EECS 4313

Software Engineering Testing

Final Project Report — Testing Thumbnailator

April 7, 2025

Group F

Isaiah Gocool
goisaiah@my.yorku.ca
218918052

Mher Eric Gyulumyan
mericg@my.yorku.ca
217990540

Daniel Di Giovanni
dand02@my.yorku.ca
218204818

Contents

1 Introduction

Testing is an essential component of the software development lifecycle, ensuring that code behaves as expected and meets specified requirements. By systematically validating the functionality and performance of a program, testing helps uncover hidden bugs, improve reliability, and facilitate future maintenance. Robust testing practices are particularly important in environments where software quality and stability are paramount, such as in production systems and mission-critical applications.

This project involved analyzing and testing an existing software project called Thumbnailator. Thumbnailator is an open-source Java library designed to simplify the creation of image thumbnails with minimal setup and external dependencies. Specifically, two classes in the Thumbnailator library will be tested here: `ThumbnailatorUtils.java` and `BufferedImageBuilder.java`.

In addition to traditional test design, this report incorporates the use of generative AI tools to enhance test case generation, increase code coverage, and analyze potential faults or refactoring opportunities. Coverage metrics are measured using JaCoCo (Miller's metric), with a focus on achieving near-complete line and branch coverage.

The remainder of this report details the chosen testing techniques, justifies their application to individual methods, documents any bugs or refactoring suggestions, and evaluates coverage results. All tests are implemented using JUnit 5 and are located in the src/test/java directory of the project.

2 Testing Methods

The testing methods for each of the two classes will be presented here, as well as their results.

2.1 ThumbnailatorUtils.java

The `ThumbnailatorUtils` class is a final utility class within the Thumbnailator library that provides static methods for querying supported image output formats and compression types. It acts as a lightweight wrapper around the Java ImageIO API, offering convenience methods for checking format availability and compatibility with specific compression types. These functions play a key role in determining how images are saved and encoded, making their correctness essential for the reliable generation of thumbnails.

2.1.1 Equivalence Class Testing for ThumbnailatorUtils.java

For the testing of the ThumbnailatorUtils class, Equivalence Class Testing was selected as the primary technique. This method was well-suited for the nature of the class, which consists of utility functions that operate on clearly defined inputs and produce predictable,

categorized outputs. Each method in this class is deterministic, and the expected outcomes can be effectively grouped into equivalence classes. For example, valid versus invalid image format strings, supported versus unsupported compression types, and null versus non-null return values from the underlying ImageIO API.

We applied a robust equivalence class testing approach to ensure that both valid and invalid inputs were covered. For methods that accepted a single input parameter, such as `getSupportedOutputFormats()` and `isSupportedOutputFormat(String format)`, we used the weak robust method, selecting one valid and one or more invalid values to test boundary and error-handling behavior. In contrast, for methods accepting multiple parameters, such as `isSupportedOutputFormatType(String format, String type)`, we applied strong robust equivalence testing to evaluate all combinations of valid and invalid inputs, ensuring thorough interaction coverage between the parameters.

Overall, equivalence class testing was an efficient and logical choice for this class because it allowed us to validate functional correctness across the full behavioral spectrum while minimizing redundant or low-value test cases. The clear structure and deterministic behavior of these methods made them ideal candidates for this technique.

2.1.2 Weaknesses of Equivalence Class Testing ThumbnailatorUtils.java

While equivalence class testing proved to be an effective technique for covering a wide range of expected behaviors, it does have limitations in this context. One of the key drawbacks is that it does not inherently account for internal control flow or edge cases within complex logic branches. For example, the method `isSupportedOutputFormatType(String format, String type)` contains multiple conditional paths, some of which are tightly coupled with constants like `ORIGINAL_FORMAT` and `DEFAULT_FORMAT_TYPE`. Although equivalence class tests can verify correctness for typical input scenarios, they may miss subtle bugs hidden in rare or unusual paths unless specifically targeted.

