SWE_261P_Final_Report

Team Members:
- Yuxin Huang: yuxinh20@uci.edu
- Changhao Liu: liuc50@uci.edu
- Ruokun Xu: ruokunx@uci.edu

Team Name: OffersAreHere

Github link: https://github.com/yx-hh/commons-io

Final Report for SWE 261: Apache-commons-io

Catalog

Introduction

Apache Commons IO is the component of the Apache Commons which is derived from Java API and covers extensive use cases for common operations of file input and output. Apache Commons IO Library implements various classes, such as utility classes which provide file and string comparison, filter classes which provide a way to filter files based on logical criteria rather than string-based comparisons, and file monitor classes which provide ways to track changes in a target file or folder and allow actions to be taken on the changes (Apache Commons IO, tutorialspoint.com).

Additional Info

- Language: Java
- Lines of Code (git ls-files | xargs cat | wc -l): 92774
- Number of classes (git ls-files | wc -l): 211
- Javadoc: https://commons.apache.org/proper/commons-io/apidocs/

Building and Running instruction

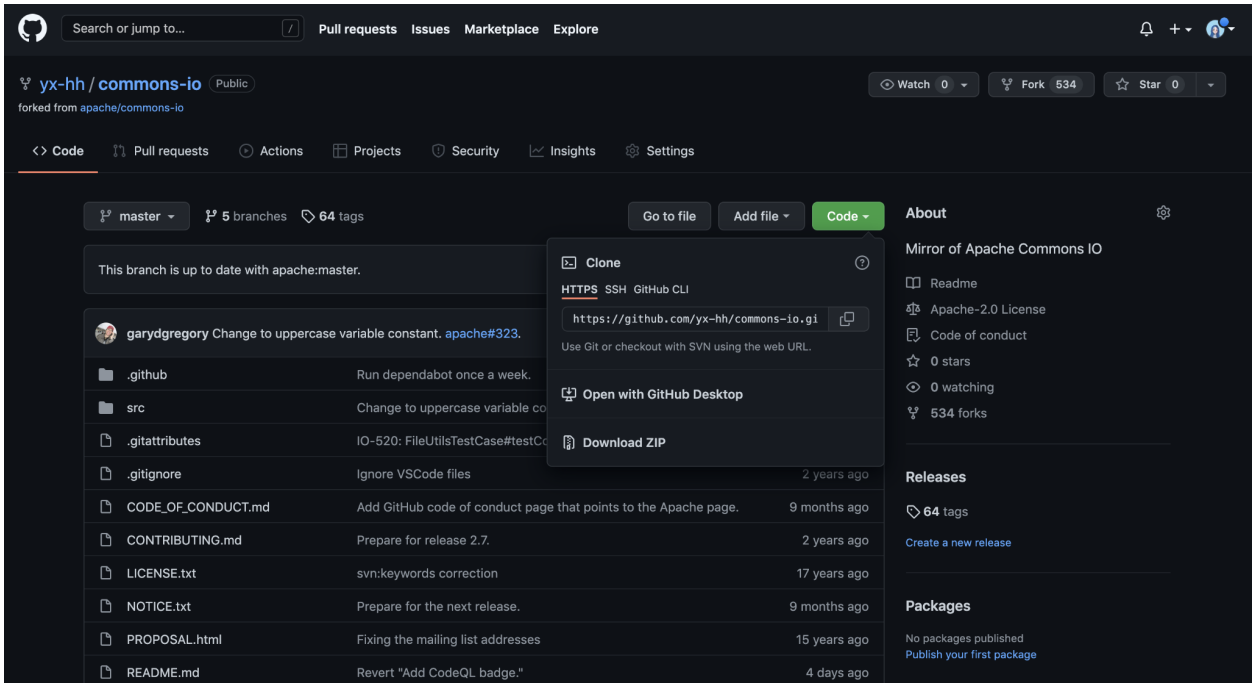
- Prerequisite: maven and java environment required

Building Instruction & Package Instruction

- Use maven dependency: copy and paste the following dependency into pom.xml in your projects and recompile.

```
<!-- https://mvnrepository.com/artifact/commons-io/commons-io -->
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.11.0</version>
</dependency>
```

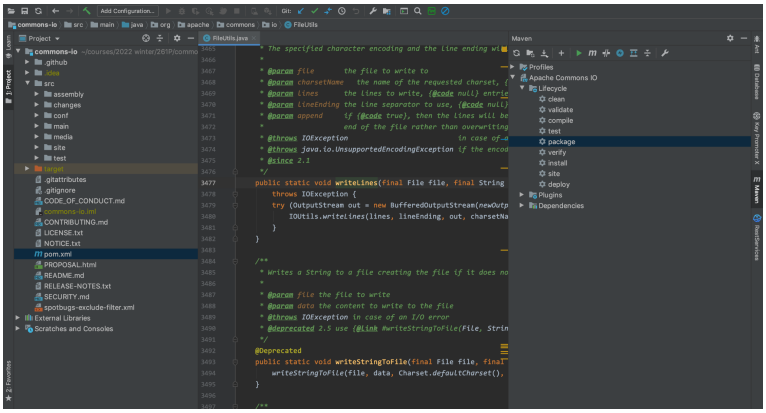
- Use jar: Download the code from GitHub repo: GitHub - yx-hh/commons-io: Mirror of Apache Commons IO



1. Open with IntelliJ Idea and set as a MAVEN project
2. Use **GUI** on **IntelliJ** or the command line in the console to compile and package the project as a jar file.

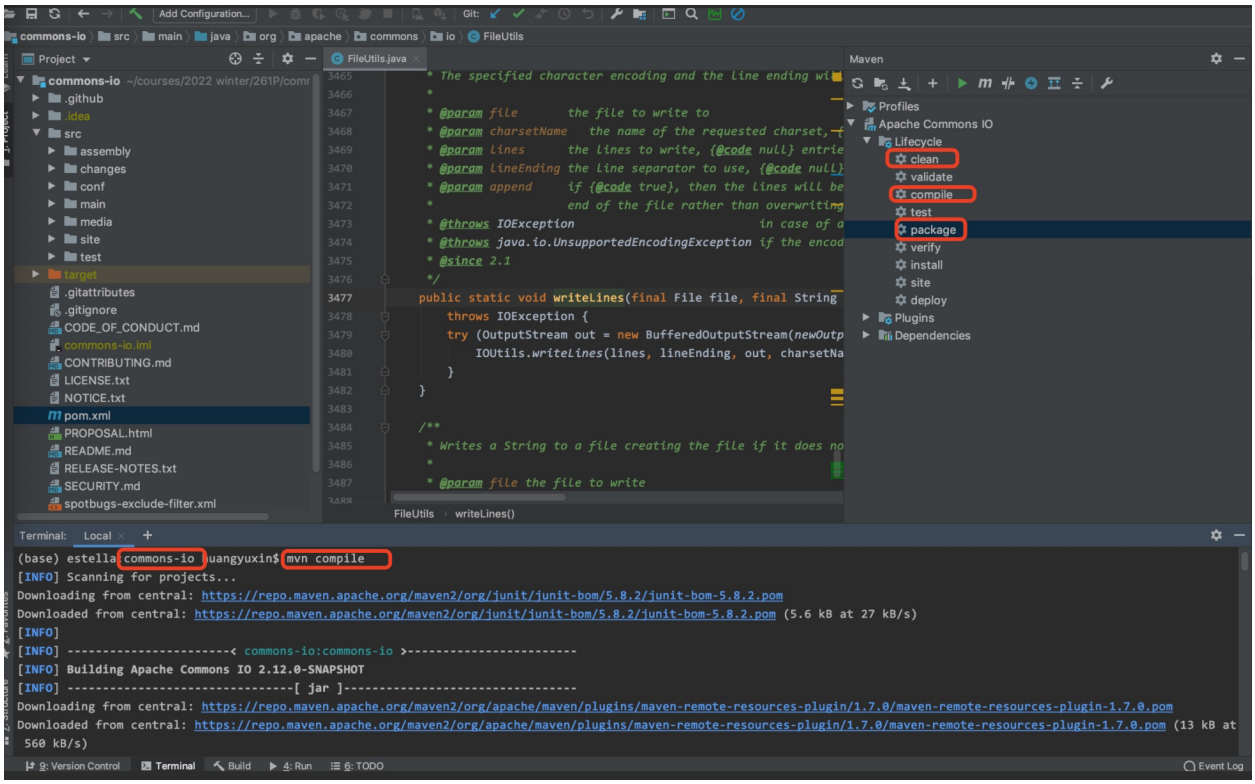
Build the project using GUI:

Find the maven table on the right side and click clean, compile and package in order. The picture instruction is as follows.

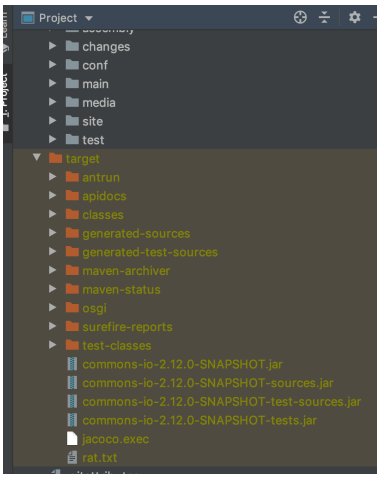


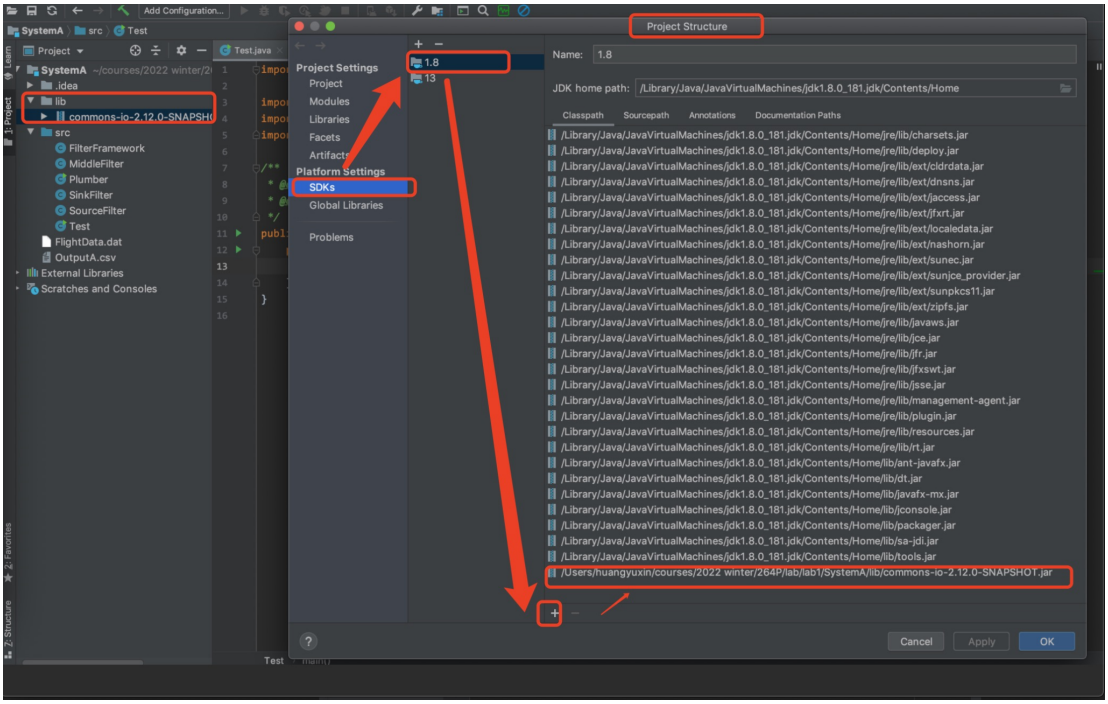
Build the project using Command Line:

Under the folder of commons-io, use command line `mvn clear, mvn compile, mvn package`. In your computer, it may be `maven clean, maven compile, maven package`, it depends on your maven setting.



