# Code Complexity and Coverage Analysis

Group 14
DD2480 Software Engineering Fundamentals
KTH Royal Institute of Technology
Jesper Lindeberg, Hugo Sacilotto, Hanzhi Zhang, Jeffrey Chang, Yuning Wang

February 21, 2025

## 1 Introduction

The purpose of this report is to analyze the complexity and test coverage of an open-source project. We aim to improve coverage by creating additional test cases and propose refactoring strategies to simplify complex functions.

## 2 Project Selection and Onboarding

The project we selected was the Apache Commons Imaging project GitHub.

### 2.1 Onboarding Process

1. How easily can you build the project? Briefly describe if everything worked as documented or not:

   - Did you have to install a lot of additional tools to build the software?

   Not really because the project was built in Java and tests were run using Maven. We have already used them in previous assignments. No other tools were required to build the software.

   (a) Were those tools well documented?

   The tools were decently well documented.

   (b) Were other components installed automatically by the build script?

   Yes (Maven)

   (c) Did the build conclude automatically without errors?

   Yes, Maven took care of the build.

   (d) How well do examples and tests run on your system(s)?

The examples ran well on the system, since it was built in java setup was minimal because our of previous Java installations. Maven took care of all of the dependency management.

2. Do you plan to continue or choose another project?

   Yes we plan to continue.

# 3 Complexity Measurement

We used OpenClover to run a complexity measurement on the code base. The following code metrics were obtained.

**Branches**: 5,354 **Statements**: 15,438 **Classes**: 451 **Packages**: 43
**LOC**: 51,026 **NCLOC**: 30,974 **Total complexity**: 6,090
**Complexity density**: 0.39 **Statements/Method**: 6.53 **Methods/Class**: 5.24
**Classes/Package**: 10.49 **Average method complexity**: 2.58

Here is the description provided in OpenClover's documentation on how complexity is calculated [1]

- empty method complexity == 1
- simple statement complexity == 0
- switch block complexity == number of case statements
- try-catch block complexity == number of catch statements
- ternary expression complexity == 1
- boolean expression complexity == number of `&&` or `||` in expression

## 3.1 Cyclomatic Complexity Analysis

We calculated the cyclomatic complexity, $M$, using the formula (provided in the lecture slides)

$$M = \pi - s + 2,$$

where $\pi$ is the number of decisions (`if, while, &&, ||`) and $s$ is the number of exit points (`return, throw`).

### 3.1.1 Findings from five complex functions

1. `TiffImageParser::getRasterData`

   OpenClover complexity count: 35

   Calculated by hand (Hugo) yielded $\pi = 34, s = 14$ which resulted in $M = 22$.

   Calculated by hand (Jesper) gave: $\pi = 32, s = 14$ which results in $M = 20$ *same as above.*

   Calculated by hand (Hanzhi) yielded $\pi = 32, s = 14$ which results in $M = 20$.

   Calculated by hand (Jeffrey) yielded $\pi = 33, s = 14$ which resulted in $M = 21$.

Calculated by hand (Yuning) gave the same result as Jeffrey.

2. `TiffImageParser::getImageInfo`

   OpenClover counted the complexity of this function to 36.

   Calculating by hand (Hanzhi) yielded $\pi = 38, s = 2$ which resulted in $M = 38$.

   Calculating by hand (Jeffrey) yielded $\pi = 38, s = 2$ which resulted in $M = 38$.

   Calculating by hand (Yuning) yielded $\pi = 36, s = 2$ which resulted in $M = 36$.

3. `GifImageParser::getXmpXml` OpenClover complexity count of the function: 12

   Calculated by hand (Jesper) gave: $\pi = 10, s = 4$ which resulted in $M = 8$

   Calculated by hand (Hanzhi) gave the same result.

   Calculated by hand (Hugo) gave the same result.

   Calculated by hand (Jeffrey) gave the same result.

   Calculated by hand (Yuning) gave the same result.

4. `MostPopulatedBoxesMedianCut::performNextMedianCut` OpenClover counted the complexity of this function to 19.

   Calculating by hand (Yuning) yielded $\pi = 20, s = 4$ which resulted in $M = 18$.

   Calculating by hand (Jesper) gave the same as Yuning which resulted in $M = 18$.

   Calculating by hand (Hugo) gave the same as Yuning which resulted in $M = 18$.

5. `TiffImageParser::getBufferedImage` OpenClover counted the complexity of this function to a CCN of 28.