Furthermore, this technique assumes that inputs within the same equivalence class will result in the same output behavior, which may not always be the case in methods that interact with external libraries like ImageIO. These dependencies can introduce inconsistencies depending on platform support or registry state, potentially causing false confidence if not supplemented with additional testing methods. In such cases, more granular structural techniques, like path testing or decision coverage, might be necessary to ensure that all control paths have been exercised.

Despite these limitations, equivalence class testing still offered strong value in efficiently validating the public-facing functionality of ThumbnailatorUtils. However, in a production environment, it would be advisable to complement it with more exhaustive white-box techniques for maximum reliability.

2.2 BufferedImageBuilder.java

The `BufferedImageBuilder.java` class is a utility designed to simplify the construction of BufferedImage objects in Java. It provides multiple constructors and method chains that allow clients to configure the width, height, and image type of the image to be created. Internally, it applies validation to ensure that dimensions are positive and handles specific cases like substituting unsupported image types (e.g., `BufferedImage.TYPE_CUSTOM`) with safe defaults. The `build()` method is responsible for returning a new image instance with the specified attributes. The class is central to how images are generated in a controlled and reusable way within the Thumbnailator library.

2.2.1 Equivalence Class Testing for BufferedImageBuilder.java

To test this class, Equivalence Class Testing was selected as the primary technique. This method was an ideal fit given that the class operates on a finite and well-defined set of input parameters: integer values for width, height, and image type. These inputs naturally form logical groups of valid and invalid values, allowing us to define equivalence classes such as positive dimensions versus zero or negative values, and supported image types versus special or custom types.

We used a robust equivalence class testing approach to ensure input validation was thoroughly exercised. For constructors and methods that accepted single inputs, such as `width(int)` and `height(int)`, we employed weak robust equivalence classes, validating representative valid values (e.g., 1, 100) and invalid ones (e.g., 0, -5). For constructors or methods involving multiple parameters, such as `(int width, int height, int imageType)`, we used strong robust equivalence testing, evaluating combinations of valid and invalid values across each parameter to ensure defensive programming logic was correctly enforced.

This approach enabled us to systematically verify the builder's correctness and input validation across both constructor-based and method-chained object creation paths without exhaustive testing of the inputs. The class's tight scope and clear validation logic made it highly compatible with this technique.

2.2.2 Weaknesses of Equivalence Class Testing BufferedImageBuilder.java

However, equivalence class testing has its limitations. It does not provide direct insight into internal control flow or ensure coverage of every possible execution path. For example, while we tested input classes that trigger validation logic, equivalence class testing does not guarantee that all edge-case interactions between methods like `width()`, `height()`, and `imageType()` have been fully explored unless explicitly designed for. Additionally, this technique assumes uniform behavior across members of the same class, but if unexpected branching or undocumented behavior were present, they could potentially go undetected.

Despite these limitations, equivalence class testing offered a structured and focused strategy for verifying the functionality of `BufferedImageBuilder`. The class's clear parameter constraints and straightforward behavior made it an ideal candidate for this technique. In a more complex system, this approach would benefit from being supplemented by structure-based techniques such as decision or condition coverage to ensure complete control flow validation.

3 Testing Results

This section explains the results of testing both the `ThumbnailatorUtils.java` and `BufferedImageBuilder.java` classes, as well as suggestions for improvement.

3.1 Testing Results of ThumbnailatorUtils.java

All of the test cases we made for the `ThumbnailatorUtils.java` class are passing, as shown in the IntelliJ screenshot in Figure 1.

