with the following:

1. Extract a Separate Method for Format Checking - The repeated `compareBytePair(MAGIC_NUMBERS_XXX, bytePair)` calls could be moved into a dedicated method that returns an `ImageFormat`. This would eliminate redundant if statements in the main function.
2. Use a Map for Magic Numbers Instead of If-Else Chains - A `Map<int[], ImageFormat>` could store magic numbers as keys and formats as values. This would allow lookup instead of multiple conditional checks.

In doing this, I reduced the complexity of the `guessFormat()` function to 9 as calculated by Clover or 5 as calculated by $M = pi - s + 2$.

pi = number of decision (if, while, &&, ||) = 7

s = number of exit points (return, throws) = 4

M = $7 - 4 + 2 = 5$

<code>guessFormat(ByteSource) : ImageFormat</code>	705	20	0%	9	0	100%
<code>checkMagicNumbers(int[], InputStream) : ImageFormat</code>	737	35	0%	16	12	79,7%
<code>getFormatMap() : Map<int[], ImageFormat></code>	791	16	0%	1	0	100%

Function 3:

- preprocess(final InputStream is, final StringBuilder firstComment, final Map<String, String> defines)
org/apache/commons/imaging/common/BasicCParser line 161
complexity: 54 coverage: 56,2% (clover)

Part 1: Complexity measurement

1.

Calculated Code Complexity (CC) using OpenClover: 54

Manual calculation of CC:

Using McCabe's complexity metric formula: $M = \pi - s + 2$

π = number of decision (if, while, &&, ||, switch)

if = 31

while = 0

for = 1

&& = 5

|| = 2

switch/case = 13

$\pi = 52$

s = number of exit points (return, throw)

return = 1

throws = 7

$s = 8$

$M = 52 - 8 + 2 = 46$

2. The getImageInfo function is quite long and has an LOC of 178 excluding blank lines and comments. Since there are numerous conditional checks especially in the form of case/switch and if/else if/else statements the CC becomes quite complex and the function becomes long. There are also numerous assignments in the function which requires a proportion of lines.
3. The function's purpose is to parse code files in C, taking into account the standard conventions of strings, characters, comments and directives. It throws exceptions when errors are detected.
4. There are two types of exceptions in the function. One is handled in the function while the other is propagated. Conditionals (if and switch statements) are used to throw the exceptions. Therefore the CC is more complex by the way the exceptions are detected and handled.
5. Very little documentation is presented about the branches. The different outcomes are more inherent from the naming of the variables and attributes in the function.

Part 2: Coverage measurement & improvement

3.5.1 Task 1: DIY

Existing coverage tool:

Using clover, the calculated coverage was 56,2 %

DIY coverage tool:

(find code in branch preprocessCoverage)

With my own manual instrumentation I got a coverage of 47,3%. The manual instrumentation is divided in the following way:

- Data structures that can hold coverage information:
src/main/java/org/apache/commons/imaging/Coverage.java :
 - Injected coverage measurements:
src/main/java/org/apache/commons/imaging/common/BasicCParser.java
 - Writes information of coverage to a file coverage.txt:
src/test/java/org/apache/commons/imaging/ImageDumpTest.java
1. The coverage tool assigns booleans to branches in the code representing taken/not taken branches. The coverage measurement takes all conditionals into account and rewrites conditionals which lack an else statement to add one if no such exists. It however doesn't account for any ternary operators. Since the conditionals are written before the all existing exceptions, the instrumentation doesn't treat them in any special way, since the branch is noted and taken through the if statement.
 2. The biggest limitation is that the tool requires manual labor in order to insert lines to set a flag/branch has been reached. Some other limitations of my tool currently is that it doesn't take for loops or individual clauses in conditionals into consideration which it could do.
 3. My results aren't entirely consistent with the given results in OpenClover. My computed coverage is slightly smaller than the one computed by Clover, which may be because of the points mentioned above. Another aspect is that Clover may be evaluating line coverage rather than branch coverage and therefore the answers differ.

3.5.2 Task 2: Coverage improvement

(find code in master branch)

The requirements of the functions are commented in the source file as untested/tested. Four additional test cases were made. Since the class didn't have a corresponding test class, one was made and the tests were written inside it. The coverage improved from 56,2% to 60,6 % after the tests were added with OpenClover and from 47,3% to 54 % with my own coverage tool.

3.5.3 Task 3: Refactoring plan

(find code in master branch)

The function I worked with had a very high complexity which may not be necessary. The plan which I utilized to refactor and reduce complexity was to divide the function into smaller units which distributes the complexity of the function across several ones. This was very applicable since the function is based on a for loop checking whether the inputStream which it is parsing is part of a comment, string, single quotes, a directive, which made it natural to divide the function in this way considering you will remain at a certain part of the code for a while based on what the boolean values are set to. Another adjustment made was that instead of passing the boolean values individually, they were accessed through a hashMap defined in the beginning of the class across the functions.

There are other potential ways to refactor the code. One way could be to consider a matrix/array which decides how the parser should operate based on input. This way one could reduce the amount of if statements as some conditionals are repetitive.