   Calculating by hand (Hanzhi) yielded $\pi = 25, s = 10$ which resulted in $M = 17$.

   Calculating by hand (Jeffrey) yielded $\pi = 24, s = 10$ which resulted in $M = 16$.

   Calculating by hand (Yuning) yielded $\pi = 23, s = 10$ which resulted in $M = 15$.

### 3.1.2 Relation Between CC and LOC of High CC Functions

A function with high LOC does not necessarily have high CC, if for instance many sequential operations are performed without much branching. A function with high CC, on the other hand, will usually have quite high LOC. It is difficult to have a lot branching without having many lines of code (unless the code is poorly formatted or contains deeply nested conditionals or loops). In our case, the functions with high CC also have quite high LOC ranging from around 50 lines to over 200 lines.

### 3.1.3 Purpose of These Functions

**Function 1:** The function `TiffImageParser::getRasterData` takes a container for a TIFF image and some parameters and returns the rasterization data for the image. It has a high complexity, mostly due to many different conditional checks that can throw exceptions. The complexity could easily be lowered by extracting some functionality into separate functions. There are for example many different if-statement checking if a

subimage adheres to the valid specification. That logic could be its own function which would lead to a complexity reduction for this function.

**Function 2:** The functionality of `getImageInfo` extracts detailed image metadata from the provided byteSource and TiffImagingParameters. The function's moderate complexity is due to several checks and calculations that ensure the image metadata is valid, including field lookups, resolution, and compression details.

**Function 3:** The functionality of `getXmpXml` extracts the XML metadata as an XML string from the provided byteSource. Its moderately high complexity count is due to several checks ensuring the data format is correct and an XMP exists in the byteSource.

**Function 4:** The `performNextMedianCut` function partitions color groups using the Median Cut algorithm to refine color quantization. Its high cyclomatic complexity (CCN) stems from multiple nested loops, numerous conditional branches (if, for, switch), and recursive decision-making for selecting the best median cut.

**Function 5:** The functionality of `getBufferedImage` is that it decodes TIFF (Tag Image File Format) image data into a `BufferedImage` by interpreting metadata, handling compression, and managing colour spaces. The reason it has a high complexity is that it validates sub-image boundaries to check width and height and processes planar/chunky pixel layouts, ensuring accurate conversion of raw pixel data into a usable image format.

### 3.1.4 Exception CC in Clover

In Java, the code utilities try and catch statements for exceptions. Clover counts each catch statement as a potential branch similar to how if-statements with a single expression are counted.

### 3.1.5 Documentation about the different branches

In general, the functions' branches aren't well documented and the outcome of each branch isn't clear. In some of the functions, a lot of Javadocs are missing regarding potential throws and early exceptions which might happen. Some of the branches taken have self-descriptive code and the outcome can be inferred by reading the code. However, that isn't the case for branches traversing several others before returning.

# 4 Coverage Measurement and Improvement

Coverage was measured using the tool OpenClover.

OpenClover measures coverage in the following way [1]

$$TPC = (BT + BF + SC + MC)/(2B + S + M) \cdot 100\%$$

where

$BT$ − branches that evaluated to "true" at least once
$BF$ − branches that evaluated to "false" at least once
$SC$ − statements covered
$MC$ − methods entered
$B$ − total number of branches
$S$ − total number of statements
$M$ − total number of methods

Below are the OpenClover coverage values for each of the five functions.

1. `TiffImageParser::getRasterData`: 72.4%
2. `TiffImageParser::getImageInfo`: 83.3%
3. `GifImageParser::getXmpXml`: 68.8%
4. `MostPopulatedBoxesMedianCut::performNextMedianCut`: 75.3%
5. `TiffImageParser::getBufferedImage`: 73.5%

## 4.1 Manual Coverage Measurement

The manual coverage measurement was implemented by creating a static int array to keep track of which branches are taken. The last test to run was identified and a function was added there to run after all tests (using the `@afterAll` decorator with JUnit) which prints the results. For each of the functions to measure coverage on, we changed to code to set the index of the boolean array corresponding to each branch id if the branch is taken.

Below are the results from our manual branch coverage measurement.

1. `TiffImageParser::getRasterData`: 64.3%

   This was implemented by Hugo. The code is available in the branch `chore/12-getRasterData-coverage`. Ternary operators were taken into account be rewriting them as regular if-statements. Exceptions were also taken into account. The result is not too far off from OpenClover's result at 72.4%, however this only includes branch coverage while OpenClover uses a more complicated formula, combining different kinds of coverage.