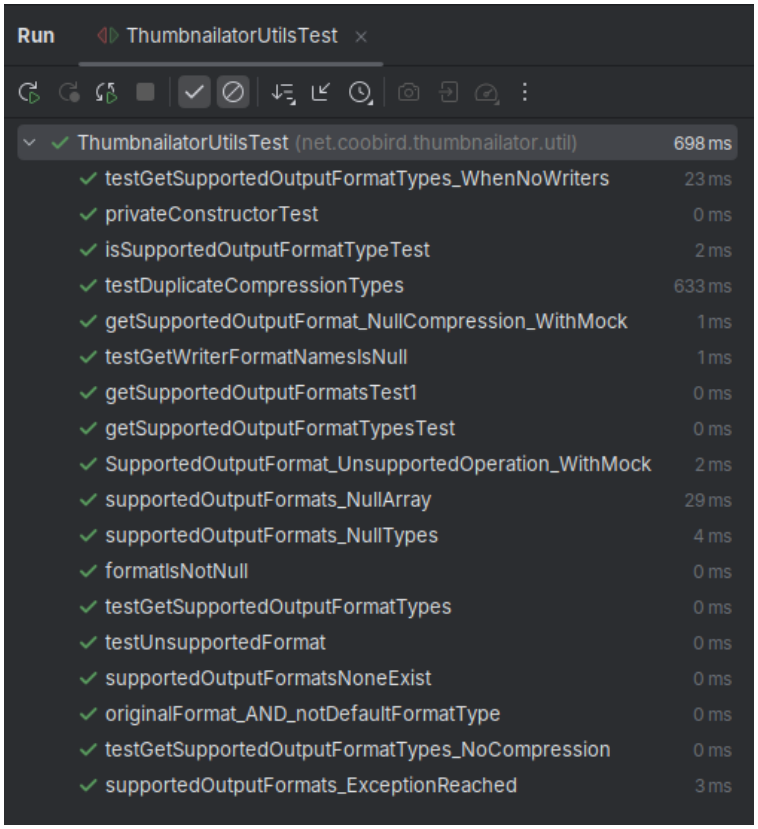
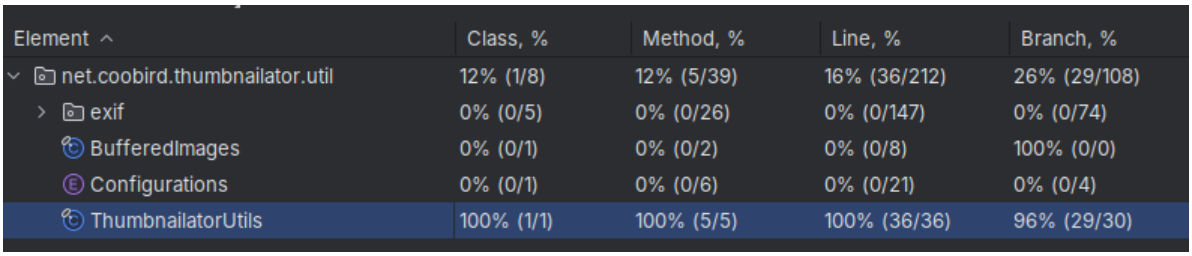


Figure 1: All test cases passing for ThumbnailatorUtils.java.

No faults were discovered during testing of the ThumbnailatorUtils class. The test suite effectively covered all public methods and validated expected behavior across a wide

range of input combinations. Code coverage analysis using JaCoCo reported 96% line and branch coverage, as shown in Figure 2.



Element	Class, %	Method, %	Line, %	Branch, %
net.coobird.thumbnailator.util	12% (1/8)	12% (5/39)	16% (36/212)	26% (29/108)
exif	0% (0/5)	0% (0/26)	0% (0/147)	0% (0/74)
BufferedImages	0% (0/1)	0% (0/2)	0% (0/8)	100% (0/0)
Configurations	0% (0/1)	0% (0/6)	0% (0/21)	0% (0/4)
ThumbnailatorUtils	100% (1/1)	100% (5/5)	100% (36/36)	96% (29/30)

Figure 2: 96% branch coverage for ThumbnailatorUtils.java.