The refactoring reduced CC by 74 % as the complexity dropped from 54 to 14, calculated using OpenClover.

Class	Line #	Total Statements	% Filtered	Complexity
BasicCParser	34	295	0%	137 ▾
preprocessinComment(int,ByteArrayOutputStream,StringBuilder) : void	164	16	0%	10
preprocessinString(int,ByteArrayOutputStream,StringBuilder) : void	190	24	0%	8
preprocessinDirective(int,ByteArrayOutputStream,StringBuilder,StringBuilder,Map<String, String>) : void	225	10	0%	7
preprocessinSingleQuotes(int,ByteArrayOutputStream,StringBuilder) : void	244	24	0%	8
preprocessHelper(int,ByteArrayOutputStream,StringBuilder,Map<String, String>) : void	280	39	0%	16
preprocess(InputStream,StringBuilder,Map<String, String>) : ByteArrayOutputStream	335	37	0%	14

Using McCabe's complexity metric formula for the refactored version we get:

$$: M = \pi - s + 2 = 12 - 5 + 2 = 9$$

This indicates complexity improvement by 80 %.

Function 4:

- `getRasterData`([final TiffDirectory](#) directory, [final](#) `ByteOrder` byteOrder, [TiffImagingParameters](#) params)
org/apache/commons/imaging/formats/tiff/TiffImageParser line 713
complexity: 35 coverage 72.4%

Part 1: Complexity measurement

1. Using Clover the calculated complexity is 35.

Manual calculation of complexity:

using formula $M = pi - s + 2$

pi = number of decision (if, while, &&, ||)

if = 21

while = 0

for = 0

&& = 9

|| = 3

pi = 33

s = number of exit points

return = 1

throws = 13

s = 8

M = 33 - 13 + 2 = **22**

2. This method is over 100 lines long 21 of which are if lines the function seems to have a slightly disproportionate CC to LOC ratio but the CC to LOC is not absurdly high.
3. The method is used to "Read the content of a TIFF file that contains numerical data samples rather than image-related pixels". Given its function a lot of the complexity stems for the use of default parameters when an arg is =null.
4. No it appears that the tool does not take into account the throws since the difference in score is exactly the number of throws.
5. Not very explicitly but does mention that the params is optional, and this params is what contains all the parameters that can be null.

Part 2: Coverage measurement & improvement

Using coverage tool:

Using Clover, the calculated coverage is 72.4%

3.5.1 DIY coverage: The DIY coverage was implemented in the branch `getRasterDataCoverage`.

My DIY analysis of coverage finds that 10 out of 21 branches are taken. That evaluates to a 47.62% coverage. A lot lower than what is found by Clover. This is most likely due to a lot of the unvisited branches throwing exceptions. It appears that Clover does not give as much weight to the exception branches and as a result has a much higher score.

1. In terms of quality my DIY does not take into account ternary operators (but I do not think there are any in the method) and branches ending in exceptions are counted.
2. My approach is limited by it being hard-coded meaning it can not be applied to other functions without significant work. It also does not consider exception branches less important than other branches leading to less useful results. It writes to a file directly (large overhead) because each time the method is called it is on a separate instance of the class. This does mean it is more robust to changes in the code.
3. The found coverage of 47.62% is a lot lower than what is found by Clover. This is most likely due to a lot of the unvisited branches throwing exceptions. It appears that Clover does not give as much weight to the exception branches and as a result has a much higher score.

3.5.3 Coverage improvement

The tests that were created for this part of the assignment are on the branch `getRasterDataNewTests` and pushed to master.

Adding 6 tests to `\src\test\java\org\apache\commons\imaging\formats\tiff\TiffImageParserTest.java` managed to raise the coverage percentage calculated by Clover from 72.4% to 88.6%, a significant increase.

Refactoring:

It seems that all of the cognitive complexity is required for the proper functioning of the method. So the best way to refactor the function to reduce cognitive complexity would be to split up the function. Having helper functions process for example all the exception cases under the `if (subImage != null)` line 749 or the

`if (sSampleFmt[0] == TiffTagConstants.SAMPLE_FORMAT_VALUE_IEEE_FLOATING_POINT)` line 798 would distribute the cognitive complexity. This distribution would lead to no function having an extremely high complexity while still leaving the code readable and coherent.

Function 5:

- `writImage(final BufferedImage src, final OutputStream os, GifImagingParameters params)`

`org/apache/commons/imaging/formats/gif/GifImageParser`

line 842

complexity: 33, coverage: 82.9% (clover)

Part 1: Complexity measurement

1. Calculated Code Complexity (CC) using OpenClover: 33

Manual calculation of CC:

Using McCabe's complexity metric formula: $M = \pi - s + 2$

π = number of decision (if, while, &&, ||, switch)

if = 26

while = 0

for = 4

&& = 0

|| = 0

switch = 0

$\pi = 30$

s = number of exit points (return, throw)

normal exit: 1

throws = 1

$s = 2$

$M = 30 - 2 + 2 = 30$

2. The `writImage` function is relatively long with 142 LOC and 30 CC but not extreme compared to other high-CC functions like `getImageInfo` (over 300 LOC). Given its role in GIF encoding, it contains multiple conditionals, contributing to a high CC, though its LOC remains within a reasonable range for its complexity.
3. The function's purpose is to write a GIF image, including handling color palettes, metadata (XMP), transparency, and compression. This directly correlates with its high CC, as it requires numerous conditional checks to determine palette configurations, compression parameters, and metadata inclusion.
4. The OpenClover tool may does take exceptions into account when calculating CC, as seen in the difference between the manual calculation (29) and OpenClover's reported CC (33). In the manual calculation, throws were counted as exit points (s), reducing the final CC value. If exceptions are considered additional branches leading to alternative execution paths, each throw increases CC by 1, which explains the difference between the two CC values.
5. The Javadoc provides a general description of the function and its parameters, but it does not fully explain how different branches lead to different outcomes within the function. While it mentions `params` as an optional argument, it does not explicitly state what happens when it is null or how different format-specific behaviors affect execution.

Part 2: Coverage measurement & improvement

Task 1: DIY

Existing coverage tool:

Using clover, the calculated coverage was 82.9%

Created coverage tool:

Using my own coverage tool (found on branch **writelImageCoverage**), the calculated coverage is 43.33%

1. The quality of the DIY coverage measurement is basic and relies on manual instrumentation. The tool accounts for ternary operators by manually analyzing different branches and track them. For example, in the code, ternary operators like `hasAlpha ? 0 : 2` were tracked by adding branch points for both outcomes. Exceptions are not automatically tracked unless `trackBranch` calls are explicitly added to exception-handling code. In the analyzed `writelImage` function, exceptions like `throw new ImagingException(...)` were not tracked, so exception branches are not included in the coverage results.
2. The limitations include:
 - Manual instrumentation: Every branch requires explicit insertion of `trackBranch` calls, which is error-prone and time-consuming.
 - Hard-coded branch count: The total branch count (`branchCnt` = 30) is fixed, so adding or removing branches requires manual updates to avoid index mismatches.
 - No support for dynamic branches: Ternary operators, loops, or exception-based control flow are not automatically handled.

If the program is modified (e.g., adding new branches or changing conditions), the instrumentation would need to:

- Adjust `branchCnt` to match the new total number of branches.
 - Insert `trackBranch` calls at all new branch points.
 - Update existing branch indices if code structure changes, which risks breaking consistency.
3. The results are not consistent with Clover. The DIY tool reported 43.3% branch coverage, while Clover reported 82.9%. There might be some reasons like:
 - Clover may measure line coverage or include exception-related branches, while the DIY tool focuses on explicit branch outcomes.
 - Some branches (e.g., exception throws, loops, or implicit ternary operators) were not tracked in the DIY tool.

Task 2: Coverage improvement

Before New Tests:

Custom Coverage Tool

- Executed Branches: 20 / 30
- Coverage Percentage: 66.67%

OpenClover

- Coverage: 82.9%

After New Tests

Custom Coverage Tool

- Executed Branches: 24 / 30
- Coverage Percentage: 80.0%

OpenClover

- Coverage: 91.5%

Self-assessment: Way of working

Seeded

The seeding was essentially put in place by the course's framework. Our first meeting was centered around making sure each member of the team understands the objectives and constraints of the project.

Formed

The team had already been formed from previous assignments. Given the low organizational complexity of this task no special roles were attributed and the responsibility to make sure everyone was making progress was shared.

Collaborating

Compared to previous assignments, this one allowed for the group members to work independently for most of the project. As such, not much collaboration was necessary between initial distribution of the work, and final compilation of the report. While it is difficult to assess our progress on collaboration, the group should aim to increase the frequency of following up on individual progress. Regular virtual meetings were used as a means to evaluate our progress, ask each other questions and decide on delegated tasks that appeared as we understood the assignment more in depth.

Performing

The team was able to meet the given requirements. A need for more timely organisation was however noted. More concretely the first meeting should be scheduled sooner to allow each member to have a large window to accomplish his designated work.

Adjourned

Not applicable at the current stage.

Overall experience

What are your main take-aways from this project? What did you learn?

- We learnt about the advantages of using tools to check for complexity and coverage.
- We learnt how to navigate and understand parts of a larger code base, understanding parts without understanding the whole

Is there something special you want to mention here?

Not particularly

Statement of contributions:

Mary Abeysekera:

- Function 3
- Attempted P+:
 - Wrote four tests
 - Used issue tracker and systematic commit messages
 - Refactored function to reduce complexity by 74 %

Oscar Judd:

- Function 4

Elliot Marklund:

- Function 1 (JpegImageParser/getImageInfo)
- Initial assessment of complexity and coverage using Clover
- Proposed the project that was ultimately chosen (commons-imaging)

Adam Mehdi:

- Function 2 (guessFormat)
- Attempted P+:
 - Wrote four tests
 - Used issue tracker and systematic commit messages
 - Refactored function to reduce complexity by at least 35 %

Xianqing Zeng:

- Function 5
- Attempted P+:
 - Wrote eight tests
 - Used issue tracker and systematic commit messages
 - Refactored function to reduce complexity by 48%