2. `TiffImageParser::getImageInfo`: 58.3%

   This was implemented by Hanzhi. The code is avaliable in the branch `feature/13-TiffImageParser-getImageInfo-coverage`. Exceptions were taken into account. The DIY result is not quite far from OpenClover's result 83.3%, not only because of the reason mentioned above but also because many other tests also use this function, but since they are scattered across various tests, it is difficult to aggregate them in one place for centralized statistics.

3. `GifImageParser::getXmpXml`: 66.7% DIY Branch Coverage
   This was implemented by Jesper. Code is available on the branch

`feature/coverage-getxmpxml`. The result is not far off OpenClover's result at 68.8%. The DIY branch coverage was very basic but due to it having slightly less complexity, 12 according to Clover and 8 with manual calculations, and thus there's less complexity for OpenClover to analyse making the DIY solution and the OpenClover implementation almost identical.

4. `MostPopulatedBoxesMedianCut::performNextMedianCut`: 56.0%
   This was implemented by Yuning. Code is available on the branch `feature/14-performNextMedianCut-Coverage-DIY`. Excepetions are taken into account. The result is lower than OpenClover's result. This basic DIY method measures branch coverage more conservatively and one-sided. Clover, on the other hand, may be more sophisticated in recognizing implicit branches or may include additional edge cases and conditions like exceptions, early returns, or loop iterations.

5. `TiffImageParser::getBufferedImage`: 65.2% DIY Branch Coverage

   This was implemented by Jeffrey. The code is available on the branch `feature/Coverage-getBufferedImage`. The result is slightly different from OpenClover's result at 82.4%. This could be due to the fact that my DIY coverage of getBufferedImage only covers explicit branch points whilst OpenClover has a more comprehensive formula, as mentioned previously. The DIY checks also only cover specific test files whilst the function is present in other files.

## 4.2 New Test Cases

Summary of new test cases added to improve coverage.

1. `TiffImageParser::getPhotometricInterpreter` 65.4% → 88.5% (OpenClover)
   This function has a complexity of 11 (OpenClover). The two tests were implemented by Hugo. Both of the tests set a parameter to different values which makes cases in a switch statement be executed that are not otherwise covered. The tests assert that the correct value is returned or exception is thrown.

2. `GifImageParser::getXmpXml`: 66.7% → 91.7% (OpenClover) 83.3% (DIY)
   The two tests were implemented by Jesper. The first test is a simple test which validates that the function returns a valid string given a correct gif image. The second test was ensuring that the code throws an ImagingException in case of a gif with two XMP blocks. Both of the gifs utilised for testing the code were uploaded and the second one was modified based on the original valid one with an additional XMP block.

3. `MostPopulatedBoxesMedianCut::performNextMedianCut`: 75.3% (OpenClover) 56.0% (DIY) → 82.5% (OpenClover) 72.0% (DIY)
   The four tests were implemented by Yuning.
   The first test `PaletteQuantizationTest::testSingleColorGroup` has only one color count, to validate if the method returns false because it can't make a valid median cut with just one group. The second test `PaletteQuantizationTest::testNoValidColorGroup` is to to ensure that a group with no valid difference (maxDiff ¡= 0) is correctly handled.
   `PaletteQuantizationTest::testMedianIndexAtLastElement` verifies that when the median index falls at the last element in the color list, the function correctly

adjusts the index and continues execution without errors.
`PaletteQuantizationTest::testSwitchStatementCoversGreen` ensures that the function selects the GREEN color component for splitting by structuring input data to favor green, thus increasing branch coverage in the switch statement.

4. `TiffImageParser::getImageInfo`: 83.3% (OpenClover) 58.3% (DIY) → 85.1% (OpenClover) 64.0% (DIY)
   The two test cases were implemented by Hanzhi. 2 `.tif` pictures `1pagefax.tif` and `Oregon Scientific DS6639 - DSC_0307 - small CCITT T.6.tiff` are input into the parser and they cover 2 Compression Algorithm branches that previous test cases have never used, which are `ccitt_1d` and `ccitt_group_4` Compression Algorithms.

5. `TiffImageParser::getBufferedImage`: 73.5% (OpenClover) 65.2% (DIY) → 80.0% (OpenClover) 73.9% (DIY)
   The two tests were implemented by Jeffrey.
   The tests were implemented using mockito to mock TIFF file inputs and the first test checks if the subImage width is of acceptable width, i.e. 0 or less. The mocking parameters test an exact 0 width which throws an exception. The second test makes sure that there is no mismatch between `samplesPerPixel` and `bitsPerSample`, so the amount of pixels equals length. Again, we mock that there is one sample pixel but an array length of 2 which throws another exception.

# 5 Refactoring Plan

Proposed changes to simplify complex functions and improve maintainability.

1. `TiffImageParser::getRasterData`

   This function is quite long at over 200 lines and its complexity can easily be decreased by extracting logic into seperate functions. One area where this can easily be done is part with several if-statements that checks for a valid subimage specification. This can be moved to a different function that throws an `ImagingException` and does not need to returning anything.

2. `MostPopulatedBoxesMedianCut::performNextMedianCut`
   To refactor the performNextMedianCut function and reduce its Cyclomatic Complexity Number (CCN), we can break down the function into smaller, more manageable methods. This helps improve readability and maintainability. Here's a refactor plan:

   - Extract Method for Finding the Color Group with Maximum Points: Create a method findMaxPointsColorGroup that iterates over colorGroups to find and return the group with the maximum points.
   - Extract Method for Calculating the Best Median Cut:Create a method calculateBestMedianCut that handles the logic for determining the best median cut for each ColorComponent.
   - Extract Method for Finding Median Index: Create a method findMedianIndex that calculates the median index based on the countHalf, oldCount, and

newCount.

- Extract Method for Creating Color Groups: Create a method createColorGroups that splits the colorCounts into lowerColors and upperColors and returns the corresponding ColorGroup objects.
- Refactor Main Logic: Refactor the main logic to use these helper methods, making the function more concise and easier to follow.

A practice following this plan can be found in the branch `feature/25-performNextMedianCut-refactor`. By splitting the codes into smaller segments, the complexity of the function decreases from 19 to 3, evaluated by OpenClover.

3. `GifImageParser::getXmpXml` Link
   This function isn't that long and a lot of the complexity comes from it's error handling and edge cases for the XML. Thus reducing it's complexity can prove to be a real tough challenge. It could be split up into several functions however it would most likely do more harm than good to the codebase due to increased jumping in the control flow. Furthermore the function is already utilising some private functions to help alleviate the complexity. This can be further built upon with the plan below:
   - Extract method getXmpStringBlocks into it's own private function instead of having a large for loop in the class.

   However, one improvement to the code which could be done is to bring out the large for loop which reads all of the XMP data and then return the string.

4. `TiffImageParser::getImageInfo`
   To refactor this function we should follow the Single Responsibility Principle (SRP), creating smaller, focused methods for each subtask to improve readability and maintainability. For example we can create a `readTiffContents()` method to handle reading the TIFF content from `ByteSource` and `TiffImagingParameters` and also eliminated nested if and switch statements to reduce the complexity.

5. `TiffImageParser::getBufferedImage`
   To refactor the `getBufferedImage` function and reduce its complexity and Cyclomatic Complexity Number (CCN), one can split the logic into separate, focused methods.
   - Extract all checks for invalid subimage dimensions (e.g., width $\leq 0$, height $\leq 0$, and out-of-bounds coordinates) into a single method which would throw exceptions if any dimension is invalid. By doing this, the main flow in `getBufferedImage` is cleaner and it avoids deeply nested `if` statements and making it more readable.
   - Move the `switch` or series of `if` statements for interpreting photometric data into a dedicated method. By doing this, `getBufferedImage` only needs to invoke that method to obtain the correct `PhotometricInterpreter` instance which keeps the main function's code concise and makes it more maintainable.

# 6 Self-Assessment

Based off our checklist in Issue #24 we're in the beginning of the **Collaborating** phase. There are some missing requirements in the **Seeded** phase however since these SEMAT requirements are a bit abstract it's difficult to effectively apply them to the project. What we're missing to progress is that the team needs to be able to continuously and seamlessly work as one cohesive unit and we need to know and trust each other which we don't fully do yet because we have only worked as a team for 2 previous assignments and this assignment was quite independent.

In general, reaching that level of competency in the team is difficult since we've only known each other from the beginning of the course and have only 6 weeks to develop our skills together as a team.

# References

[1] Atlassian. *About Clover code metrics*. URL: `https://openclover.org/doc/manual/latest/general--about-openclover-code-metrics.html` (visited on 02/19/2025).