Upon inspection of the coverage report, the uncovered branch was found in a conditional check that is unreachable by design. This code section is shown below in Listing 1. The second condition of the second branch (`type != ThumbnailParameter.DEFAULT_FORMAT_TYPE`) did not get fully tested, as shown in the JaCoCo screenshot in Figure 3.

```
1 if (format == ThumbnailParameter.ORIGINAL_FORMAT
2         && type == ThumbnailParameter.DEFAULT_FORMAT_TYPE) {
3     return true;
4 } else if (format == ThumbnailParameter.ORIGINAL_FORMAT && type !=
    ThumbnailParameter.DEFAULT_FORMAT_TYPE) {
5     return false;
6 } else if (type == ThumbnailParameter.DEFAULT_FORMAT_TYPE) {
7     return true;
8 }
```

Listing 1: Code section that did not achieve 100% coverage.

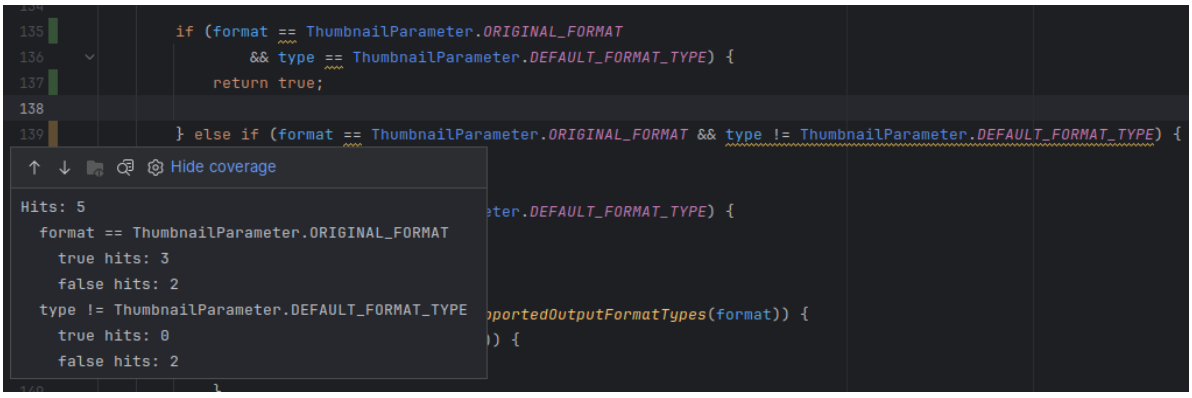


Figure 3: JaCoCo feedback of untested conditional branch.

To aid in analysis, we simplify the code as shown in Listing 2, where `A` represents the condition `format == ThumbnailParameter.ORIGINAL_FORMAT` and `B` represents the condition `type == ThumbnailParameter.DEFAULT_FORMAT_TYPE` .

```
if (A && B) {
    return true;
} else if (A && !B) {
    return false;
} else if (B) {
    return true;
}
```

Listing 2: Simplified code analysis of Listing 1.

There are four possibilities for the values of `A` and `B`:

1. `A = T, B = T`: First branch evaluates to true and is taken, the rest of the conditional block is untested.
2. `A = T, B = F`: First branch evaluates to false second branch evaluates to true, so both `A` and `!B` are tested in the second branch.
3. `A = F, B = T`: Both the first and second branches evaluate to false by short-circuiting on the first condition, which means the second condition of the second branch (`!B`) does not get tested.
4. `A = F, B = F`: Both the first and second branches evaluate to false by short-circuiting on the first condition, which means the second condition of the second branch (`!B`) does not get tested.

As such, due to the structure of this conditional block and the short-circuiting behavior of Java conditional expressions, the second condition of the second statement will never be tested in the negative (when `!B = F`).

This highlights an opportunity for refactoring, which is discussed in the next section.

3.2 Refactoring Suggestions for ThumbnailatorUtils.java

Although no functional faults were found in `ThumbnailatorUtils.java`, code coverage analysis and inspection revealed areas that could benefit from refactoring to improve clarity and testability. These suggestions are also available as formal bug reports in Appendices A and B.

