Testing and Quality Analysis of commons-imaging Project

`GitHub Repository`

Nima Mehranfar

January 7, 2025

Abstract

This report presents the testing and quality analysis conducted on the commons-imaging project hosted on GitHub. The project undergoes thorough software quality analysis, CI/CD pipeline setup, code coverage analysis, performance testing, mutation testing, security assessment, and more. The results of these analyses are presented along with the actions taken to improve the quality of the commons-imaging codebase.

Contents

List of Figures

List of Tables

1 Introduction

The *commons-imaging* project is a Java library developed by the Apache Software Foundation. It provides a comprehensive API for working with various image formats, enabling developers to read, write, and manipulate images with ease. This project focuses on evaluating the *commons-imaging* codebase through a rigorous quality assurance process to ensure reliability, maintainability, and performance.

1.1 Objectives

This testing project aims to achieve the following objectives:

- Establish CI/CD pipelines for the *commons-imaging* project using GitHub Actions.
- Perform software quality analysis leveraging SonarCloud for static code evaluation and reporting.
- Develop a Docker image to facilitate streamlined deployment and testing environments.
- Conduct code coverage analysis using JaCoCo to identify untested areas in the codebase.
- Apply mutation testing with PiTest to evaluate the robustness of existing test cases.
- Execute stress testing on project components with JMH to assess performance under load.
- Generate automated test cases for inadequately tested code segments using Randoop.
- Perform security analysis using Snyk to detect and mitigate potential vulnerabilities.

2 CI/CD Pipeline

The *commons-imaging* project has been integrated into a CI/CD pipeline that can be built both locally and on cloud servers, including GitHub. This pipeline automates the build, test, and deployment processes to ensure continuous integration and delivery, enabling efficient and reliable software updates.

2.1 CI/CD Configuration

The CI/CD pipeline configuration utilizes GitHub Actions to automate the workflow. Below is the YAML file for the GitHub Actions workflow configuration:

Listing 2.1: GitHub Actions Workflow Configuration (`maven.yml`)

```
name: Java CI

on: [push, pull_request, workflow_dispatch]

permissions:
  contents: read

jobs:
  build:
    runs-on: ${{ matrix.os }}
    continue-on-error: ${{ matrix.experimental }}
    strategy:
      matrix:
        os: [windows-latest, macos-13, ubuntu-latest]
        java: [8, 11, 17, 21]
        experimental: [false]
        include:
            - java: 23
              os: ubuntu-latest
              experimental: false
            - java: 24-ea
              os: ubuntu-latest
              experimental: true

    steps:
    # Configure Git settings to avoid line-ending issues
    - name: Prepare git
      run: |-
        git config --global core.autocrlf false
        git config --global core.eol lf
    - uses: actions/checkout@v2  # Checkout the repository code
      with:
        persist-credentials: false
    - name: Set up JDK ${{ matrix.java }}
      uses: actions/setup-java@v2  # Setup JDK version
      with:
        distribution: 'temurin'
        java-version: ${{ matrix.java }}
    - name: Build with Maven
      run: mvn --errors --show-version --batch-mode --no-transfer-progress
```

3 Software Quality Analysis with SonarCloud

The software quality analysis of the *commons-imaging* project is conducted using SonarCloud, a powerful tool that identifies potential code quality issues and provides recommendations for improvements. By integrating SonarCloud into the development workflow, developers can continuously monitor and address various code quality aspects to ensure a robust and maintainable codebase.

3.1 Issues Categorization

SonarCloud categorizes the issues it identifies into three main types:

- **Maintainability**: These refer to unnecessary complexity or suboptimal code patterns that may not be bugs but could lead to maintenance challenges.
- **Reliability**: Errors in the code that could lead to functional failures or unexpected behavior.
- **Security**: Potential security risks within the code that could be exploited by malicious actors.

3.2 Refactoring and Rationale

Several refactoring actions were undertaken to address the issues identified by SonarCloud. For those issues that were skipped, a clear rationale was provided.
As illustrated in the image below (Figure 3.1), a total of 2325 issues were found, including 18 critical "blocker" severity issues and 333 "high" severity issues. All the "blocker" issues were maintainability; And I prioritized addressing some of the these issues to improve the overall code quality.

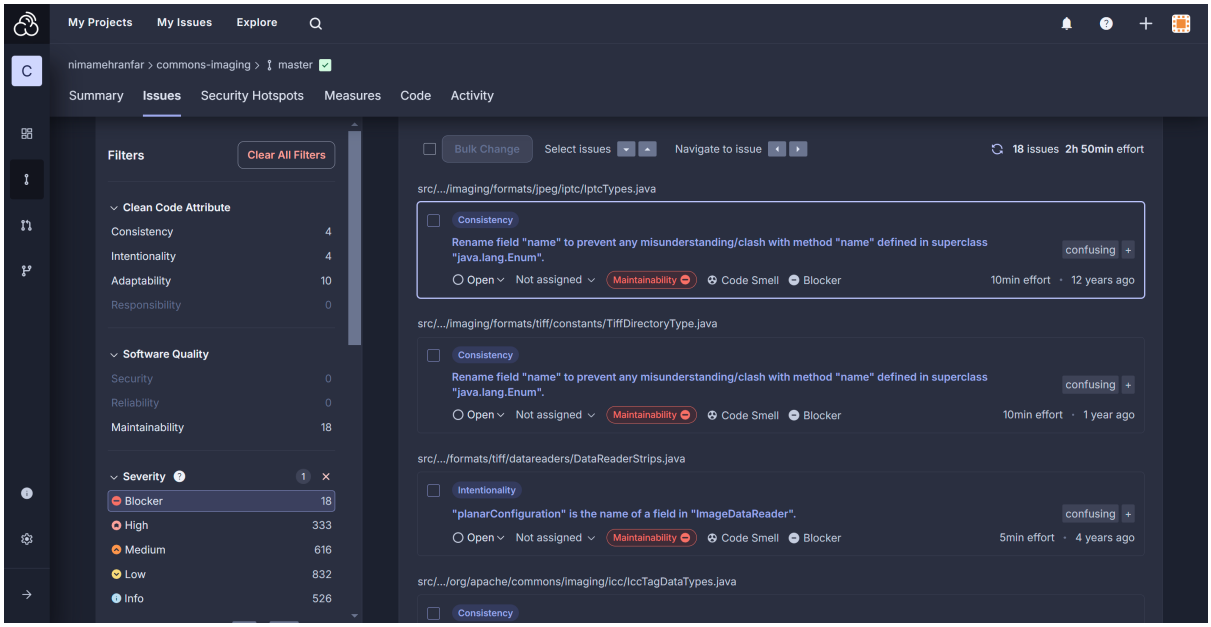


Figure 3.1: SonarCloud Issues Overview: 2325 issues detected, including blocker and high severity issues.

Some of the common issues included a lack of assertions in test cases, confusing or duplicate method/variable names and missing break in switch cases, as demonstrated in the images below (Figures 3.2, 3.3, 3.4).