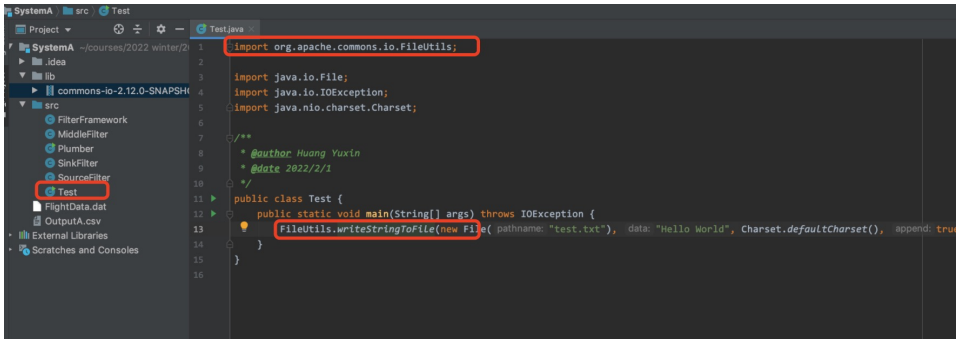
3. After successfully building the project, we can find the jar file under the folder of the target, copy the commons-io-2.12.0-SNAPSHOT.jar to a normal java project, create a lib folder and import this file to java lib. This jar file is an executable file that contains the source code.





Running instruction

- Use API from commons-io like the following, and static methods can be used directly.

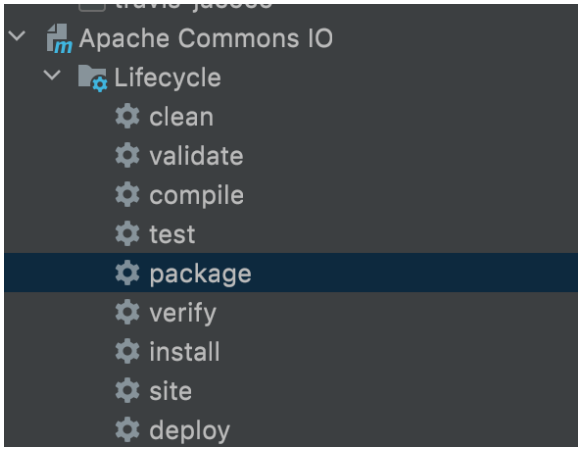


Existing Test Cases

Commons IO utilizes a variety of testing techniques that we learned during class, including smoke testing, black box testing, partition, etc. Those techniques play an essential role in helping developers and clients to maintain a great coverage of the codebase and to detect bugs at an early stage.

Smoke Testing

Commons IO has utilized smoke testing, which automatically runs after each “build” to make sure all the main features work well. To understand the role of smoking testing, we should first learn what “build” is: build is to take all the source codes, compile them into machine-readable codes and convert them into software artifacts. Smoking testing can verify the build and check whether the essential features are working. If one of the test cases breaks, it will reject the build and stop running all the further tests to save some time and bandwidth. As you can see from the example below, when we start the build, Commons IO begins to run some smoke tests such as FileUtilsTest to confirm that the main functionalities are working. If any test cases fail, it will reject the build and stop running the rest of the test cases.




```
Downloading from central: https://repo.maven.apache.org/maven2/org/assertj/assertj-parent-pom/2.1.9/assertj-parent-pom-2.1.9.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/assertj/assertj-parent-pom/2.1.9/assertj-parent-pom-2.1.9.pom (15 kB at
  kB/s)
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running org.apache.commons.io.FileUtilsTest
[WARNING] Tests run: 159, Failures: 0, Errors: 0, Skipped: 1, Time elapsed: 2.939 s - in org.apache.commons.io.FileUtilsTest
[INFO] Running org.apache.commons.io.FileUtilsDirectoryContainsTest
[INFO] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.221 s - in org.apache.commons.io.FileUtilsDirectoryContainsTest
```

Black Box Testing

Common IO also utilizes black box testing. According to IEEE, black box testing is “testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions”. In short, testers do not understand the internal system but write test cases to validate some functionalities. Therefore, black box testing mainly focuses on the inputs and expected outputs.

Here is one of the examples: In this screenshot, this specific test case 1. creates a new file 2. writes a string in the file 3. checks the input stream's first character is 'H'.

```
@Test
public void test_openInputStream_exists() throws Exception {
    final File file = new File(tempDirFile,  child: "test.txt");
    TestUtils.createLineBasedFile(file, new String[]{"Hello"});
    try (FileInputStream in = FileUtils.openInputStream(file)) {
        assertEquals( expected: 'H', in.read());
    }
}
```

Boundary Checking:

Boundary checking is quite essential in checking whether the program is able to run successfully in some corner cases such as inputs being null, negative numbers, zero, or positive numbers. Commons IO utilizes this technique to check valid inputs as well as invalid inputs, looking for some potential bugs that may cause the program to break. For example, here are two cases to test the boundary values for openOutputStream. The first test case checks when the file exists and is valid, while the second one tests when the file is a directory, which is an invalid input. Both two tests use boundary values for detecting some potential bugs.

```
@Test
public void test_openOutputStream_exists() throws Exception {
    final File file = new File(tempDirFile,  child: "test.txt");
    TestUtils.createLineBasedFile(file, new String[]{"Hello"});
    try (FileOutputStream out = FileUtils.openOutputStream(file)) {
        out.write( b: 0);
    }
    assertTrue(file.exists());
}

@Test
public void test_openOutputStream_existsButIsDirectory() {
    final File directory = new File(tempDirFile,  child: "subdir");
    directory.mkdirs();
    assertThrows(IllegalArgumentException.class, () -> FileUtils.openOutputStream(directory));
}
```

Testing Framework and how to run tests:

The testing framework of Commons IO is JUnit, which is a popular framework for Java developers. There are multiple ways to run JUnit test cases for the entire suite, a single class, or a single test case.

- To run all the test cases, you can execute “mvn clean test” in the terminal.

```
changhao@changhaoliusmbp commons-io % mvn clean test
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< commons-io:commons-io >-----------------------
[INFO] Building Apache Commons IO 2.12.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ commons-io ---
[INFO] Deleting /Users/changhao/Desktop/style-winter/commons-io/target
[INFO]
```

- To run test cases for a specific class, use “mvn clean test -Dtest=xxxxTest” such as “mvn clean test -Dtest=FileSystemTest”

```
changhao@changhaoliusmbp commons-io % mvn clean test -Dtest=FileSystemTest
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< commons-io:commons-io >-----------------------
[INFO] Building Apache Commons IO 2.12.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ commons-io ---
[INFO] Deleting /Users/changhao/Desktop/style-winter/commons-io/target
```

- To run one single test case, execute “`mvn clean test -Dtest=xxxxTest#testA`” such as “`mvn clean test -Dtest=FileSystemTest#testSorted`”

```
changhao@changhaoliusmbp commons-io % mvn clean test -Dtest=FileUtilsTest#test_openInputStream_notExists
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< commons-io:commons-io >-----------------------
[INFO] Building Apache Commons IO 2.12.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ commons-io ---
[INFO] Deleting /Users/changhao/Desktop/style-winter/commons-io/target
[INFO]
[INFO] --- maven-enforcer-plugin:3.0.0:enforce (enforce-maven-version) @ commons-io ---
[INFO]
[INFO] --- maven-enforcer-plugin:3.0.0:enforce (enforce-maven-3) @ commons-io ---
[INFO]
[INFO] --- maven-enforcer-plugin:3.0.0:enforce (enforce-maven) @ commons-io ---
[INFO]
[INFO] --- apache-rat-plugin:0.13:check (rat-check) @ commons-io ---
[INFO] Enabled default license matchers.
[INFO] Will parse SCM ignores for exclusions...
[INFO] Parsing exclusions from /Users/changhao/Desktop/style-winter/commons-io/.gitignore
[INFO] Finished adding exclusions from SCM ignore files.
```

- If you prefer to use GUI, you can simply click the green arrow of a test class or a single test case:

Systematic functional testing

Systematic functional testing is a type of black-box testing. When testing the system, QA will test the software as a whole, and the software system will be divided into different functionalities. Importantly, QA developers write test cases to those functions individually without looking into the source code. The test functions under the specifications and sets input data and expected output. After testing all the functions as a system, testers will validate the system's complete and integrate functions together to make sure the system runs smoothly.

Systematic functional testing example

An automated company manufactures a computer system that allows users to get access to a variety of car features such as video player, GPS, voice control, and climate control. Without an understanding of how the inner system was built, testers would write test cases to all of these features individually without checking the system internally, and “they must also test them as a complete system to ensure interoperability and a good user experience.” [15 Functional Testing Types with Examples - Applause](https://www.applause.com/blog/functional-testing-types-examples)

Systematic Functional testing typically involves seven steps

1. Divide the software system into different functions
2. Identify the expected behaviors of each function
3. Create input data based on each function's specifications
4. Confirm the expected and valid output based on the function's specifications
5. Run the test cases with those inputs
6. Compare the expected output and actual output to ensure that they are equal
7. Finish running all the test cases to ensure that the entire system works as expected
[Functional Testing - Wikipedia]

Partitioning Test

Partitioning test is a type of testing that separates input values into a variety of cases that cover all situations, reducing the number of test cases and decreasing testing time. This method is quite essential in defining test cases to cover all the potential errors and to reduce the time of writing test cases, as there is no need to write every combination thanks to this technique. [cite from [Equivalence partitioning - Wikipedia]]

Partitioning Test Example

Here is an example of an online shopping site that utilizes partition tests. With the unique id and name of each product, we are able to acquire the specific product either by using the product's name or ID.

Product	ID
Belt	111
Glove	121
Monitor	88
Mouse	30
Butter	20
Cable	1

If a client enters an invalid product ID, the application will present an error page. If the user types a valid one, for example, enter 111 for Belt, the application will present a valid page. [example cited from [Equivalence Partitioning Method - GeeksforGeeks]]

Partition Test four steps

1. Divide test data into equivalence partitions according to the specifications
2. Select some test cases from each partition
3. Write test cases regarding expected behaviors.
4. Produce and execute actual tests

Commons-IO partitioning test case

In Commons-IO project, we have FileUtils, IOUtils, CopyUtils, etc features to deal with all kinds of problems related to IO Operation. In this part, we chose FileUtils feature for the partitioning test. For partition boundaries, we can use a variety of file size, charset types, and timestamps.

For example, in FileUtils class, we have a method named: public static boolean isFileOlder(final File file, final long timeMillis). In its specification noted the time reference measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970). When we have input such as timeMillis is null or input file is invalid which does not match the partition principle, we will get error feedback. Test cases related to this example as following:

New test cases in JUnit

In IntelliJ choose method and right click run test case
Here is a test case to check whether refFile's last modified time is older than the input time

```
    @Test
    public void testIsFileOlderTrue() throws Exception {
        final File refFile = TestUtils.newFile(tempDirFile,
"FileUtils-reference.txt");
        TestUtils.generateTestData(refFile, 1);
        long time = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").parse("2100-01-01 01:01:01").getTime();
        assertTrue(FileUtils.isFileOlder(refFile, time), "Old File -
Older - File");
    }
```

Here is a test case to check whether refFile's last modified time is newer than the input time

```
    @Test
    public void testIsFileOlderFalse() throws Exception {
        final File refFile = TestUtils.newFile(tempDirFile,
"FileUtils-reference.txt");
        TestUtils.generateTestData(refFile, 1);
        long time = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").parse("2001-01-01 01:01:01").getTime();
        assertFalse(FileUtils.isFileOlder(refFile, time), "Old File -
older - File");
    }
```

Here is test case to check whether NullPointerException Throw when refFile is invalid

```
    @Test
    public void testIsFileOlderFileInvalid() throws Exception {
        assertThrows(NullPointerException.class, () ->
FileUtils.isFileOlder(null, Instant.now()));
    }
```

Here is test case to check IllegalArgumentException when time data is invalid

```
    @Test
    public void testIsFileOlderFileTimeBoundaryInvalid() throws
Exception {
        final File refFile = TestUtils.newFile(tempDirFile,
"FileUtils-reference.txt");
        TestUtils.generateTestData(refFile, 1);
        final File invalidFile = TestUtils.newFile(tempDirFile,
"FileUtils-invalid.txt");
        // Invalid reference File
        assertThrows(IllegalArgumentException.class, () ->
FileUtils.isFileNewer(refFile, invalidFile));
     }
```

Finite State Machine

Finite State Machine Introduction

Finite State Machine (FSM) is a computational model based on a hypothetical machine consisting of one or more states to simulate the flow of state transition. FSM is a directed graph with each node representing a single state, and each edge is the action that takes the artifact from one state to another(Pezzè and Young, pg 65). Apart from these, each machine only has one active state at a time. Each state has a set of transitions depending on user actions (Cited from [What is a Finite State Machine?]). Finite state machines facilitate functional testing by describing the program as a state-transition model, helping testers understand the characteristics of the program and save their time to write cases to fully test the program(Pezzè and Young, pg 65).

Usage and Advantages of Finite Models

Finite models represent a simpler version of the artifact but keep the main characteristics of the artifact, making it easier for testers to understand the transitions within the feature and to analyze results based on a variety of actions(Pezzè and Young, pg 55). Thanks to this model, testers are able to trace the error states as well as complete states for the artifact. Each time the system only stays at one state, which is controlled by inputs from testers. In other words, we can draw a state transition flow of the program according to the finite model theory. The state transition flow is able to clearly illustrate the state of the current process and help testers save time and comprehensively examine the program with fewer test cases. Furthermore, with a finite model we can easily observe which branch the bug exists in. (Cited from [Model Based Testing Tutorial: What is, Tools & Example])

Feature Chosen

The feature with the FSM function model we chose is to check whether the directory is empty by analyzing its corresponding path. There are a few valid states and three error states that may occur:

- Return true if the directory of that path is valid and empty.
- Return false if the directory is valid and has no content in it.
- Throw NotDirectoryException if the path doesn’t lead to a directory, as such a file cannot be opened.

- Throw IOException if any I/O errors happen.
- Throw SecurityException if the installed security manager finds that the user doesn't have the proper access permission to the directory.

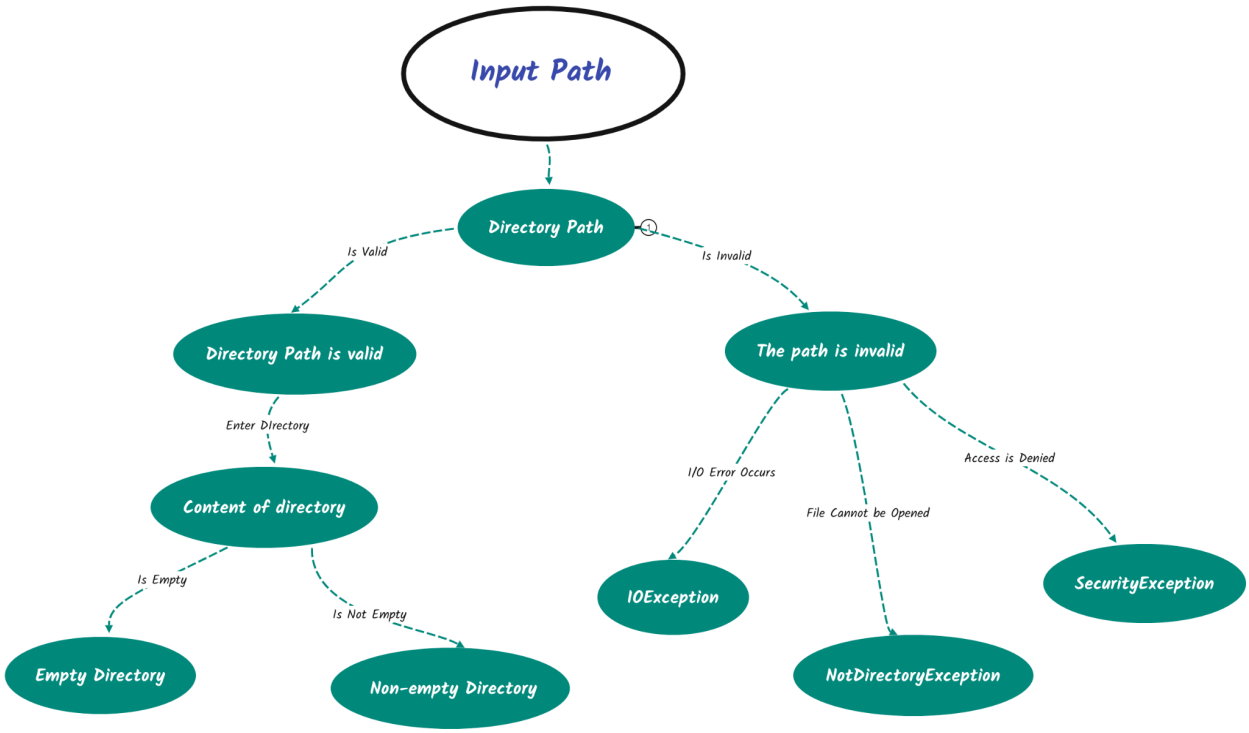
Functional Model of this feature

As you can see from the model we drew below, the feature that we tested can be converted to a functional model. First, the model starts in a directory path state. Depending on whether the path is valid or not, the model will move to a valid path state or an invalid path state. In the valid directory state, we can transition to the "Content of directory" state by entering the directory. After that, by checking the content of the directory, we can move to the "Empty Directory" state or the "Non-empty Directory" state.

If the directory path is invalid, the model moves from the "Directory Path" state to the invalid path state. The next transition depends on the occurrence of each error. For example, if an I/O error occurs, the model will proceed to the IOException state.

In conclusion, building a functional model guides us to gain a deeper understanding of this feature and to write test cases to comprehensively cover some corner cases.

Finite State Flow Graph



Test Cases covers the Functional Model

1. Test case for checking whether directory is empty state or not: expect true with empty content under the directory

```
@Test
  public void testIsEmptyDirectoryStateTrue() throws IOException {
     final Path tempDir = Files.createTempDirectory(getClass().getCanonicalName());
     assertTrue(PathUtils.isEmptyDirectory(tempDir));
     Files.delete(tempDir);
  }
```

2. Test case for checking whether directory is empty state or not: expect false with file under the directory

```
@Test
  public void testIsEmptyDirectoryStateFalse() throws IOException {
     final Path tempDir = Files.createTempDirectory(getClass().getCanonicalName());
```

```
    Files.createTempFile(tempDir, "prefix", null);
    assertFalse(PathUtils.isEmptyDirectory(tempDir));
    FileUtils.deleteDirectory(tempDir.toFile());
  }
```

3. Test case for checking whether directory is empty state or not: expect NotDirectoryException exception with input a file path rather than a directory path

```
@Test
  public void testIsEmptyDirectoryStateNotDirectoryException() throws IOException {
    final Path filePath = Files.createTempFile(tempDirPath, "prefix", null);
    assertThrows(NotDirectoryException.class, () -> PathUtils.isEmptyDirectory(filePath));
    Files.delete(filePath);
  }
```

4. Test case for checking whether directory is empty state or not: expect IOException exception with input a non-exist path rather than a real path

```
@Test
  public void testIsEmptyDirectoryStateIOException() throws IOException {
    final Path tempDir = Files.createTempDirectory(getClass().getCanonicalName());
    Files.delete(tempDir);
    assertThrows(IOException.class, () -> PathUtils.isEmptyDirectory(tempDir));
  }
```

Structural Testing

Structural testing is one of the four major software testing types defined by the International Software Testing Qualifications Board. As its name suggests, structural testing evaluates the structure of the code and guides developers to find out whether the test suite is sufficient to cover the codebase (Pezzè and Young, pg 211 - pg 215). Undoubtedly, a program is not sufficiently tested if some of its elements have never been executed. To assess the thoroughness of existing test cases, developers utilize structural testing to locate elements that are not executed, including statements, branches, conditions, and paths (Pezzè and Young, pg 212). It verifies aspects such as the correct implementation of the conditional statements of the code and whether each statement in the code executes and runs smoothly. Structural-based testing cases are usually provided by developers, as they understand how the system was built, to improve the thoroughness of the current test suite rather than examine if the function can output the correct result.

Example: As for a program that calculates the average value of an int array, structural testing focuses on evaluating steps of counting the average value, rather than checking the correctness of the result(*Structural Testing Tutorial, 2022*).

Advantages:

- Thoroughly tests the codebase to reduce the undetected errors that may occur.
- Has some existing tools to automate the process.
- Helps developers to deeply understand code's structure, fix any errors and implement higher quality code.
- It can be executed during the system implementation progress, saving time.
- Prunes unnecessary code.
- Runs by developers, and there is no need to wait for other testers such as QAs.

Code Coverage Analysis

We utilized jacoco, a coverage report generator tool, to analyze the coverage of the existing test suite: Commons IO has already integrated with Jacoco. In the pom.xml file, there is a line with `<commons.jacoco.version>0.8.7</commons.jacoco.version>` , informing us current Jacoco version used in Commons IO is 0.8.7.

How to run Jacoco

Jacoco automatically generates a coverage report when we start a build or run the test suite. To start a build, developers can simply run `mvn` in the command line and wait for the build to complete. After such build completes, we are able to see a detailed report generated by Jacoco about the test suite coverage.

Coverage of the existing test suite

Jacoco provides a few coverage measures, including class, method, line, and branch, guiding us to understand the coverage of the existing test suite. As you can see from Figure 1 below, Commons IO has class coverage of 94%, method 88%, line 87%, and branch coverage 84%.

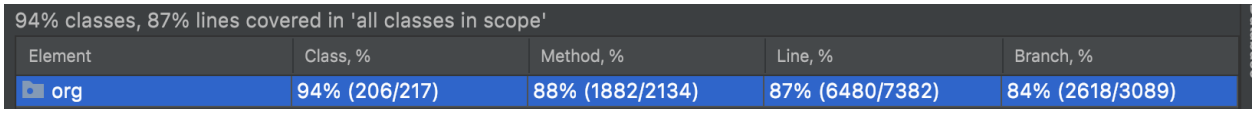


Figure 1: Overall Code Coverage for Apache Commons IO, From IntelliJ, By Changhao Liu

Detailed Report

Jacoco is able to generate a detailed report into an html page. By clicking a specific class name or method name on the html page, we can examine its coverage information. As you can see from Figure 2 below, Commons IO has a variety of packages with different coverage statistics:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.io.jmh.jmh_generated		0%		0%	538	538	2,432	2,432	130	130	20	20
org.apache.commons.io		89%		83%	579	2,601	792	9,305	343	1,768	3	111
org.apache.commons.io.input		92%		87%	288	1,980	456	6,630	152	1,364	0	133
org.apache.commons.io.output		85%		84%	145	940	443	3,185	107	798	0	90
org.apache.commons.io.filefilter		91%		77%	137	618	120	2,323	64	436	1	43
org.apache.commons.io.jmh		0%		0%	76	76	196	196	20	20	2	2
org.apache.commons.io.file		88%		64%	126	554	151	1,345	35	387	3	41
org.apache.commons.io.input.buffer		47%		41%	49	81	68	173	11	29	1	4
org.apache.commons.io.input.compatibility		66%		61%	65	142	108	276	12	34	0	4
org.apache.commons.io.comparator		90%		67%	35	128	12	353	1	76	0	19
org.apache.commons.io.monitor		94%		86%	31	176	45	674	15	111	0	9
org.apache.commons.io.test		81%		70%	12	50	19	119	3	33	0	6
org.apache.commons.io.serialization		95%		96%	4	97	7	254	3	84	0	13
org.apache.commons.io.function		97%		75%	5	70	8	153	4	68	0	7
org.apache.commons.io.charset		93%		100%	2	14	2	30	2	12	0	4
org.apache.commons.io.file.spi		98%		80%	2	18	1	33	0	13	0	2
org.apache.commons.io.file.attribute		100%		n/a	0	17	0	37	0	17	0	2
Total	24,035 of 132,180	81%	1,801 of 5,434	66%	2,094	8,100	4,860	27,518	902	5,380	30	510

Figure 2: Code Coverage for each package in Apache Commons IO, From Jacoco, By Changhao Liu

For example, as you can see from Figure 3, after clicking “org.apache.commons.io”, we were routed to a page that contains coverage information for that specific package. Furthermore, not only can we check the number of lines, methods, or classes that the test suite misses, but also we can rank classes regarding their coverage ascendingly or descendingly.

org.apache.commons.io

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
FileUtilsTest.java		88%		57%	138	339	61	1,744	82	270	0	8
FileUtilsDeleteDirectoryBaseTest.java		0%		n/a	9	9	123	123	9	9	1	1
IOUtilsTest.java		90%		70%	38	204	91	872	35	199	0	12
FileSystemUtilsTest.java		68%		60%	43	91	7	183	35	81	0	5
FilenameUtilsTest.java		96%		50%	17	59	43	875	1	43	0	1
FileUtils.java		92%		94%	18	268	52	590	5	159	0	1
IOUtils.java		90%		85%	44	288	49	508	15	154	0	1
FileUtilsDeleteDirectoryLinuxTest.java		0%		0%	8	8	44	44	5	5	1	1
DirectoryWalkerTest.java		84%		96%	1	57	29	238	0	43	0	6
CloseableURLConnection.java		18%		n/a	41	48	50	62	41	48	0	1
FileCleaningTrackerTest.java		83%		40%	15	33	15	170	5	22	0	1
IOCaseTest.java		88%		50%	28	48	0	181	17	37	0	1
DirectoryWalkerTestCaseJava4.java		90%		96%	1	51	19	206	0	38	0	5
FileSystemUtils.java		86%		73%	24	66	20	160	2	14	0	1
FileUtilsFileNewerTest.java		67%		50%	10	18	4	38	6	14	0	1
IOUtilsWriteTest.java		95%		n/a	15	69	0	387	15	69	0	1
IOUtilsCopyTest.java		92%		n/a	15	48	0	236	15	48	0	1
ByteOrderMarkTest.java		82%		n/a	0	11	12	66	0	11	0	1
FileUtilsDeleteDirectoryWindowsTest.java		0%		0%	4	4	15	15	2	2	1	1
FilenameUtils.java		96%		93%	21	238	17	386	2	49	0	1
FileDeleteStrategyTest.java		81%		50%	5	11	3	68	2	8	0	1
FileSystem.java		92%		78%	12	40	13	69	5	21	0	1
FileUtilsDirectoryContainsTest.java		89%		n/a	0	14	8	81	0	14	0	1
DeleteDirectoryTest.java		84%		n/a	3	11	2	41	3	11	0	1
ThreadMonitorTest.java		58%		n/a	0	6	9	30	0	6	0	1
HexDumpTest.java		96%		100%	0	25	9	132	0	4	0	1
CopyUtils.java		80%		100%	2	15	9	47	2	13	0	1
FileCleaner.java		18%		n/a	7	9	12	14	7	9	0	1
StreamIterator.java		44%		0%	5	7	10	15	3	5	0	1
LineIteratorTest.java		96%		90%	5	40	5	150	3	30	0	3

Figure 3: Code Coverage for package: Commons IO, From Jacoco, By Changhao Liu

Uncovered Test Cases

In “org.apache.commons.io”, one particular class, CloseableURLConnection, only has 18% of coverage: missed 50 out of 62 lines and 41 out of 48 methods according to Figure 4. By diving deeply into this class, we can see there are a few lines of code that are not covered by the original test suite. Jacoco marked the uncovered lines to red color and covered ones to green, as you can see from the code snippet below. To increase the code coverage of CloseableURLConnection, we have added a few test cases which will be discussed in the next section.

serialization	100% (4/4)	100% (19/19)	96% (54/56)	100% (20/20)
ByteOrderMark	100% (1/1)	100% (9/9)	100% (45/45)	100% (24/24)
ByteOrderParser	100% (1/1)	100% (1/1)	100% (5/5)	100% (4/4)
Charsets	100% (1/1)	100% (4/4)	100% (17/17)	100% (4/4)
CloseableURLConnection	**100% (1/1)**	**14% (7/48)**	**19% (12/62)**	**100% (0/0)**
CopyUtils	100% (1/1)	84% (11/13)	80% (38/47)	100% (4/4)

Figure 4: Original Code Coverage for ClosableURLConnection From Jacoco, By Changhao Liu

Here is one example that Jacoco marked unexecuted codes as red:

```
1.  @Override
2.     public void addRequestProperty(final String key, final String value) {
3.         urlConnection.addRequestProperty(key, value);
4.     }
```

```
5.
6.      @Override
7.      public void close() {
8.          IOUtils.close(urlConnection);
9.      }
10.
11.     @Override
12.     public void connect() throws IOException {
13.         urlConnection.connect();
14.     }
15.
16.     @Override
17.     public boolean equals(final Object obj) {
18.         return urlConnection.equals(obj);
19.     }
20.
21.     @Override
22.     public boolean getAllowUserInteraction() {
23.         return urlConnection.getAllowUserInteraction();
24.     }
25.
26.     @Override
27.     public int getConnectTimeout() {
28.         return urlConnection.getConnectTimeout();
29.     }
30.
31.     @Override
32.     public Object getContent() throws IOException {
33.         return urlConnection.getContent();
34.     }
35.
36.     @Override
37.     public Object getContent(@SuppressWarnings("rawtypes") final Class[] classes) throws IOException {
38.         return urlConnection.getContent(classes);
39.     }
40.
41.     @Override
42.     public String getContentEncoding() {
43.         return urlConnection.getContentEncoding();
44.     }
45.
```

Newly Added Test Cases to improve coverage

For the whole project, the coverage for classes, methods, line and branch are 94%, 88%, 87% and 84% separately. We chose to increase the coverage for ClosableURLConnection class, which is responsible for dealing issues with URI and URL resources, an essential feature in commons-io. Before we added more test cases, the original coverage was 14% methods and 19% lines covered. After adding new test cases, we successfully increased the coverage from 14% methods to 29 % methods, and from 19% lines to 32% lines covered as you can see from Figure 5.

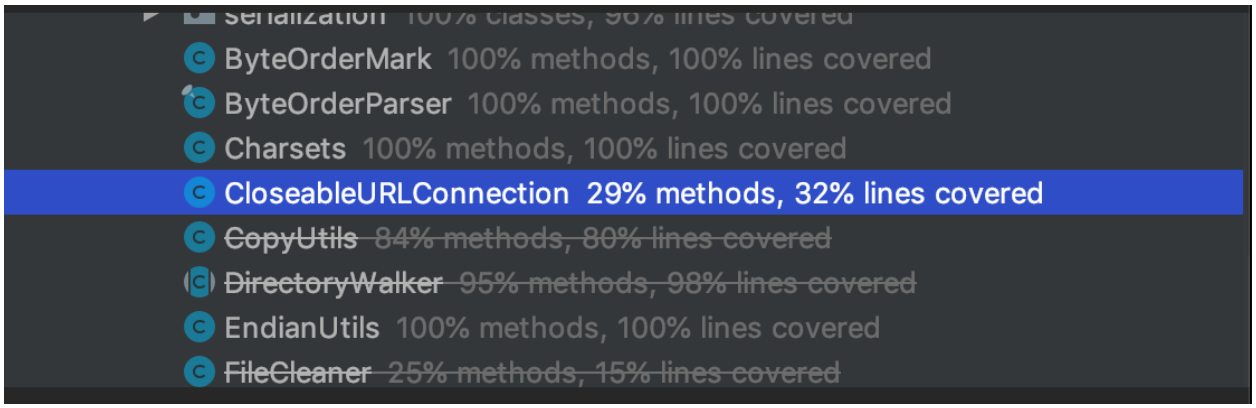


Figure 5: New Code Coverage for ClosableURLConnection From Jacoco, By Changhao Liu

Test Case Details

1.Test case for checking whether it can open a CloseableURLConnection with a URI resource

```
@Test
void open() throws IOException {
   final File testFile = TestUtils.newFile(tempDirFile, "testFile.txt");
   CloseableURLConnection uriConnection =
CloseableURLConnection.open(testFile.toURI());
   assertNotNull(uriConnection);
   uriConnection.close();
}
```

2.Test case for checking whether it can open a CloseableURLConnection with a URL resource

```
@Test
void testOpen() throws IOException {
   final File testFile = TestUtils.newFile(tempDirFile, "testFile.txt");
   CloseableURLConnection urlConnection =
CloseableURLConnection.open(testFile.toURI().toURL());
   assertNotNull(urlConnection);
   urlConnection.close();
```

```
}
```

3. Test case for checking whether it can add requestProperty to CloseableURLConnection

```
@Test
void addRequestProperty() throws IOException {
   final CloseableURLConnection connection = CloseableURLConnection.open(new
URL("https://www.google.com/"));
   final String key = "phone";
   final String value = "123456";
   connection.addRequestProperty(key, value);
   final String res = connection.getRequestProperty("phone");
   assertEquals(res, value);
}
```

4. Test case for checking whether the CloseableURLConnection can be closed as expected

```
@Test
void close() throws IOException {
   File testFile = TestUtils.newFile(tempDirFile, "testFile.txt");
   CloseableURLConnection connection =
CloseableURLConnection.open(testFile.toURI().toURL());
   connection.close();
   assertThrows(FileNotFoundException.class, () -> connection.getContent());
}
```

5.Test case for checking whether two ClosableURLConnection are equal or not

```
@Test
void testEquals() throws IOException {
   File testFile = TestUtils.newFile(tempDirFile, "testFile.txt");
   CloseableURLConnection connection1 =
CloseableURLConnection.open(testFile.toURI().toURL());
   CloseableURLConnection connection2 =
CloseableURLConnection.open(testFile.toURI().toURL());
   assertFalse(connection1.equals(connection2));
}
```

6.Test case for checking whether it can get connectTimeout or not

```
@Test
void getConnectTimeout() throws Exception {
   CloseableURLConnection connection = CloseableURLConnection.open(new
URL("https://www.google.com/"));
   connection.setConnectTimeout(5);
```

```
    assertEquals(5, connection.getConnectTimeout());
}
```

7. Task case for checking whether it can get connection's content or not

```
@Test
void getContent() throws IOException {
    // write data to test file
    File testFile = TestUtils.newFile(tempDirFile, "testFile.txt");
    final String content = "Hello World!";
    FileUtils.write(testFile, content, Charset.defaultCharset());
    // open as connection
    final CloseableURLConnection connection =
CloseableURLConnection.open(testFile.toURI().toURL());
    // read connection content
    final InputStream is = (InputStream) connection.getContent();
    final BufferedReader br = new BufferedReader(new InputStreamReader(is));
    String res = "";
    String currentLine = "";
    while ((currentLine = br.readLine()) != null) {
        res += currentLine;
    }
    // compare expect result and real result
    assertEquals(content, res);
    // delete test file and close connection
    FileUtils.delete(testFile);
    connection.close();
}
```

8. Test case for checking whether it can get correct content length or not

```
@Test
void getContentLength() throws IOException {
    final File testFile = TestUtils.newFile(tempDirFile, "testFile.txt");
    final String content = "Hello World!";
    FileUtils.write(testFile, content, Charset.defaultCharset());
    CloseableURLConnection connection =
CloseableURLConnection.open(testFile.toURI().toURL());
    assertEquals(content.length(), connection.getContentLength());
}
```

9.Test case for checking whether it can get the expected content type. When writing String to file and create connection, we expect its type to be “text/plain”.

```
@Test
void getContentType() throws IOException {
   final File testFile = TestUtils.newFile(tempDirFile,
"testFile.txt");
   final String content = "Hello World!";
   FileUtils.write(testFile, content, Charset.defaultCharset());
   CloseableURLConnection connection =
CloseableURLConnection.open(testFile.toURI().toURL());
   assertEquals("text/plain", connection.getContentType());
}
```

Continuous Integration

What is Continuous Integration?

Continuous Integration allows developers to automatically test and build the latest software artifact when there is a new commit. It continuously and proactively checks the codebase to find the latest code change and triggers a build accordingly(Jones, pg4). If all the test cases pass as expected and the build is successful, the continuous integration tool will deploy the latest artifact to an environment such as development, test, or production environments.

Purpose of Continuous Integration

It’s quite essential to learn about the reasons for using continuous integration for software development. Here are some reasons:

- It allows multiple software developers or teams to collaborate with one another, as it automatically builds the latest version of a software artifact developed by the team (Jones, pg4).
- It boosts developer productivity by automating the testing, building, and deploying pipeline so that developers don’t need to manually trigger a build or deploy (Jones, pg4).
- It helps developers to maintain different versions of the artifact and to check the build status of each version.

- It assists developers in finding out which commit breaks the build and which test cases fail so that potential bugs are detected in the early stage (Jones, pg5), preventing introducing them to real customers.
- It delivers the latest features to customers quickly if those updates pass all the tests and build successfully (Jones, pg6).
- It guides developers to locate bugs (Jones, pg5), as the continuous integration tool usually displays information about which commit breaks the test case with error logs.

Continuous Integration Tool

We signed up for Github Action as our CI/CD platform to help us automate the testing, building, and deploying process. With Github action, we can easily build workflows, with multiple sequential or parallel jobs, that are triggered by certain events such as new code changes being pushed to the codebase or a new pull request being created.

Set up Github Actions

First, we enabled Github Actions in GitHub repo as you can see from Figure 1.

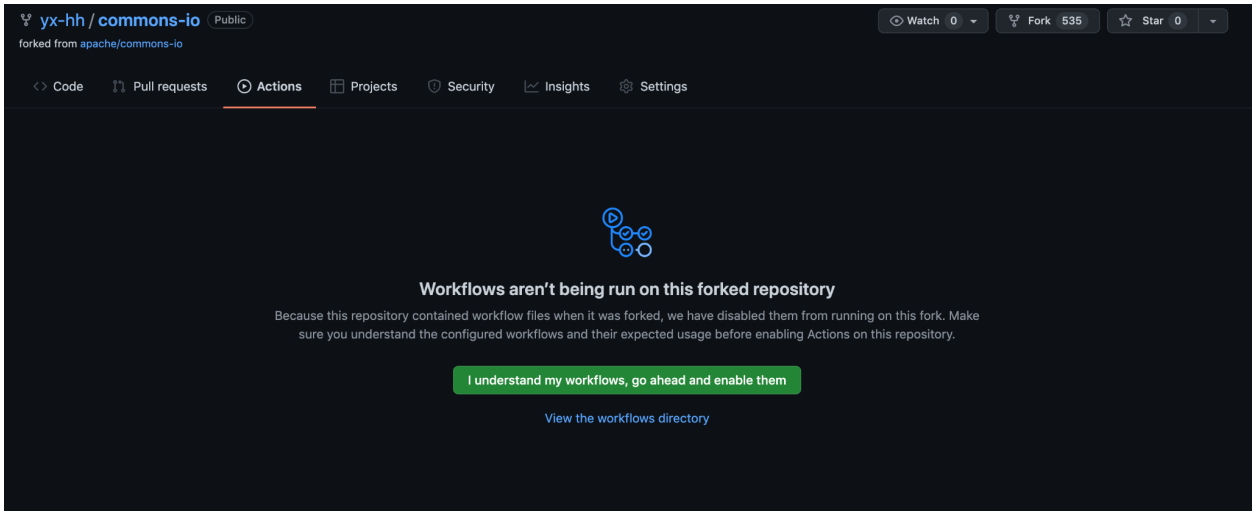


Figure 1: Enable Github Actions for our repository, From Github, By Changhao Liu

Then we created a Github Action configuration file: maven.yml under the folder of .github/workflows in our repository to configure Github Action. It checks whether there is a "push" event to our repository. If so, it will run the "**mvn package -Drat.skip=true"** command to run all the test cases, build and package our project sequentially: Github Actions will automatically execute the maven package command and upload jar files to the staging folder from which we can download our artifacts.

```yaml
# workflow name
name: Java CI

# listen to push operation to master

on:
 push:
  branches: [ master ]

# jobs need to execute when occur above operations
jobs:
 build:
  runs-on: ubuntu-latest # choose server

  steps:
   - uses: actions/checkout@v2
   - name: Set up JDK 8
     uses: actions/setup-java@v2
     with:
       java-version: '8'
       distribution: 'adopt'
   - name: Build with Maven
     run: mvn package -Drat.skip=true
   - run: mkdir staging && cp target/*.jar staging
   - uses: actions/upload-artifact@v2
     with:
       name: Package
       path: staging
```

Figure 2, Maven.yml by, By Yuxin Huang

By clicking a specific workflow in the Github Action's dashboard, we are able to check the detailed information of each build triggered by different commits in Figure 3. As for this workflow, the build was successful and took 6 minutes and 14 seconds to complete. To retrieve the artifact, we just need to click the "Package" button to retrieve our item. If we have a deployment environment for this project, we can also set up the workflow in Github Actions to deploy our artifact directly to our environments such as development, test, or production.

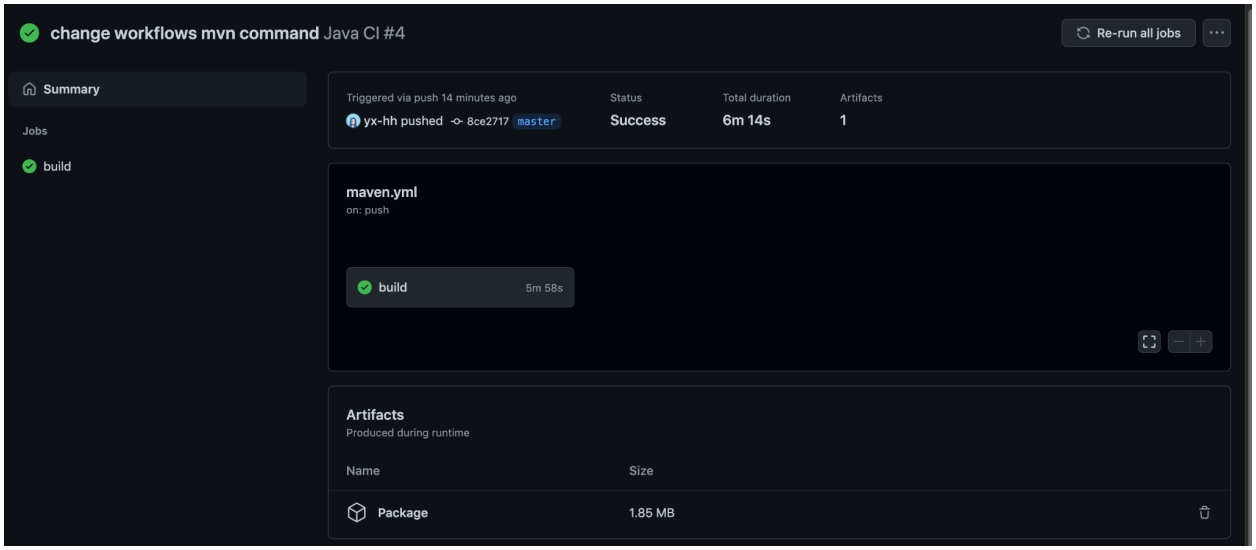


Figure 3, Build Success Info of Github Actions, By Yuxin Huang

By clicking the “build” button, we can see each workflow’s steps as in Figure 4.

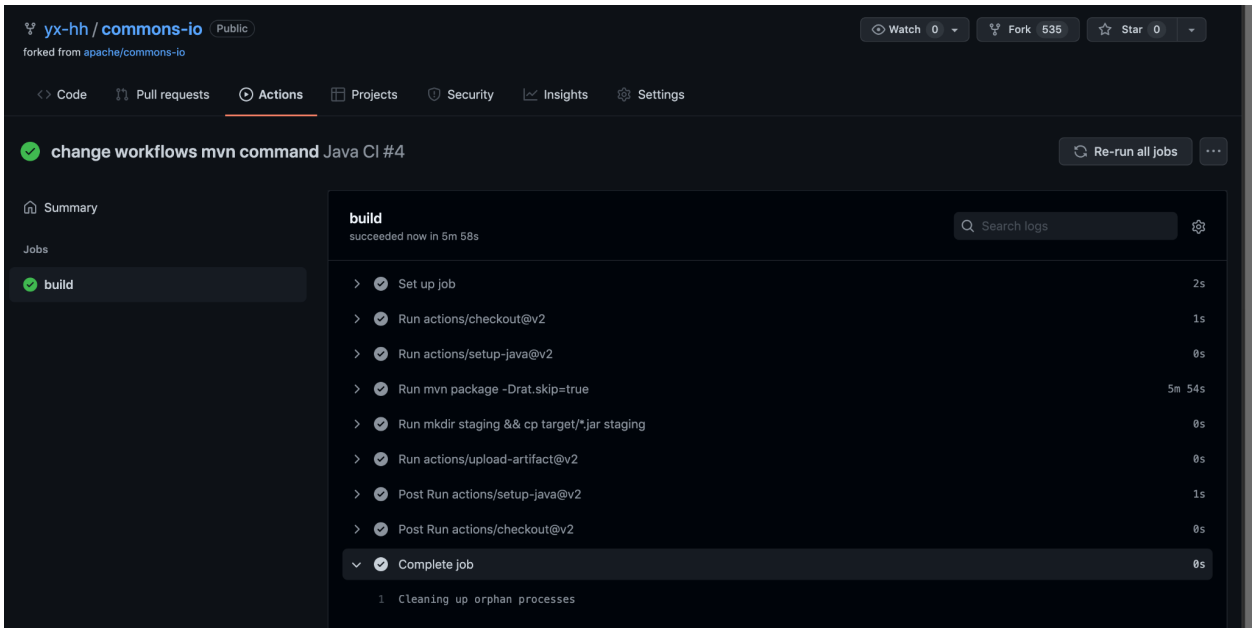


Figure 4, Workflow Jobs in Github Actions, By Yuxin Huang

To check the detailed log that records testing and build status as in Figure 5, we can read the log by unfolding the maven package command line.

```
Run mvn package -Drat.skip=true
[INFO] Running org.apache.commons.io.input.MessageDigestCalculatingInputStreamTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.135 s - in
org.apache.commons.io.input.MessageDigestCalculatingInputStreamTest
[INFO] Running org.apache.commons.io.input.UncheckedFilterReaderTest
[INFO] Tests run: 17, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.152 s - in
org.apache.commons.io.input.UncheckedFilterReaderTest
[INFO] Running org.apache.commons.io.input.CloseShieldInputStreamTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.093 s - in
org.apache.commons.io.input.CloseShieldInputStreamTest
[INFO] Running org.apache.commons.io.input.SequenceReaderTest
[INFO] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.147 s - in
org.apache.commons.io.input.SequenceReaderTest
[INFO] Running org.apache.commons.io.input.QueueInputStreamTest
[INFO] Tests run: 49, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.36 s - in
org.apache.commons.io.input.QueueInputStreamTest
[INFO] Running org.apache.commons.io.input.CircularInputStreamTest
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.109 s - in
org.apache.commons.io.input.CircularInputStreamTest
[INFO] Running org.apache.commons.io.input.TailerTest
```

```
[INFO] --- maven-jar-plugin:3.2.0:test-jar (default) @ commons-io ---
[INFO] Building jar: /home/runner/work/commons-io/commons-io/target/commons-io-2.12.0-SNAPSHOT-tests.jar
[INFO]
[INFO] --- maven-source-plugin:3.2.1:jar-no-fork (create-source-jar) @ commons-io ---
[INFO] Building jar: /home/runner/work/commons-io/commons-io/target/commons-io-2.12.0-SNAPSHOT-sources.jar
[INFO]
[INFO] --- maven-source-plugin:3.2.1:test-jar-no-fork (create-source-jar) @ commons-io ---
[INFO] Building jar: /home/runner/work/commons-io/commons-io/target/commons-io-2.12.0-SNAPSHOT-test-sources.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  05:42 min
[INFO] Finished at: 2022-02-24T00:09:12Z
[INFO] ------------------------------------------------------------------------
```

Figure 5, Test and Build Status, By Yuxin Huang

Problems we encountered

We faced a Maven command error which was caused by maven package command incorrect, and downloading some maven packages took a long time.

Testable Design

Introduction of Testable Design

Testable design is a modular design that avoids certain implementation styles which bring challenges to testers to write test cases. The goal of testable design is to make implementations easier to test (Davis, 2019).

Details of Testable Design

For testable design, we should

- Avoid complex private methods (Jones, page.17).
    - We cannot test with private methods, as we can hardly access them.
    - Private methods with complex logic may lead to some tricky bugs and are not able to be found by direct testing.

- Avoid static methods (Jones, page.18).
    - Static methods can only be called by the class instead of the object.
    - Static methods cannot be used flexibly, making functionality that has side effects or that has randomness difficult to test.

- Be careful of hardcoding in “new” (Jones, page.19).
    - We cannot stub objects if they are created using the “new” keyword.
    - An object reference can be retrieved using dependency injection instead.

- Avoid logic in constructors (Jones, page.20).
    - Constructors are hard to work around with, as a subclass's constructor always calls one of its superclass's constructors.

- Avoid Singleton Pattern (Jones, page.21).

Testable Design example

By testable design principles, we should avoid private methods with complex logic. In our project Commons-io, there is a method, named doGetFullPath (see Figure 1), which is under the

path /src/main/java/org/apache/commons/io/FilenameUtils. It is a private method that cannot be tested.

Original code:

```
private static String doGetFullPath(final String fileName, final boolean
includeSeparator) {
   if (fileName == null) {
       return null;
   }
   final int prefix = getPrefixLength(fileName);
   if (prefix < 0) {
       return null;
   }
   if (prefix >= fileName.length()) {
       if (includeSeparator) {
           return getPrefix(fileName);  // add end slash if necessary
       }
       return fileName;
   }
   final int index = indexOfLastSeparator(fileName);
   if (index < 0) {
       return fileName.substring(0, prefix);
   }
   int end = index + (includeSeparator ?  1 : 0);
   if (end == 0) {
       end++;
   }
   return fileName.substring(0, end);
}
```

Figure 1, FilenameUtils, by Ruokun Xu

To revise the method based on testable design principles, we changed the private into the public to make it testable.

Test case for Testable Design

Test case for getFullPath method by a string of file path and parameter of whether include the end separator. The following test case test different file path with their expected output.

```
@Test
public void testDoGetFullPath(){
   assertEquals("\\a\\b", FilenameUtils.doGetFullPath("\\a\\b\\c.txt", false));
```

```
   assertEquals("\\a\\b\\", FilenameUtils.doGetFullPath("\\a\\b\\c.txt",
true));
   assertEquals("D:/a/b/", FilenameUtils.doGetFullPath("D:/a/b/c", true));
   assertEquals("/a/b", FilenameUtils.doGetFullPath("/a/b/c", false));
}
```

Mocking and its Utilities

Mocking uses fake objects to check whether unit tests pass or fail by observing the interactions among objects(Jones, pg 23). It is a technique that helps developers to test components without relying on their actual dependencies, as mock objects substitute those dependencies and simulate behaviors (Hall, StackOverFlow, 2010).

To fully understand the benefit of mocking, we should first learn more about some challenges in integration testing. With integration testing, developers and testers check some dependent software components in a group to verify software quality and look for potential integration issues (Jones, page 10). However, integration testing may face a few challenges:

- When an issue arises, it is hard to detect which dependency causes the error.
- It is difficult to test the interactions between software modules when some parts are not implemented yet.
- Some testing may change the state of other components such as databases, leading to extra work to reset to the original state.

With the help of mocking, developers tackle those problems as mocking has a few benefits:

- Mock objects can simulate the behaviors of some real objects
- Mocking allows developers to develop and write integration tests at the same time
- Mock objects won't affect the real object state, thus avoiding side-effects

Tested a Feature Using Mockito

As you may recall, our project Apache Commons IO is a library of utility features helping developers deal with IO. The feature, in the PathFileComparator class, that we picked is to compare two files' paths depending on the case sensitivity. It should be noted that it has dependency on File class, which makes it challenging to test.

Without mocking, it is difficult to test this feature for the following reasons:

- We had to create new files in the current directory and to call File class's constructor for the setup
- We had to find out which folder authorizes us to write those testing files
- We had to spend a lot of time writing logic in creating and cleaning testing files
- We had to allocate some disk space for those testing files, which is a waste of space

Thanks to Mockito, a mocking framework that allows us to write clean and lightweight test cases using its APIs, we were able to test our features and resolve all the issues that we mentioned before. With Mockito, we can easily mock File class using `Mockito.mock(File.class)` syntax. And assign expected behaviors to the mock objects using `.when` and `.thenReturn`. For example, Mockito.when(file1.getPath()).thenReturn("test/file1.txt") allows file1 mock object to return a string with value of "test/file1.txt".

Therefore, we successfully tested our feature using Mockito and enjoyed a few benefits:

- There was no need to create actual files, as we mocked the File class.
- We didn't need to clean any actual files after running our test cases
- Increased our productivity and saved development time, as we didn't add any testing files in any folders
- Specified the behavior of mocking object to test our feature without worrying about the internal implementation of dependent components.

Test cases that we wrote in Mockito

Here is a code snippet that we utilized Mockito to test this feature:

- Used Mockito to mock different files and to compare the path of two files.
- Tested case-sensitive comparator and case insensitive comparator and expected compare function's output.

```
@Test
public void testCompare(){
   final File file1 = Mockito.mock(File.class);
   Mockito.when(file1.getPath()).thenReturn("test/file1.txt");
   final File file2 = Mockito.mock(File.class);
   Mockito.when(file2.getPath()).thenReturn("test/file1.txt");
   final File file3 = Mockito.mock(File.class);
   Mockito.when(file3.getPath()).thenReturn("TEST/file1.txt");
   final File file4 = Mockito.mock(File.class);
   Mockito.when(file4.getPath()).thenReturn("tes/file1.txt");

   final Comparator<File> sensitiveComparator = new PathFileComparator();
```

```
   assertEquals(0, sensitiveComparator.compare(file1, file2), "sensitive file1
& file2 = 0");
   assertTrue(sensitiveComparator.compare(file1, file3) > 0, "sensitive file1 &
file3 > 0");
   assertTrue(sensitiveComparator.compare(file1, file4) > 0, "sensitive file1 &
file4  > 0");

   final Comparator<File> insensitiveComparator =
PathFileComparator.PATH_INSENSITIVE_COMPARATOR;
   assertEquals(0, insensitiveComparator.compare(file1, file2), "insensitive
file1 & file2 = 0");
   assertEquals(0, insensitiveComparator.compare(file1, file3), "insensitive
file1 & file3 = 0");
   assertTrue(insensitiveComparator.compare(file1, file4) > 0, "insensitive
file1 & file4 > 0");
   assertTrue(insensitiveComparator.compare(file3, file4) > 0, "insensitive
file3 & file4  > 0");
}
```

Static Analysis

Introduction to Static Analysis

Before discussing static analysis tools, we should first learn more about what static analysis is and its purposes. Static analysis, as its name implies, is to seek for potential bugs in software programs without running the program. Its goal is to identify potential issues in the program in the early stage.

There are several ways to conduct static analysis and one of them is through code review, which is to invite other developers to examine the codes. Usually, developers will utilize their past experience and intelligence to provide some feedback and may find some flaws. Such practice allows the development team to find and fix bugs in early stages. There are a few good practices for code reviews (Jones, pg 3):

- Review small chunks of the program at a time(Jones, pg 3)
- Add comments and feedback(Jones, pg 3)
- Review the code individually(Jones, pg 3)
- Create some checklist for essential actionable items(Jones, pg 3)

Besides code review by developers, another way of static analysis is to utilize some automatic static analysis tools, such as Checkstyle or SportBug etc. We will discuss the usage of those tools

in the next section. However, some tools may display some warnings that may not be actual problems (Jones, pg 6), so we should be critical about the result and take actions accordingly.

Benefits of Static Analysis

There are some benefits brought by static analysis. Here is a list of benefits that we concluded (Hamilton, 2022):

- Detect potential issues early
- Make good use of every developer's work experience and intelligence to locate defects in design or implementation
- Reduce time and cost for testing
- Facilitates developers' teamwork in the project
- Makes the codebase maintainable, as review checks whether the new codes are consistent with the coding style

Two Different Static Analyzers' usage

Checkstyle - Static Analyzer

It checks whether Java code is consistent with a specific coding standard in different modes such as Sun Checks Mode or Google Checks Mode..

1. Sun Checks Mode

We first installed the Checkstyle plugin and then clicked Analyze -> Inspect Code. Results are as Figure

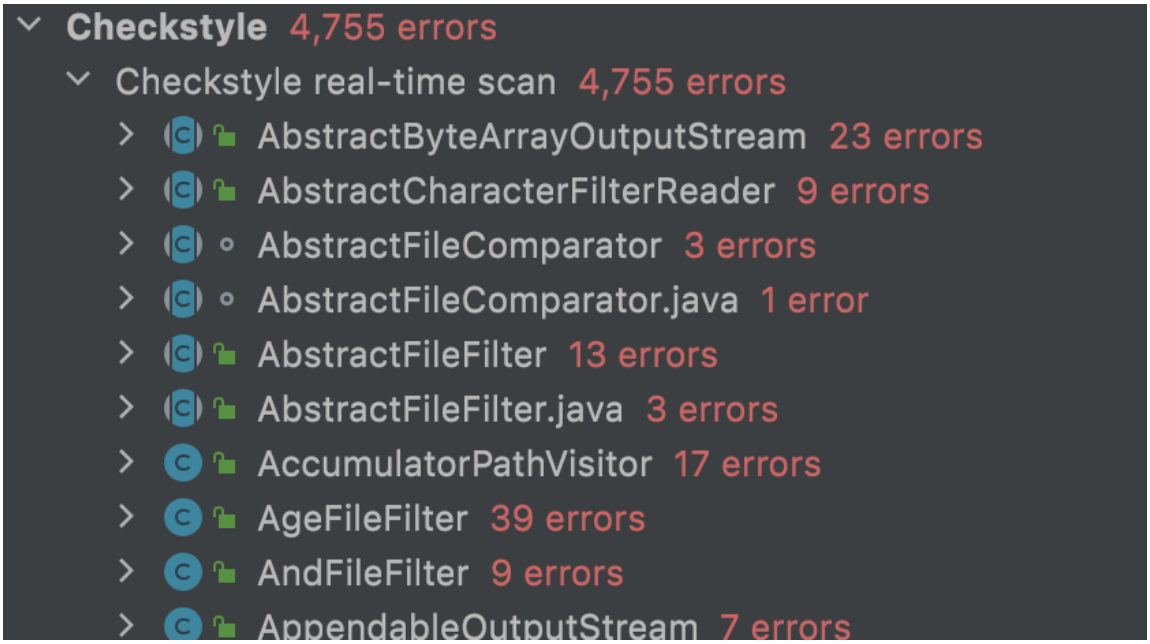


Figure 1, Sun Checks result, by Ruokun Xu

- New line: '+' should be on a new line when nothing follows '+'. It is not a problem because the position of '+' will not affect results when executed. See Figure 1-1 below.

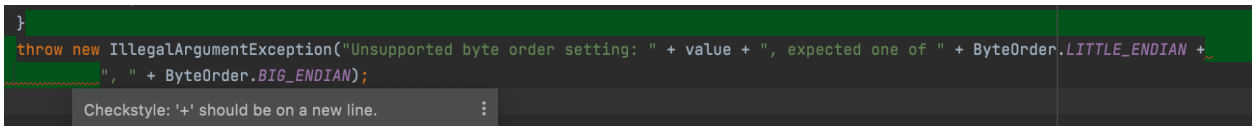


Figure 1-1, new line, by Ruokun Xu

- Long line: Line is longer than 80 characters. It is not a problem because the length of each line will not affect results when executed. See Figure 1-2 below.

Figure 1-2, long line, by Ruokun Xu

- Declaration: Cass should be declared as final. It is not a problem because the declaration of final will not affect results when executed. See Figure 1-3 below.

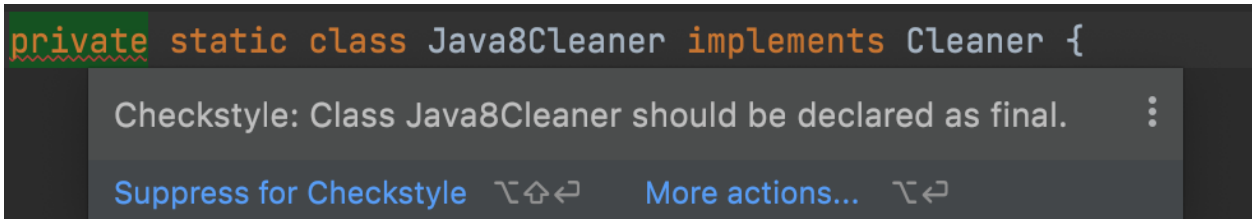


Figure 1-3, declaration, by Ruokun Xu

2. Google Checks Mode

Click Analyze -> Inspect Code. Results are as Figure 2.

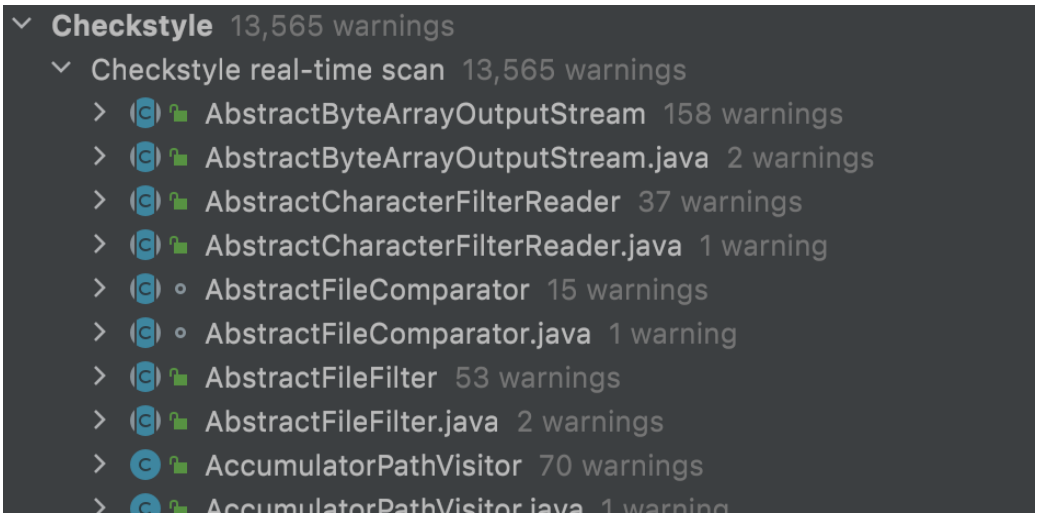


Figure 2, Google Checks result, by Ruokun Xu

- Incorrect indentation: the expected indentation is 2 tabs, and we got an error warning because there are 4 tabs. It is not a problem because the number of tabs will not make an error when executed. See Figure 2-1 below.

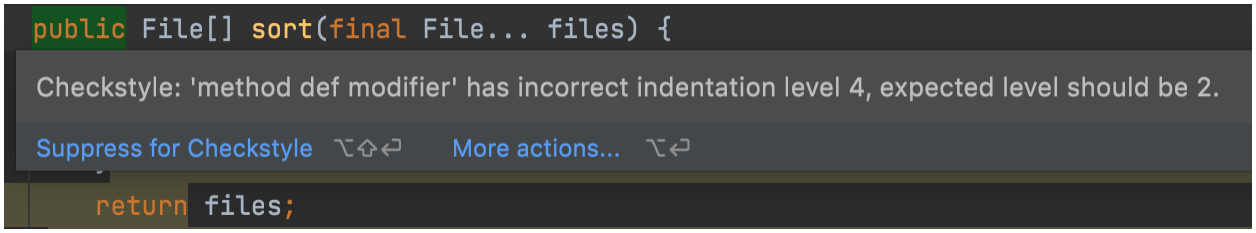


Figure 2-1, indentation, by Ruokun Xu

- Package position: ‘package’ should be isolated. It is not a problem because the position of the package line will not make errors when executed. See Figure 2-2 below.

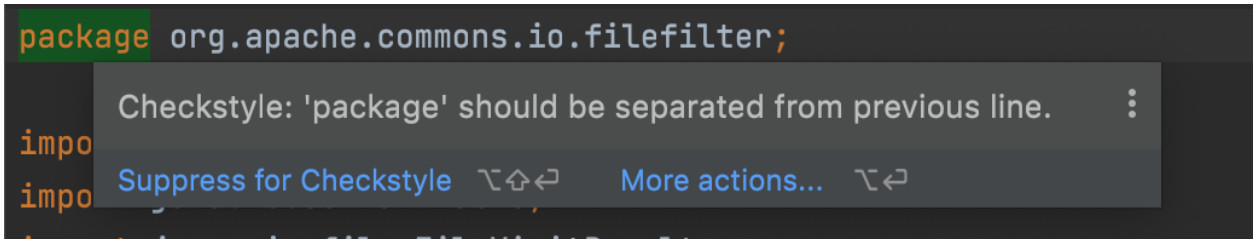


Figure 2-2, line position, by Ruokun Xu

- White space: white space should be added. It is not a problem because missing whitespace will not make an error when executed. See Figure 2-3 below.

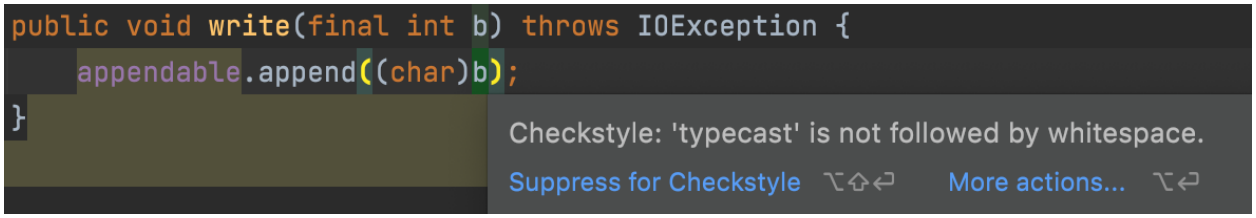


Figure 2-3, whitespace, by Ruokun Xu

Infer - Static Analyzer

Infer is a static analysis tool that examines your Java or C/C++/Objective-C code and identifies some potential issues. Infer allows developers to locate and fix bugs before shipping them to production. Also, it prevents potential crashes or poor performance in the program (*Infer Static Analyzer*, 2022).

We ran Infer our Commons-IO project to check some potential issues, including null pointer exceptions, resource leaks, annotation reachability, missing lock guards, and concurrency race conditions (*Infer Static Analyzer*, 2022).

Install Infer in MacOS system
Infer only supports Intel Chip. We can execute **brew install infer** command in terminal to install this software.

Run Infer for Maven Project
We can run **infer -- mvn package -Drat.skip=true -PMaster** to analyze the master branch

Report by Infer:
Figure 3 presents the **report.txt** under the folder of **infer-out**

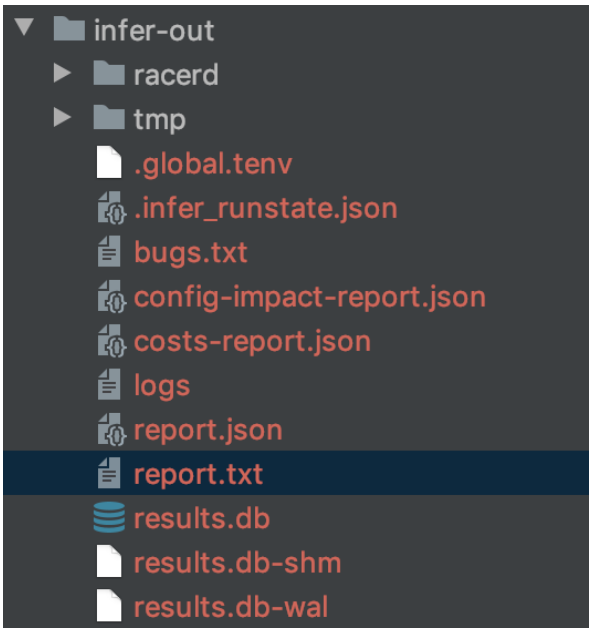


Figure 3, report location, by Yuxin Huang

There are a total of 100 issues and warnings found in common-io projects, including 56 Thread Safety violations, 4 Resource Leak warnings, and 3 Null Dereference. Figure 4 illustrates nume of issues and issue types reported by Infer. Figure 5 is a detailed example of an issue that Inter found: It displays the serial number of the issue, the issue type and detailed code line with the problem in order.

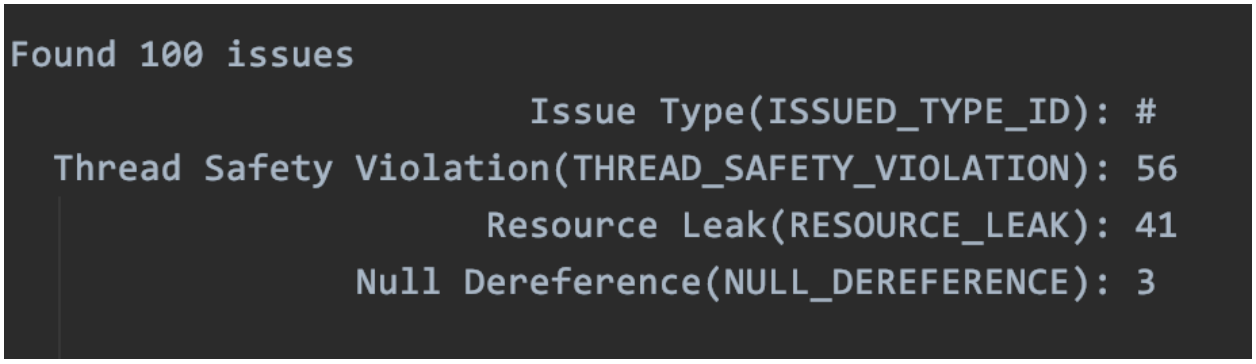


Figure 4, issue summary, by Yuxin Huang

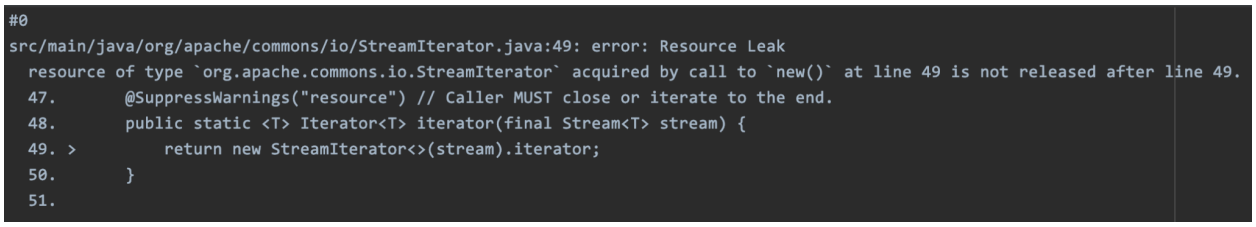


Figure 5, issue style, by Yuxin Huang

Issue example

Thread Safety Violation Example

Figure 6 is an example of Thread Safety Violation found by Infer. It points out that a non-private method writes `this.pos` field out of synchronization.In figure 7, we can see that pos is a private field of the BoundedInputStream class. For the threads inside of this class, `pos` field is a public variable. If two threads operate on this `pos` at the same time, it will have a Thread Safety Violation issue.

```
#89
src/main/java/org/apache/commons/io/input/BoundedInputStream.java:220: warning: Thread Safety Violation
  Unprotected write. Non-private method `BoundedInputStream.skip(...)` writes to field `this.pos` outside of synchronization.
 Reporting because another access to the same memory occurs on a background thread, although this access may not.
  218.           final long toSkip = max>=0 ? Math.min(n, max-pos) : n;
  219.           final long skippedBytes = in.skip(toSkip);
  220. >         pos+=skippedBytes;
  221.           return skippedBytes;
  222.       }
```

Figure 6, Thread Safety Violation Issue, by Yuxin Huang

```
/** the number of bytes already returned */
private long pos;
```

Figure 7, pos field location, by Yuxin Huang

Resource Leak Example

Figure 8 is an example of Resource Leak reported by Infer. It points out resources obtained by sw are not released: It may cause resource leak. This is an actual issue because sw is an object of CharArrayWriter, which will use IO resources if not closed after using.

```
#43
src/main/java/org/apache/commons/io/IOUtils.java:2770: error: Resource Leak
  resource of type `java.io.CharArrayWriter` acquired to `sw` by call to `new()` at line 2768 is not released after line 2770.
**Note**: potential exception at line 2769
  2768.           final CharArrayWriter sw = new CharArrayWriter();
  2769.           copy(reader, sw);
  2770. >         return sw.toCharArray();
  2771.       }
  2772.
```

Figure 8, Resource Leak Example Issue, by Yuxin Huang

Null Dereference Example

Figure 9 presents an example of the Null Dereference identified by Infer. It points out that countDirectory (directory) can be null and will cause Null Dereference issue. This issue will happen when the countDirectory method returns a null value and raises a NullPointerException.

```
#39
src/main/java/org/apache/commons/io/file/PathUtils.java:1459: error: Null Dereference
  object returned by `countDirectory(directory)` could be null and is dereferenced at line 1459.
  1457.         */
  1458.        public static long sizeOfDirectory(final Path directory) throws IOException {
  1459. >          return countDirectory(directory).getByteCounter().getLong();
  1460.        }
  1461.
```

Figure 9, Null Dereference Example Issue, by Yuxin Huang

Static Analysis Tools Comparison

Two tools that we used, Checkstyle and Infer are fundamentally different in their purposes

- Checkstyle aims to check code format and with different standards, including length of line, declaration, and indentation.
- Infer aims to check null pointer exceptions, resource leaks, annotation reachability, missing lock guards, and concurrency race conditions

Reference

Jones. (2022a). *Code Review, and Automated Static Analysis Tools* [Slides]. Canvas. https://canvas.eee.uci.edu/courses/43617/files/folder/Lecture%20Slides?preview=17870946

Hamilton. (2022). *What is Static Testing? What is a Testing Review?* Guru99. https://www.guru99.com/testing-review.html

Infer Static Analyzer. (2022). Infer. https://fbinfer.com/

Davis, H. (2019, October 5). *What is a testable design? – QuickAdviser*. Harry Davis. https://quick-adviser.com/what-is-a-testable-design/

Jones, J. (2022b). Integration_Testing_Mocking_Testable_Design [Slides]. Canvas. https://canvas.eee.uci.edu/courses/43617/files/folder/Lecture%20Slides?preview=17765481

What is Mocking? (2010, April 19). Stack Overflow. https://stackoverflow.com/questions/2665812/what-is-mocking

Jones, J. (2022b). *Continuous Integration* [Slides]. Canvas. https://canvas.eee.uci.edu/courses/43617/files/folder/Lecture%20Slides?preview=17655785

Building and testing Java with Maven. (2012). GitHub Docs. https://docs.github.com/en/github-ae@latest/actions/automating-builds-and-tests/building-and-testing-java-with-maven

Structural Testing Tutorial - What Is Structural Testing. (2022, February 3). Software Testing Help. https://www.softwaretestinghelp.com/structural-testing-tutorial/

Hamilton, T. (2021, December 18). What is WHITE Box Testing? Techniques, Example & Types. Guru99. https://www.guru99.com/white-box-testing.html

Pezzè Mauro, and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, 2008.

Bors, Mátyás Lancelot. *"What Is a Finite State Machine?"* Medium, Medium, 10 Mar. 2018, https://medium.com/@mlbors/what-is-a-finite-state-machine-6d8dec727e2c.

Hamilton, T. (2022, January 1). *Model based testing tutorial: What is, Tools & Example. Guru99.* Retrieved February 9, 2022, from https://www.guru99.com/model-based-testing-tutorial.html#:~:text=one%20by%20one%3A-,Finite%20State%20Machines,inputs%20given%20from%20the%20testers

Apache Commons IO - Overview. (n.d.). Retrieved February 2, 2022, from https://www.tutorialspoint.com/commons_io/commons_io_overview.htm

15 Functional Testing Types with Examples - Applause, from
https://www.applause.com/blog/functional-testing-types-examples

Equivalence Partitioning Method - GeeksforGeeks, from
https://www.geeksforgeeks.org/equivalence-partitioning-method/

Equivalence partitioning - Wikipedia, from
https://en.wikipedia.org/wiki/Equivalence_partitioning

Functional Testing - Wikipedia, from
https://en.wikipedia.org/wiki/Functional_testing