3.2.1 Redundant Conditions

In the method `isSupportedOutputFormatType(String format, String type)`, as shown in 1, the conditional block contains redundant and unreachable logic. By refactoring the code as shown in Listing 3, the clarity is improved and 100% code coverage is possible.

```
if (format == ThumbnailParameter.ORIGINAL_FORMAT
        && type == ThumbnailParameter.DEFAULT_FORMAT_TYPE) {
    return true;
```

```
} else if (format == ThumbnailParameter.ORIGINAL_FORMAT) {
    return false;
} else if (type == ThumbnailParameter.DEFAULT_FORMAT_TYPE) {
    return true;
}
```

Listing 3: Rafctoring suggestion of Listing 1 for improved clarity and code coverage.

3.2.2 Non-Null API Assumption

In the method `getSupportedOutputFormats()`, the following logic checks for a `null` return from `ImageIO.getWriterFormatNames()`, as shown in Listing 4.

```
public static List<String> getSupportedOutputFormats() {
    String[] formats = ImageIO.getWriterFormatNames();

    if (formats == null) {
        return Collections.emptyList();
    } else {
        return Arrays.asList(formats);
    }
}
```

Listing 4: Redundant null checks for ImageIO library.

According to the official Java documentation and observed runtime behavior, `ImageIO.getWriterFormatNames()` never returns `null`. Instead, it returns an empty array if no formats are found. The `null` check can only be tested through mocking, which introduces artificiality into test scenarios. Removing this condition (or replacing it with a check for an empty array) would simplify the logic and reduce reliance on mocks to achieve full coverage. This is shown in Listing 5.

```
public static List<String> getSupportedOutputFormats() {
    return Arrays.asList(ImageIO.getWriterFormatNames());
}
```

Listing 5: Refactored-out null checks for ImageIO library.

By addressing these areas, the codebase can be slightly streamlined without impacting functionality, while also improving maintainability and correspondence between tests and actual runtime behavior.

3.3 Testing Results of BufferedImageBuilder.java

All of the test cases we made for the `BufferedImageBuilder.java` class are passing, as shown in the IntelliJ screenshot in Figure 4.

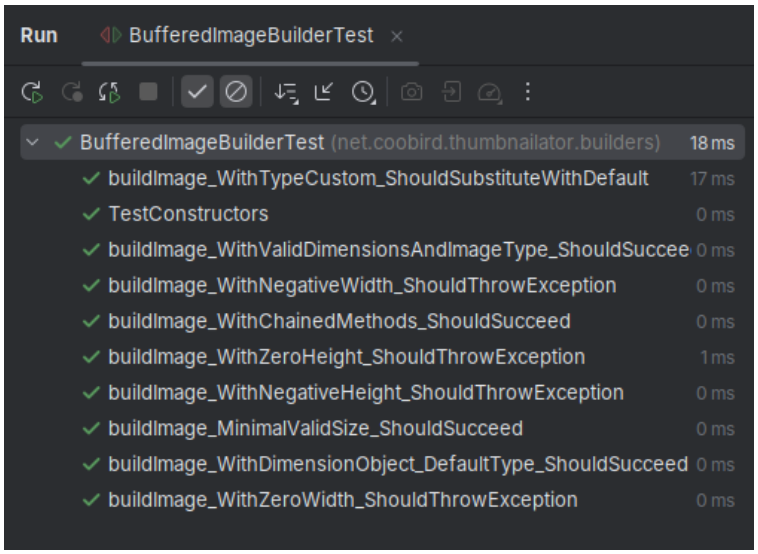
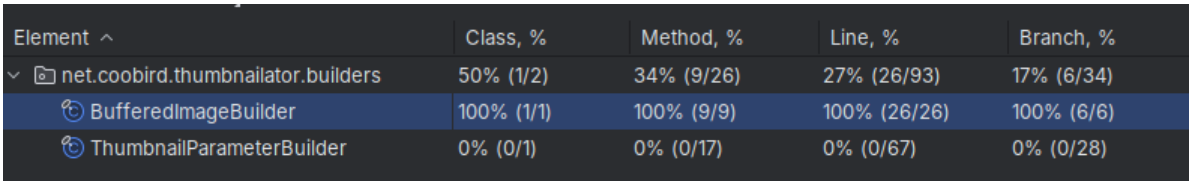


Figure 4: All test cases passing for BufferedImageBuilder.java.