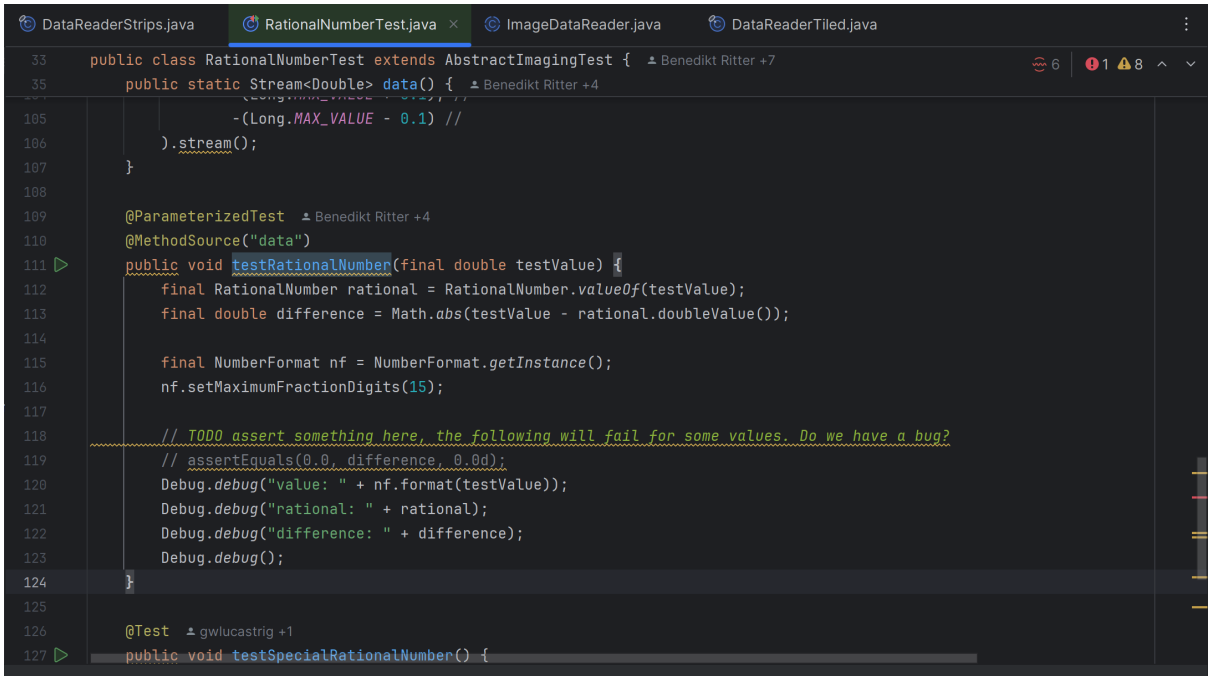


Figure 3.2: Example of missing assertions in test cases.

Figure 3.3: Example of confusing or duplicate method/variable names.

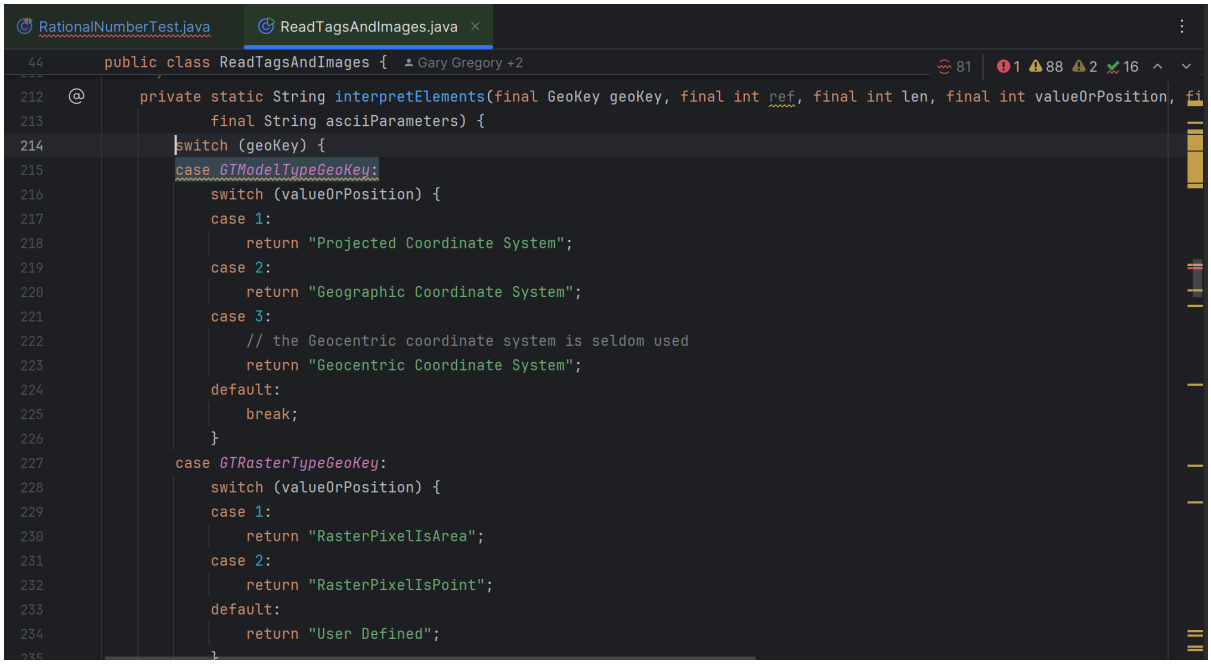


Figure 3.4: Example of missing break in switch case.

As shown in Figure 3.5, I successfully resolved 9 issues, but during the process, I also

introduced 1 additional issue which is related to commented-out lines of code in the failing test case assertion. This occurred due to a test case that revealed a bug in the code (as noted by the developer). The addition of assertions caused the build process to halt, which is evident in the error log below.

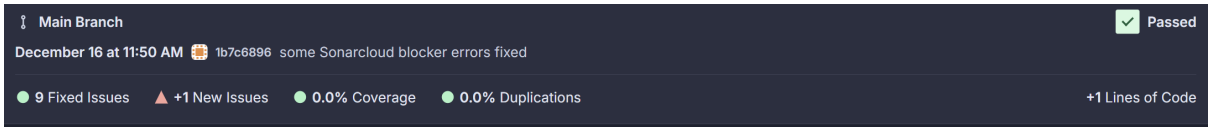


Figure 3.5: 9 issues fixed

The error message below demonstrates the issue with floating-point precision during the test execution:

```
Assertion Error: \\
[ERROR] Failures: \\
[ERROR]   RationalNumberTest.testRationalNumber:120 The difference between the test
    ↪ value and the rational number representation exceeds the tolerance. \\
      ==> expected: <0.0> but was: <0.0020000040531158447> \\
[INFO] \\
[ERROR] Tests run: 1076, Failures: 15, Errors: 0, Skipped: 7 \\
[INFO]

Patterns in Failures: \\
Small differences (<1e-9) might be acceptable due to floating-point arithmetic. \\
Large differences (e.g., 0.09999990463256836 or 9.223372034707292E18) are likely bugs
    ↪ or limitations in RationalNumber handling. \\
Therefore, the implementation of \texttt{rationalNumber.valueOf()} must be reviewed
    ↪ by the developer. \\
This problem was tagged as "Accepted" on SonarQube.
```

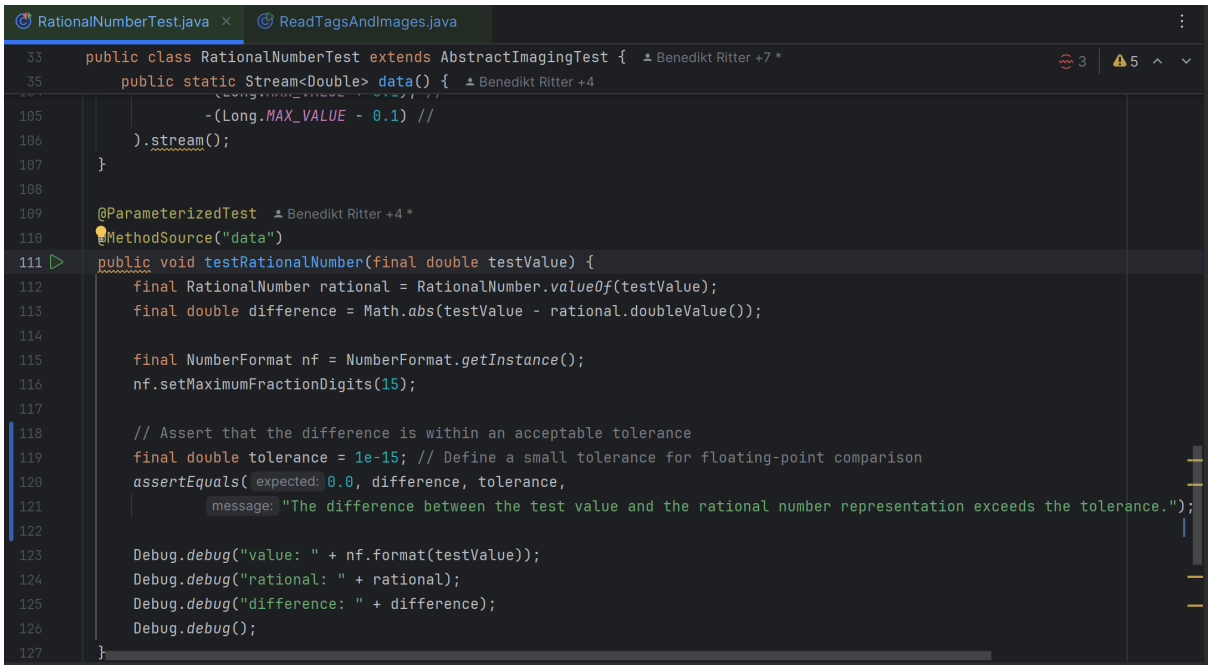


Figure 3.6: Build halted due to failed assertions and floating-point precision issue of this test case.

The issues related to floating-point precision need careful attention, as small discrepancies in numerical computations are often acceptable, but larger discrepancies indicate potential bugs. The developer has flagged the issue for further investigation; So I tagged this issue as "Accepted" on SonarQube.

Also in (Figures 3.7, 3.8), there is an issue labeled "child class fields should not shadow parent class fields" which is unfixable because the parent class field is a protected variable and cannot be accessed by the child class so any change to both the naming and/or syntax/keywords used for variables could result in an unwanted scenario and/or a bug. So I tagged this issue as "accepted" too.

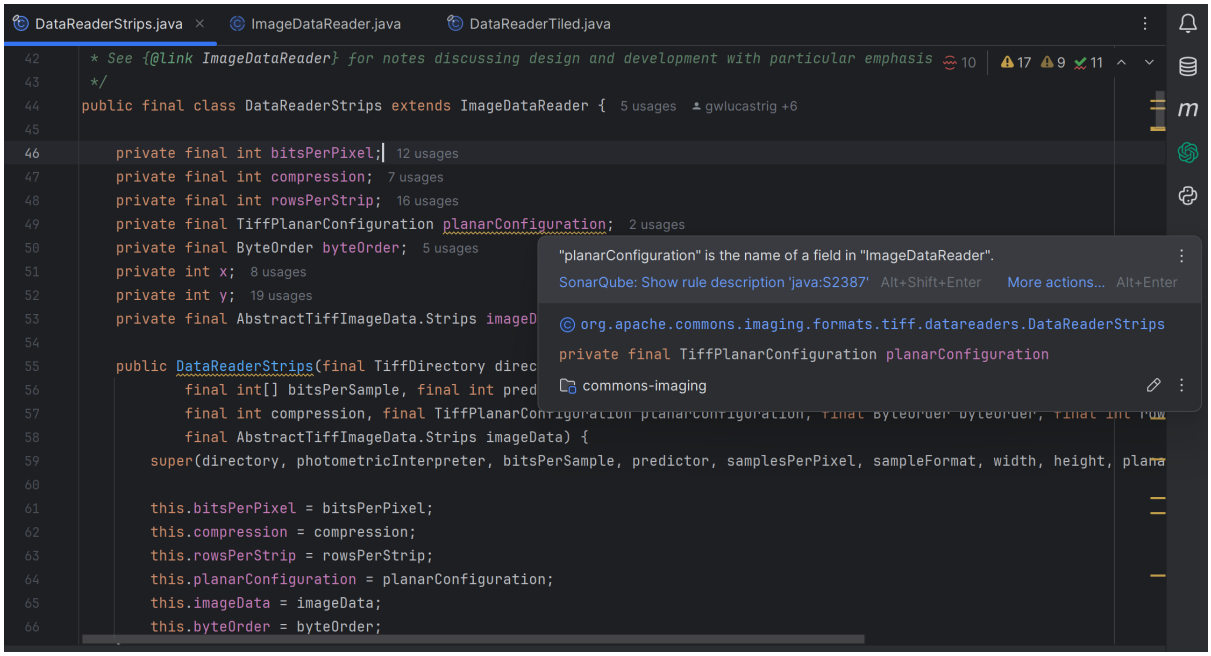


Figure 3.7: Unfixable naming issue

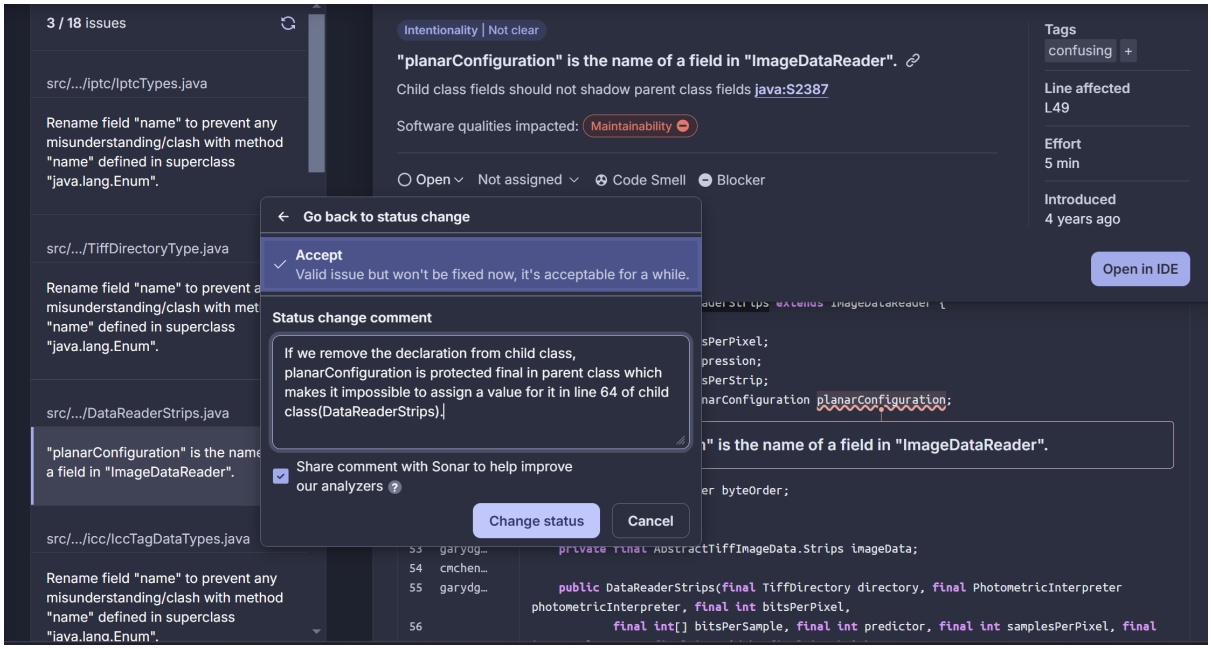


Figure 3.8: Unfixable naming issue tagged as ”accepted”

4 Docker Image Creation and Containerization

The *commons-imaging* project, being an API and utility without a main method, required an additional web application for demonstration purposes. A simple web application was created using React.js for the frontend and Java Spring Boot for the backend. This setup allows the utility to be integrated into a functional system for testing and deployment.

4.1 Docker Image and Containerization

Two Docker images were created for the web application's frontend and backend. These images were pushed to DockerHub and configured to run together using Docker Compose. This ensures seamless orchestration and ease of deployment.

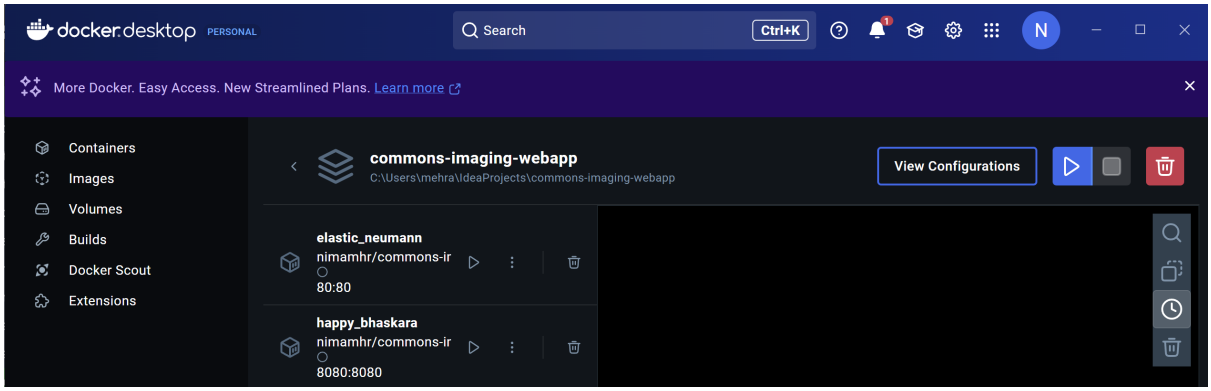


Figure 4.1: Docker Architecture for the commons-imaging-webapp.

4.2 Dockerfile for Frontend

The Dockerfile for the frontend builds a React.js application and serves it using Nginx. Below is the configuration:

Listing 4.1: Dockerfile for commons-imaging-webapp Frontend

```
FROM node:18-alpine AS build
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm install
COPY . ./
RUN npm run build
FROM nginx:alpine
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

4.3 Dockerfile for Backend

The backend is a Java Spring Boot application. The Dockerfile compiles the application and creates an executable JAR file:

Listing 4.2: Dockerfile for commons-imaging-webapp Backend

```
FROM openjdk:24-jdk
WORKDIR /app
COPY .mvn/ .mvn
COPY mvnw pom.xml ./
RUN ./mvnw dependency:go-offline -B
COPY src/ ./src
RUN ./mvnw clean package -DskipTests
EXPOSE 8080
CMD ["java", "-jar", "target/test-0.0.1-SNAPSHOT.jar"]
```

4.4 Docker Compose Configuration

The Docker Compose file orchestrates both the frontend and backend services, setting up a shared network and dependencies.

Listing 4.3: Docker Compose File for Orchestration

```
version: '3.7'

services:
  backend:
    build:
      context: ./Backend
    ports:
      - "8080:8080"
    networks:
      - app-network

  frontend:
    build:
      context: ./commons-imaging-test
    ports:
      - "80:80"
    environment:
      - REACT_APP_API_URL=http://backend:8080  # API URL for backend
    networks:
      - app-network
    depends_on:
      - backend

networks:
  app-network:
    driver: bridge
```

4.5 DockerHub Synchronization

The Docker images for both the frontend and backend are synchronized with DockerHub for easy access. They can be pulled and run using the following commands:

4.5.1 Pulling Docker Images

Listing 4.4: Pull Docker Images from DockerHub

```
docker pull nimamhr/commons-imaging-webapp-frontend:latest
docker pull nimamhr/commons-imaging-webapp-backend:latest
```

4.5.2 Running Docker Images

Listing 4.5: Run Docker Images

```
docker run -d -p 8080:8080 nimamhr/commons-imaging-webapp-backend:latest
docker run -d -p 80:80 nimamhr/commons-imaging-webapp-frontend:latest
```

4.5.3 Accessing the Web Application

Once the containers are running, the web application can be accessed at: `http://localhost/`

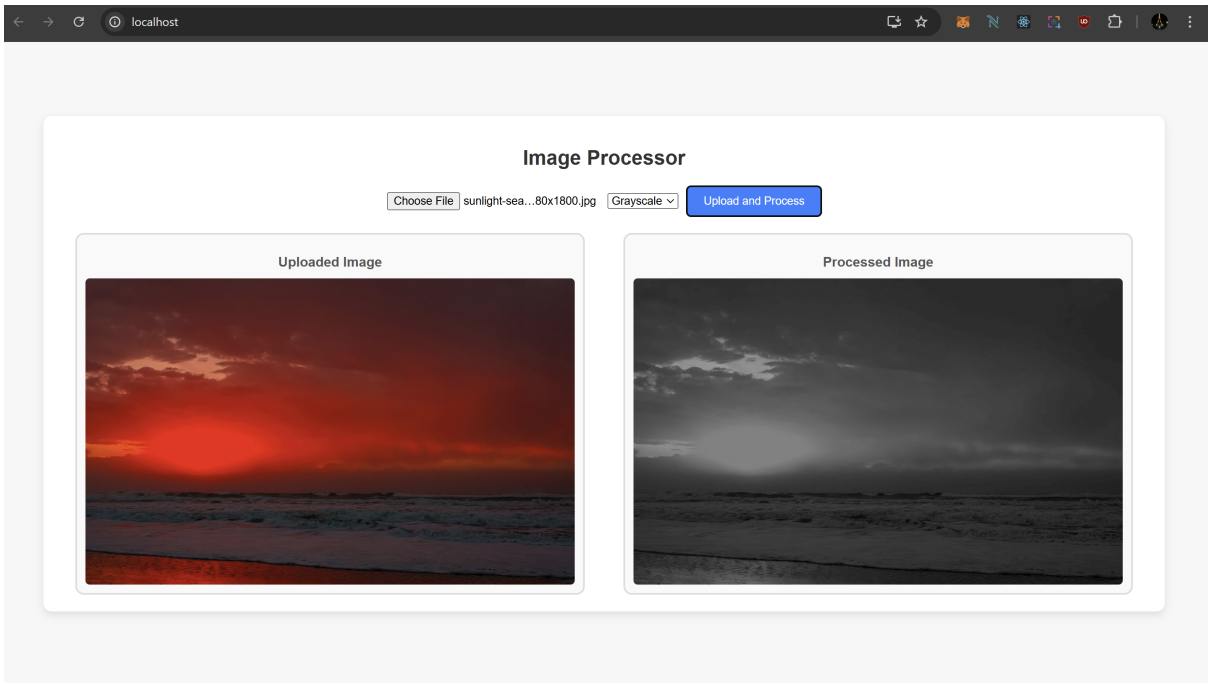


Figure 4.2: Commons-Imaging Test Web Application Interface.

5 Code Coverage Analysis

Code coverage analysis was performed using JaCoCo to assess the extent of test coverage within the *commons-imaging* project. This analysis helped identify untested or poorly tested portions of the codebase, guiding the addition of manual and automated tests.

5.1 Setting Up JaCoCo

To set up JaCoCo for the project, the JaCoCo Maven plugin was added to the `pom.xml` file. The following commands were then executed to generate code coverage reports:

Listing 5.1: Commands to Generate Code Coverage Reports

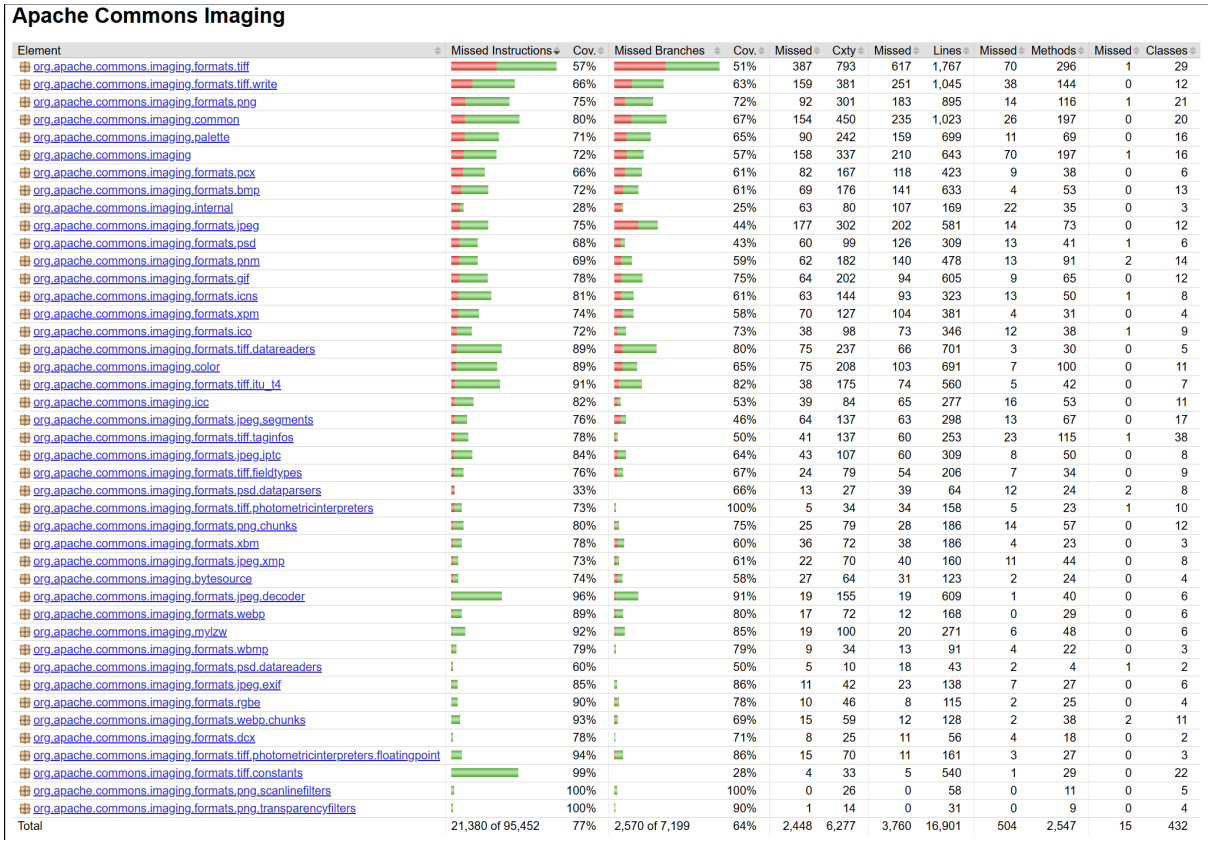
```
mvn clean test
mvn clean verify
```

5.2 Initial Coverage Results

The initial code coverage results, before adding tests, are summarized below:

Element	Missed Instructions	I. Cov. (%)	Missed Branches	B. Cov. (%)
Total	21,380 of 95,452	77%	2,570 of 7,199	64%

Table 5.1: Initial Code Coverage Results.



Apache Commons Imaging

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.imaging.formats.tiff		57%		51%	387	793	617	1,767	70	296	1	29
org.apache.commons.imaging.formats.tiff.write		66%		63%	159	381	251	1,045	38	144	0	12
org.apache.commons.imaging.formats.png		75%		72%	92	301	183	895	14	116	1	21
org.apache.commons.imaging.common		80%		67%	154	450	235	1,023	26	197	0	20
org.apache.commons.imaging.palette		71%		65%	90	242	159	699	11	69	0	16
org.apache.commons.imaging		72%		57%	158	337	210	643	70	197	1	16
org.apache.commons.imaging.formats.pcx		66%		61%	82	167	118	423	9	38	0	6
org.apache.commons.imaging.formats.bmp		72%		61%	69	176	141	633	4	53	0	13
org.apache.commons.imaging.internal		28%		25%	63	80	107	169	22	35	0	3
org.apache.commons.imaging.formats.jpeg		75%		44%	177	302	202	581	14	73	0	12
org.apache.commons.imaging.formats.psd		68%		43%	60	99	126	309	13	41	1	6
org.apache.commons.imaging.formats.pnm		69%		59%	62	182	140	478	13	91	2	14
org.apache.commons.imaging.formats.gif		78%		75%	64	202	94	605	9	65	0	12
org.apache.commons.imaging.formats.icns		81%		61%	63	144	93	323	13	50	1	8
org.apache.commons.imaging.formats.xpm		74%		58%	70	127	104	381	4	31	0	4
org.apache.commons.imaging.formats.ico		72%		73%	38	98	73	346	12	38	1	9
org.apache.commons.imaging.formats.tiff.datareaders		89%		80%	75	237	66	701	3	30	0	5
org.apache.commons.imaging.color		89%		65%	75	208	103	691	7	100	0	11
org.apache.commons.imaging.formats.tiff.itu_t4		91%		82%	38	175	74	560	5	42	0	7
org.apache.commons.imaging.icc		82%		53%	39	84	65	277	16	53	0	11
org.apache.commons.imaging.formats.jpeg.segments		76%		46%	64	137	63	298	13	67	0	17
org.apache.commons.imaging.formats.tiff.taginfos		78%		50%	41	137	60	253	23	115	1	38
org.apache.commons.imaging.formats.jpeg.iptc		84%		64%	43	107	60	309	8	50	0	8
org.apache.commons.imaging.formats.tiff.fieldtypes		76%		67%	24	79	54	206	7	34	0	9
org.apache.commons.imaging.formats.psd.dataparsers		33%		66%	13	27	39	64	12	24	2	8
org.apache.commons.imaging.formats.tiff.photometricinterpreters		73%		100%	5	34	34	158	5	23	1	10
org.apache.commons.imaging.formats.png.chunks		80%		75%	25	79	28	186	14	57	0	12
org.apache.commons.imaging.formats.xbm		78%		60%	36	72	38	186	4	23	0	3
org.apache.commons.imaging.formats.jpeg.xmp		73%		61%	22	70	40	160	11	44	0	8
org.apache.commons.imaging.bytesource		74%		58%	27	64	31	123	2	24	0	4
org.apache.commons.imaging.formats.jpeg.decoder		96%		91%	19	155	19	609	1	40	0	6
org.apache.commons.imaging.formats.webp		89%		80%	17	72	12	168	0	29	0	6
org.apache.commons.imaging.mylzw		92%		85%	19	100	20	271	6	48	0	6
org.apache.commons.imaging.formats.wbmp		79%		79%	9	34	13	91	4	22	0	3
org.apache.commons.imaging.formats.psd.datareaders		60%		50%	5	10	18	43	2	4	1	2
org.apache.commons.imaging.formats.jpeg.exif		85%		86%	11	42	23	138	7	27	0	6
org.apache.commons.imaging.formats.rgbe		90%		78%	10	46	8	115	2	25	0	4
org.apache.commons.imaging.formats.webp.chunks		93%		69%	15	59	12	128	2	38	2	11
org.apache.commons.imaging.formats.dcx		78%		71%	8	25	11	56	4	18	0	2
org.apache.commons.imaging.formats.tiff.photometricinterpreters.floatingpoint		94%		86%	15	70	11	161	3	27	0	3
org.apache.commons.imaging.formats.tiff.constants		99%		28%	4	33	5	540	1	29	0	22
org.apache.commons.imaging.formats.png.scanlinefilters		100%		100%	0	26	0	58	0	11	0	5
org.apache.commons.imaging.formats.png.transparencyfilters		100%		90%	1	14	0	31	0	9	0	4
Total	21,380 of 95,452	77%	2,570 of 7,199	64%	2,448	6,277	3,760	16,901	504	2,547	15	432

Figure 5.1: Visualization of Initial Code Coverage Results.

5.3 Final Coverage Results After Test Improvements

After generating additional tests using automated tools and adding manual test cases (details are in chapters 6 & 8), the code coverage improved. The final coverage results are summarized below:

Element	Missed Instructions	I. Cov. (%)	Missed Branches	B. Cov. (%)
Total	18,077 of 95,452	81%	2,327 of 7,199	67%

Table 5.2: Final Code Coverage Results After Test Improvements.

Apache Commons Imaging

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.imaging.formats.tiff		69%		62%	282	793	415	1,767	14	296	1	29
org.apache.commons.imaging.formats.png		75%		72%	92	301	183	895	14	116	1	21
org.apache.commons.imaging.palette		71%		65%	90	242	159	699	11	69	0	16
org.apache.commons.imaging.common		82%		68%	142	450	213	1,023	18	197	0	20
org.apache.commons.imaging.formats.pcx		66%		61%	82	167	118	423	9	38	0	6
org.apache.commons.imaging		76%		58%	145	337	170	643	56	197	0	16
org.apache.commons.imaging.formats.bmp		72%		61%	69	176	141	633	4	53	0	13
org.apache.commons.imaging.formats.tiff.write		85%		77%	96	381	101	1,045	3	144	0	12
org.apache.commons.imaging.formats.jpeg		75%		44%	177	302	202	581	14	73	0	12
org.apache.commons.imaging.formats.psd		68%		43%	60	99	126	309	13	41	1	6
org.apache.commons.imaging.formats.pnm		69%		59%	62	182	140	478	13	91	2	14
org.apache.commons.imaging.formats.gif		78%		75%	64	202	94	605	9	65	0	12
org.apache.commons.imaging.formats.icns		81%		61%	63	144	93	323	13	50	1	8
org.apache.commons.imaging.formats.xpm		74%		58%	70	127	104	381	4	31	0	4
org.apache.commons.imaging.internal		48%		44%	46	80	74	169	13	35	0	3
org.apache.commons.imaging.formats.ico		72%		73%	38	98	73	346	12	38	1	9
org.apache.commons.imaging.formats.tiff.datareaders		89%		80%	74	237	65	701	3	30	0	5
org.apache.commons.imaging.icc		82%		53%	39	84	65	277	16	53	0	11
org.apache.commons.imaging.formats.jpeg.segments		76%		46%	64	137	63	298	13	67	0	17
org.apache.commons.imaging.color		92%		65%	74	208	81	691	6	100	0	11
org.apache.commons.imaging.formats.jpeg.iptc		84%		64%	43	107	60	309	8	50	0	8
org.apache.commons.imaging.formats.psd.dataparsers		35%		66%	9	27	35	64	8	24	0	8
org.apache.commons.imaging.formats.png.chunks		80%		75%	25	79	28	186	14	57	0	12
org.apache.commons.imaging.formats.xbm		78%		60%	36	72	38	186	4	23	0	3
org.apache.commons.imaging.formats.tiff.fieldtypes		82%		80%	19	79	38	206	3	34	0	9
org.apache.commons.imaging.formats.tiff.itu_t4		95%		89%	27	175	35	560	4	42	0	7
org.apache.commons.imaging.formats.jpeg.xmp		73%		61%	22	70	40	160	11	44	0	8
org.apache.commons.imaging.bytesource		74%		58%	27	64	31	123	2	24	0	4
org.apache.commons.imaging.formats.jpeg.decoder		96%		91%	19	155	19	609	1	40	0	6
org.apache.commons.imaging.formats.tiff.taginfos		91%		61%	16	137	27	253	2	115	0	38
org.apache.commons.imaging.formats.webp		89%		80%	17	72	12	168	0	29	0	6
org.apache.commons.imaging.mylzw		92%		85%	19	100	20	271	6	48	0	6
org.apache.commons.imaging.formats.wbmp		79%		79%	9	34	13	91	4	22	0	3
org.apache.commons.imaging.formats.psd.datareaders		60%		50%	5	10	18	43	2	4	1	2
org.apache.commons.imaging.formats.jpeg.exif		85%		86%	11	42	23	138	7	27	0	6
org.apache.commons.imaging.formats.rgbe		90%		78%	10	46	8	115	2	25	0	4
org.apache.commons.imaging.formats.webp.chunks		93%		69%	15	59	12	128	2	38	2	11
org.apache.commons.imaging.formats.dcx		78%		71%	8	25	11	56	4	18	0	2
org.apache.commons.imaging.formats.tiff.photometricinterpreters.floatingpoint		95%		90%	11	70	9	161	3	27	0	3
org.apache.commons.imaging.formats.tiff.photometricinterpreters		96%		100%	0	34	8	158	0	23	0	10
org.apache.commons.imaging.formats.tiff.constants		100%		100%	0	33	0	540	0	29	0	22
org.apache.commons.imaging.formats.png.scanlinefilters		100%		100%	0	26	0	58	0	11	0	5
org.apache.commons.imaging.formats.png.transparencyfilters		100%		90%	1	14	0	31	0	9	0	4
Total	18,077 of 95,452	81%	2,327 of 7,199	67%	2,178	6,277	3,165	16,901	345	2,547	10	432

Figure 5.2: Visualization of Final Code Coverage Results.

The results demonstrate some improvements in test coverage and a reduction in missed methods and classes, indicating better validation of the codebase.

6 Mutation Testing with PiTest

Mutation testing is performed to evaluate the effectiveness of the test cases by introducing small modifications (mutations) to the code and ensuring that the tests can detect the changes.
After adding the necessary plugins and dependencies, the following command was run to initiate the mutation testing process:

Listing 6.1: Mutation Testing Command

```
mvn org.pitest:pitest-maven:mutationCoverage
```

6.1 Mutation Test Results

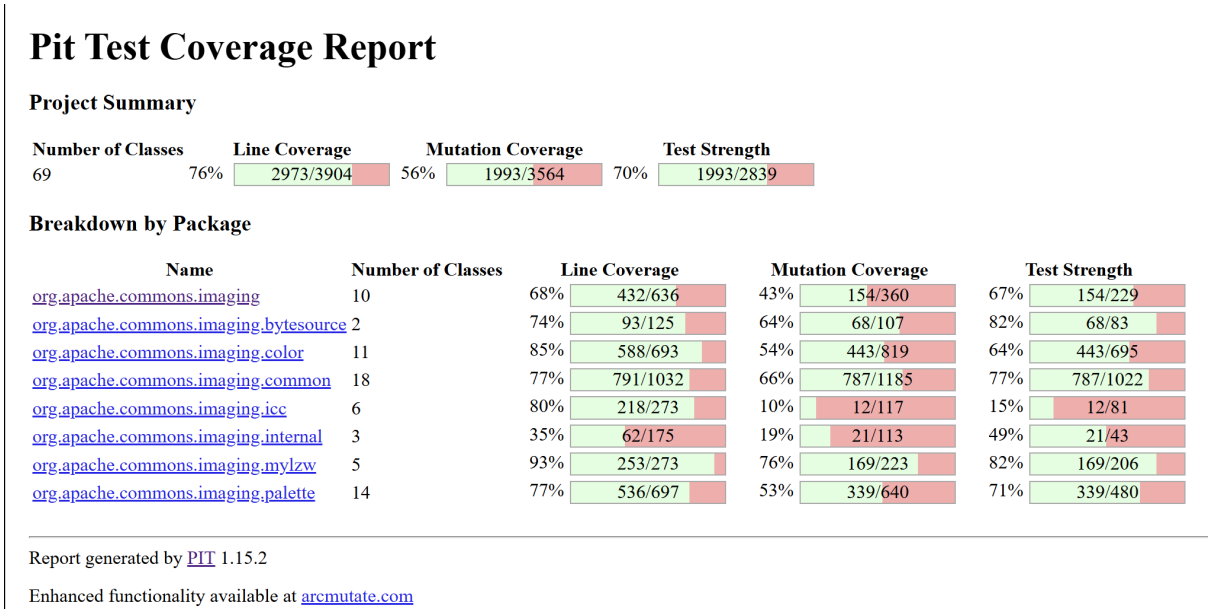
The overall mutation test results for the commons-imaging project are as follows:

Metric	Classes	Line Cov.	Mutation Cov.	Survived Mut.	Test Strength
Overall	69	76% (2973/3904)	56% (1993/3564)	44% (1571/3564)	70% (1993/2839)

Table 6.1: Mutation Test Results Overview.

The number of survived mutants for the whole project is calculated as:

$$\text{Survived Mutants} = \text{Total Mutants} - \text{Killed Mutants}$$
$$\text{Survived Mutants} = 3564 - 1993 = 1571$$



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
69	76%	2973/3904	56%	1993/3564	70%	1993/2839

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.apache.commons.imaging	10	68%	432/636	43%	154/360	67%	154/229
org.apache.commons.imaging.bytesource	2	74%	93/125	64%	68/107	82%	68/83
org.apache.commons.imaging.color	11	85%	588/693	54%	443/819	64%	443/695
org.apache.commons.imaging.common	18	77%	791/1032	66%	787/1185	77%	787/1022
org.apache.commons.imaging.icc	6	80%	218/273	10%	12/117	15%	12/81
org.apache.commons.imaging.internal	3	35%	62/175	19%	21/113	49%	21/43
org.apache.commons.imaging.mylzw	5	93%	253/273	76%	169/223	82%	169/206
org.apache.commons.imaging.palette	14	77%	536/697	53%	339/640	71%	339/480

Report generated by PIT 1.15.2

Enhanced functionality available at arcmutate.com

Figure 6.1: Visualization of PIT Coverage Results.

6.2 Detailed Analysis: ColorTools.java

The results for `ColorTools.java` before and after adding additional tests (`ColorToolsTest.java`) were as follows:

Metric	Classes	Line Cov.	Mutation Cov.	Survived Mut.	Test Strength
Before	1	0% (0/62)	0% (0/29)	100% (29/29)	Undefined (0/0)
After	1	60% (37/62)	62% (18/29)	38% (11/29)	90% (18/20)

Table 6.2: Mutation Test Results for `ColorTools.java`.

The following images show the progress made in terms of test coverage for `ColorTools.java`:

Figure 6.2: Initial Coverage and Mutation Results for `ColorTools.java`.

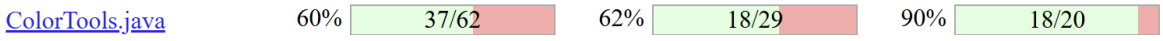


Figure 6.3: Improved Coverage and Mutation Results for `ColorTools.java`.

While 100% line coverage wasn't achieved, this improvement shows a significant increase in coverage, reducing the number of surviving mutants and improving overall test strength.

7 Performance Testing with JMH

Performance tests are implemented using JMH to identify performance bottlenecks in the commons-imaging project.

7.1 Benchmark Setup

The directory containing the benchmarks is located at:

`../src/test/java/org/apache/commons/imaging/benchmark`

The following files are part of the benchmark setup:

- `ImagingBenchmark.java`
- `sample.jpg`
- `sample_baseline.jpg`
- `sample.png`
- `sample.tiff`
- `sample.bmp`

The benchmarks were run by first executing the following Maven command:

```
mvn clean install
```

Then, the `ImagingBenchmark` class was run with the following configuration:

Listing 7.1: ImagingBenchmark-configuration

```
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Thread)
@Fork(1)
@Warmup(iterations = 3)
@Measurement(iterations = 5)
```

7.2 Stress Testing Results

The most resource-intensive components were stress-tested using JMH, and the results are summarized in the table below:

Benchmark	Mode	Score	Units
`ImagingBenchmark.testBmpDecoding`	avgt	1.157	ms/op
`ImagingBenchmark.testBmpImageInfoRetrieval`	avgt	0.300	ms/op
`ImagingBenchmark.testBmpImageWriting`	avgt	8.348	ms/op
`ImagingBenchmark.testImageInfoRetrieval`	avgt	0.476	ms/op
`ImagingBenchmark.testImageResizing`	avgt	36.858	ms/op
`ImagingBenchmark.testJpegDecoding`	avgt	212.311	ms/op
`ImagingBenchmark.testPngEncoding`	avgt	132.896	ms/op
`ImagingBenchmark.testTiffMetadataExtraction`	avgt	1.217	ms/op

Table 7.1: Stress Testing Results of Commons-Imaging Project.

```
Benchmark                                    Mode  Cnt    Score   Error  Units
ImagingBenchmark.testBmpDecoding             avgt         1.157          ms/op
ImagingBenchmark.testBmpImageInfoRetrieval   avgt         0.300          ms/op
ImagingBenchmark.testBmpImageWriting         avgt         8.348          ms/op
ImagingBenchmark.testImageInfoRetrieval      avgt         0.476          ms/op
ImagingBenchmark.testImageResizing           avgt        36.858          ms/op
ImagingBenchmark.testJpegDecoding            avgt       212.311          ms/op
ImagingBenchmark.testPngEncoding             avgt       132.896          ms/op
ImagingBenchmark.testTiffMetadataExtraction  avgt         1.217          ms/op

Process finished with exit code 0
```

Figure 7.1: Benchmark Results of Selected Methods of Commons-Imaging Project.

8 Automated Test Generation

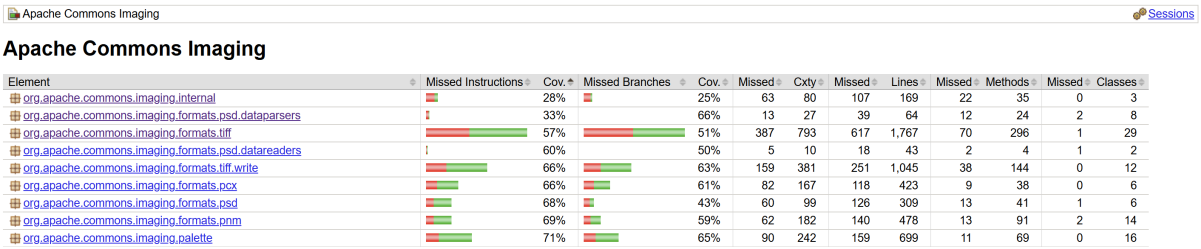
Automated tests were generated using Randoop to improve coverage on poorly tested components. These tests focus on edge cases and scenarios not previously covered.

8.1 Low Coverage Packages

From the JaCoCo report, we identified that three packages had low coverage:

- `internal`
- `formats/psd/dataparsers`
- `formats/tiff`

The report also highlights the classes within these packages that need additional test coverage, as shown in the image below:


Apache Commons Imaging Sessions

Apache Commons Imaging

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.imaging.internal		28%		25%	63	80	107	169	22	35	0	3
org.apache.commons.imaging.formats.psd.dataparsers		33%		66%	13	27	39	64	12	24	2	8
org.apache.commons.imaging.formats.tiff		57%		51%	387	793	617	1,767	70	296	1	29
org.apache.commons.imaging.formats.psd.datareaders		60%		50%	5	10	18	43	2	4	1	2
org.apache.commons.imaging.formats.tiff.write		66%		63%	159	381	251	1,045	38	144	0	12
org.apache.commons.imaging.formats.pcx		66%		61%	82	167	118	423	9	38	0	6
org.apache.commons.imaging.formats.psd		68%		43%	60	99	126	309	13	41	1	6
org.apache.commons.imaging.formats.pnm		69%		59%	62	182	140	478	13	91	2	14
org.apache.commons.imaging.palette		71%		65%	90	242	159	699	11	69	0	16

Figure 8.1: JaCoCo report showing low coverage in specific packages.

8.2 Test Generation Process

To generate tests for all the classes within these packages, we first listed the classes using the following commands:

```
cd target/classes
find org/apache/commons/imaging/internal -name "*.class" | sed 's/\//./g' | sed
    ↪ 's/\.class$//'
find org/apache/commons/imaging/formats/psd/dataparsers -name "*.class" | sed
    ↪ 's/\//./g' | sed 's/\.class$//'
find org/apache/commons/imaging/formats/tiff -name "*.class" | sed 's/\//./g' | sed
    ↪ 's/\.class$//'
```

These commands search for all ‘.class‘ files in the specified directories and convert them into fully qualified class names, which are then copied and saved to a text file called `Randoop_Test_Classes.txt` manually.

8.3 Running Randoop

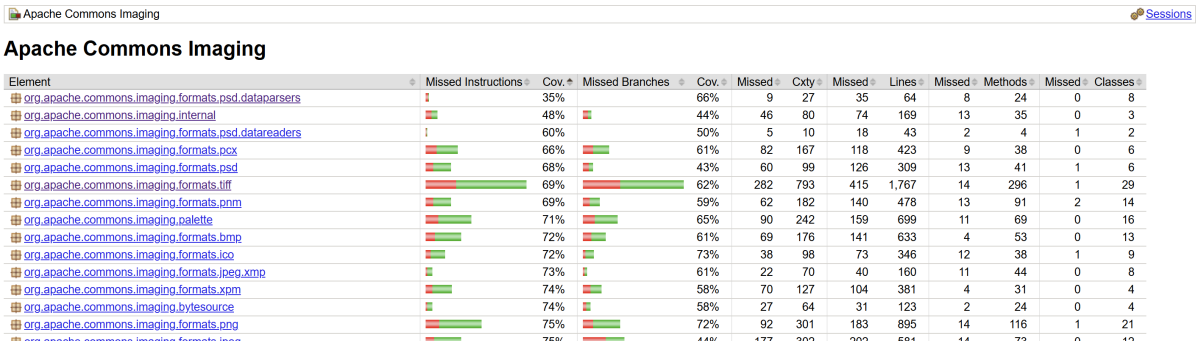
Once the class names were added to the `Randoop_Test_Classes.txt` file, we ran the following command to generate the tests:

```
java -cp "randoop-all-4.3.2.jar;target/classes" randoop.main.Main gentests
    ↪ $(Get-Content Randoop_Test_Classes.txt | ForEach-Object \{ "--testclass=$\_"\}
    ↪ ) --time-limit=200
    ↪ --junit-output-dir=src/test/java/org/apache/commons/imaging/randoop/
```

This command uses the Randoop tool to generate tests for the classes listed in the text file. The generated tests are stored in the `randoop` folder and are written in JUnit format.

8.4 Test Coverage Improvement

After generating the tests, we observed an improvement in coverage, as shown in the following image:


Apache Commons Imaging

Sessions

Apache Commons Imaging

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.imaging.formats.psd.dataparsers		35%		66%	9	27	35	64	8	24	0	8
org.apache.commons.imaging.internal		48%		44%	46	80	74	169	13	35	0	3
org.apache.commons.imaging.formats.psd.datareaders		60%		50%	5	10	18	43	2	4	1	2
org.apache.commons.imaging.formats.pcx		66%		61%	82	167	118	423	9	38	0	6
org.apache.commons.imaging.formats.psd		68%		43%	60	99	126	309	13	41	1	6
org.apache.commons.imaging.formats.tiff		69%		62%	282	793	415	1,767	14	296	1	29
org.apache.commons.imaging.formats.pnm		69%		59%	62	182	140	478	13	91	2	14
org.apache.commons.imaging.palette		71%		65%	90	242	159	699	11	69	0	16
org.apache.commons.imaging.formats.bmp		72%		61%	69	176	141	633	4	53	0	13
org.apache.commons.imaging.formats.ico		72%		73%	38	98	73	346	12	38	1	9
org.apache.commons.imaging.formats.jpeg.xmp		73%		61%	22	70	40	160	11	44	0	8
org.apache.commons.imaging.formats.xpm		74%		58%	70	127	104	381	4	31	0	4
org.apache.commons.imaging.bytesource		74%		58%	27	64	31	123	2	24	0	4
org.apache.commons.imaging.formats.png		75%		72%	92	301	183	895	14	116	1	21

Figure 8.2: Coverage improvement after generating tests using Randoop.

Metric	Instruction Coverage	Branch Coverage
Before	28%	25%
After	48%	44%

Table 8.1: Coverage for `org.apache.commons.imaging.internal`.

Metric	Instruction Coverage	Branch Coverage
Before	33%	66%
After	35%	66%

Table 8.2: Coverage for `org.apache.commons.imaging.formats.psd.dataparsers`.

Metric	Instruction Coverage	Branch Coverage
Before	57%	51%
After	69%	62%

Table 8.3: Coverage for `org.apache.commons.imaging.formats.tiff`.

8.5 Challenges and Limitations

While this method improved coverage, it has some limitations:

- The generated tests often contain redundancy.
- It does not guarantee sufficient coverage for all methods.
- The process generates more tests than are necessary, leading to inefficiency.

Despite these limitations, automated test generation with Randoop provides a valuable first step in improving test coverage, especially for poorly tested components.

9 Security Analysis with Snyk

9.1 Steps to Run Snyk

To perform security analysis using Snyk, follow these steps:

```
npm install -g snyk
snyk auth
snyk test
snyk test --json > snyk-report.json
```

9.2 Vulnerability Report

There were no vulnerabilities found for the commons-imaging project, as seen in the image below. However, there are some vulnerable paths in the web app backend Dockerfile, which will be reported instead.

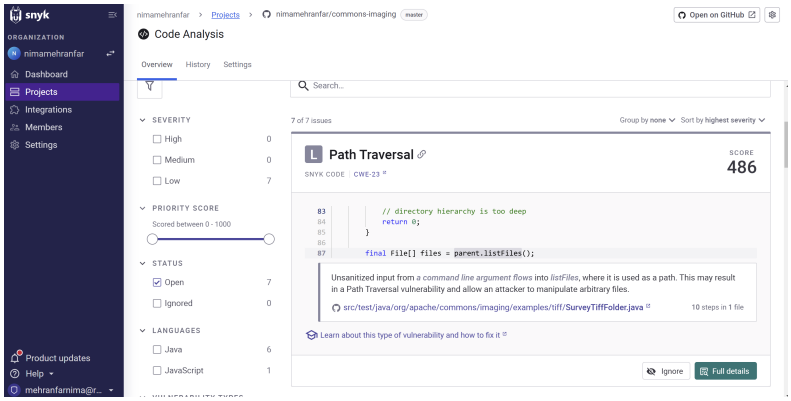


Figure 9.1: Snyk report showing no vulnerabilities for commons-imaging.

By changing the OpenJDK version to a newer one, many of the vulnerabilities were resolved, as shown in the image below:

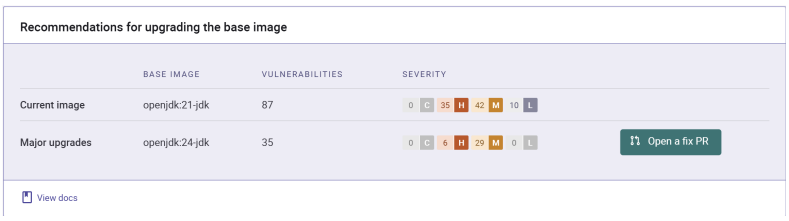


Figure 9.2: Vulnerabilities reduced after updating the OpenJDK version.

9.3 Security Issues Found

During the analysis, the following security issues were identified:

- 7 ”low” level issues, all of which were related to unsanitized input.
- This issues seem to be false positives because the project is an API and utility, and there is no allowed directory. Users can use any directory they want.

- However, if we want to limit the user to their current directory, we can fix these issues. For example, unsanitized input from a command-line argument was found in the file `src/test/java/org/apache/commons/imaging/examples/tiff/SurveyTiffFolder.java:collectPaths.class`, leading to a potential `Path Traversal` vulnerability.

Below is an image showing the identified vulnerability:

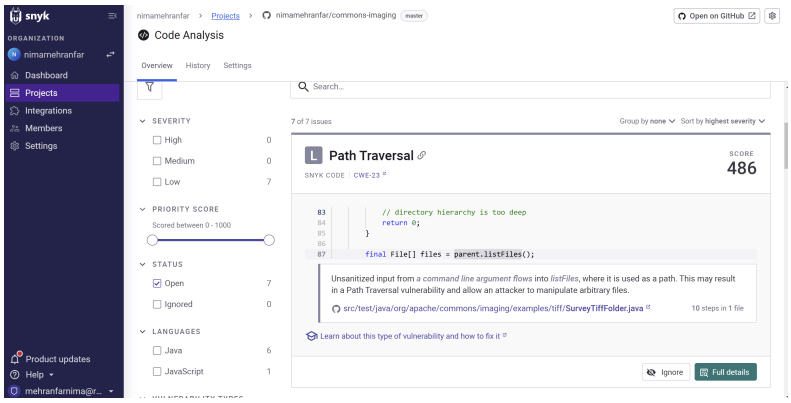


Figure 9.3: Unsanitized input vulnerability found in SurveyTiffFolder.java:collectPaths.

To fix this issue, we can ensure that the file paths remain within the base directory, as shown in the code snippet below:

Listing 9.1: SurveyTiffFolder.java:collectPath.class fix

```
private static final String BASE_DIRECTORY = System.getProperty("user.dir");

private static boolean isWithinBaseDirectory(File file) throws IOException {
    // Get the canonical path of the file
    String canonicalPath = file.getCanonicalPath();
    String baseDirectoryPath = new File(BASE_DIRECTORY).getCanonicalPath();

    // Check if the canonical path starts with the base directory path
    return canonicalPath.startsWith(baseDirectoryPath);
}

private static int collectPaths() {
    ...
    try {
        // Resolve the file's canonical path to get the absolute path
        File canonicalFile = f.getCanonicalFile();

        // Normalize the path and ensure it is within the base directory
        if (!isWithinBaseDirectory(canonicalFile)) {
            continue; // Skip files outside the base directory
        }

        final String[] temp = Arrays.copyOf(scratch, depth + 1);
        pathList.add(temp);
    } catch (IOException e) {
        // Handle any IOExceptions (e.g., if getCanonicalPath fails)
    }
    ...
}
```

This fix ensures that the file operations are limited to the base directory, preventing path traversal vulnerabilities.
All other identified issues can be fixed in a similar manner if they are not false positives, by sanitizing the inputs and ensuring that file operations are safe.

10 Conclusion

The commons-imaging project underwent various quality assurance practices. Through CI/CD pipeline integration, refactoring based on SonarCloud issues, performance and mutation testing, and security analysis, we have significantly improved the project's code quality and created a test Docker for it. Future work will depend on project developers focusing on addressing any remaining vulnerabilities and expanding the test coverage.

The source code for this project is available at `GitHub Repository`.

11 References

- SonarCloud Documentation: `https://sonarcloud.io/`.
- Docker Documentation: `https://docs.docker.com/`.
- JaCoCo Documentation: `https://www.jacoco.org/jacoco/`.
- PiTest Documentation: `https://pitest.org/`.
- JMH Documentation: `https://openjdk.java.net/projects/code-tools/jmh/`.
- Randoop Documentation: `http://randoop.github.io/`.
- Snyk Documentation: `https://snyk.io/`.
- ChatGPT: `https://chatgpt.com/`.