No refactoring was deemed necessary for the `BufferedImageBuilder.java` class. This class adheres well to the Single Responsibility Principle, with its sole purpose being the creation of `BufferedImage` instances based on configurable parameters. The class design is straightforward, with clear constructor overloads and method chains for setting width, height, and image type. Input validation is explicitly handled through concise checks with informative exception messages, and the `build()` method reliably encapsulates object creation.

Given its focused scope, minimal complexity, and high readability, the class is already well-structured and easy to test. All logic paths are meaningful and reachable, and coverage was easily brought to 100% without requiring mocking or workaround strategies, as shown in Figure 5. As such, no structural or stylistic changes are recommended.



Element	Class, %	Method, %	Line, %	Branch, %
net.coobird.thumbnailator.builders	50% (1/2)	34% (9/26)	27% (26/93)	17% (6/34)
BufferedImageBuilder	100% (1/1)	100% (9/9)	100% (26/26)	100% (6/6)
ThumbnailParameterBuilder	0% (0/1)	0% (0/17)	0% (0/67)	0% (0/28)

Figure 5: 100% coverage for BufferedImageBuilder.java.

4 Use of Generative AI

Generative AI tools were incorporated throughout the testing process to improve coverage quality, identify refactoring opportunities, and streamline test development. These tools were used in a complementary role to human analysis, helping both accelerate and validate the design of the test suite.

4.1 AI Assistance for ThumbnailatorUtils.java

For the `ThumbnailatorUtils.java` class, AI tools were primarily used to apply more comprehensive coverage testing methods. Specifically, we consulted AI to explore advanced usage patterns of Mockito, including static mocking via `MockedStatic`, which was essential for simulating edge cases related to the ImageIO API. We were also advised by the AI tool to upgrade our Mockito dependency to version 4.11.0 and add the Mockito-inline version 4.11.0 dependency for static mocks. This guidance allowed us to test behavior that would otherwise be unreachable using standard input values alone, such as `ImageIO.getWriterFormatNames()` returning null, or mocked `ImageWriter` instances throwing `UnsupportedOperationException`.

Additionally, AI was used for providing refactoring suggestions based on unreachable branches identified through coverage analysis. After sharing specific blocks of logic, the tool helped confirm that certain conditions (e.g., unreachable branches in `isSupportedOutputFormatType`) were redundant and could be simplified. This aided in creating a more concise and maintainable version of the logic without altering functionality.

4.2 AI Assistance for BufferedImageBuilder.java

In the case of `BufferedImageBuilder.java`, AI was used for automated test case generation. After analyzing the class structure and its parameter constraints, we used AI to outline equivalence classes and generate a corresponding set of JUnit test cases. These included valid and invalid inputs for dimensions, special cases like `TYPE_CUSTOM`, and boundary conditions such as minimal valid size (1x1). This significantly accelerated the test writing process while ensuring methodical and consistent coverage across all input categories.

By using generative AI as a development partner, we were able to improve the efficiency, coverage, and depth of our testing effort, particularly in areas involving external dependencies and low-level control validation. This approach also ensured alignment with modern best practices in test-driven development and code quality assurance.

5 Conclusion

This report presented a comprehensive testing approach for two classes in the Thumbnailator library: `ThumbnailatorUtils.java` and `BufferedImageBuilder.java`, as assigned to Group F.

Using equivalence class testing as the primary method, we systematically identified and validated input-output behaviors across all relevant functional paths. For BufferedImageBuilder, testing resulted in 100% line and branch coverage, with no faults discovered and no need for refactoring. The class's simplicity, clear responsibilities, and input validation

made it ideal for this type of testing. For `ThumbnailatorUtils.java`, testing achieved 96% coverage, with all functional paths validated except for one unreachable branch. This led to targeted refactoring suggestions aimed at improving clarity and reducing dead code.

Generative AI tools were used strategically to enhance the testing process, from assisting in advanced Mockito usage and identifying refactoring opportunities to helping generate automated test cases for complex equivalence classes. These tools significantly improved testing depth and efficiency, particularly for edge cases involving external dependencies.

Overall, the testing strategies applied to both classes demonstrated a high level of coverage and confidence in correctness, while also showcasing the value of combining structured testing techniques with modern AI-assisted workflows.

Appendices

Appendix A Bug Report for isSupportedOutputFormat-Type Method of ThumbnailatorUtils Class

EECS 4313 ACTIVITY NAME: Final Project | PROBLEM REPORT #: 1

PROGRAM: ThumbnailatorUtils.java | RELEASE: 0 | VERSION: 0.4.20

REPORT TYPE (1-6): 3 - Suggestion | SEVERITY: 3 - Minor | ATTACHMENTS: ☐

IF ATTACHED STATE: _____

PROBLEM SUMMARY: The method isSupportedOutputFormatType(String format, String type) contains redundant conditional logic. Specifically, one of the conditional branches

CAN YOU REPRODUCE THE PROBLEM? Yes

PROBLEM AND HOW TO REPRODUCE IT: As identified via manual inspection and code coverage analysis, the second condition of the second branch unreachable.

SUGGESTED FIX (optional): Simplify the logic by removing or refactoring the unreachable condition, like removing the redundant `type != ThumbnailParameter.DEFAULT_FORMAT_TYPE` condition in 2nd branch.

REPORTED BY: Group F | DATE: 4/7/25

FOR DEVELOPMENT TEAM ONLY

FUNCTIONAL AREA: _____ | ASSIGNED TO: _____

COMMENTS:

STATUS: _____ | PRIORITY: _____ | RESOLUTION (1-9): _____

RESOLVED BY: _____ | DATE: 4/7/25

RESOLUTION TESTED BY: _____ | DATE: 4/7/25

RESOLUTION VERSION: 0 | TREAT AS DEFERRED: ☐

Appendix B Bug Report for getSupportedOutputFormats Method of ThumbnailatorUtils Class

EECS 4313 ACTIVITY NAME: Final Project

PROBLEM REPORT #: 2

PROGRAM: ThumbnailatorUtils.java

RELEASE: 0

VERSION: 0.4.20

REPORT TYPE (1-6): 3 - Suggestion

SEVERITY: 3 - Minor

ATTACHMENTS:

IF ATTACHED STATE:

PROBLEM SUMMARY: The method getSupportedOutputFormats() checks for a null return from ImageIO.getWriterFormatNames(), which is not consistent with the Java API.

CAN YOU REPRODUCE THE PROBLEM? Yes

PROBLEM AND HOW TO REPRODUCE IT: As identified via inspection of the Java API and official Java documentation, ImageIO.getWriterFormatNames() will never return null, so `if (formats == null)` is always true.

SUGGESTED FIX (optional): Remove the null check and replace with a simpler, more representative implementation, like `return Arrays.asList(ImageIO.getWriterFormatNames());`

REPORTED BY: Group F

DATE: 4/7/25

FOR DEVELOPMENT TEAM ONLY

FUNCTIONAL AREA:

ASSIGNED TO:

COMMENTS:

STATUS:

PRIORITY:

RESOLUTION (1-9):

RESOLVED BY:

DATE: 4/7/25

RESOLUTION TESTED BY:

DATE: 4/7/25

RESOLUTION VERSION: 0

TREAT AS DEFERRED: