CS4218 Software Testing

Milestone 2

Test-Driven Development

Execution

- We found out that the provided test cases are much more comprehensive in terms of the positive scenarios. Yet, our own test cases have better coverage on the negative scenarios (such as nullish inputs).
- As such, we decided to keep both sets of test cases to be used for TDD. Note: *src/test/java/tdd* contains the provided test cases while *src/test/java/.../cs4218* contains our own test cases.
- Unsurprisingly, there were numerous compilation issues we had to first resolve among the provided test cases before we can start TDD. These issues stem from the difference in implementation such as having the *setCurrentDirectionary* method or having some methods that throw exceptions but the provided test cases were not written to catch those exceptions.
- Thereafter, we split our team into 2 groups for TDD. The 1st group is in charge of performing TDD on the implemented functionalities from MS1 (BF+EF1) and the focus is on sieving out and fixing any newly identified bugs. The 2nd group is to perform TDD on the unimplemented functionalities (EF2), focusing on the implementation and following a workflow that aligns more closely with a typical TDD cycle.

Test Evaluation

- When we first added the provided test cases, we had many test cases on implemented functionalities that failed (~100).
- Upon inspection, we realized that the provided test cases failed due to differences in the assumed specifications or even bugs in the test cases themselves. In other words, our program outputs our own expected result, but this result differs from what was expected from the provided test cases.
- Hence, some of the provided test cases as well as our implementations were modified to resolve the differences.
- See Test Evaluation Result to view the differences and actions taken to resolve the issues:

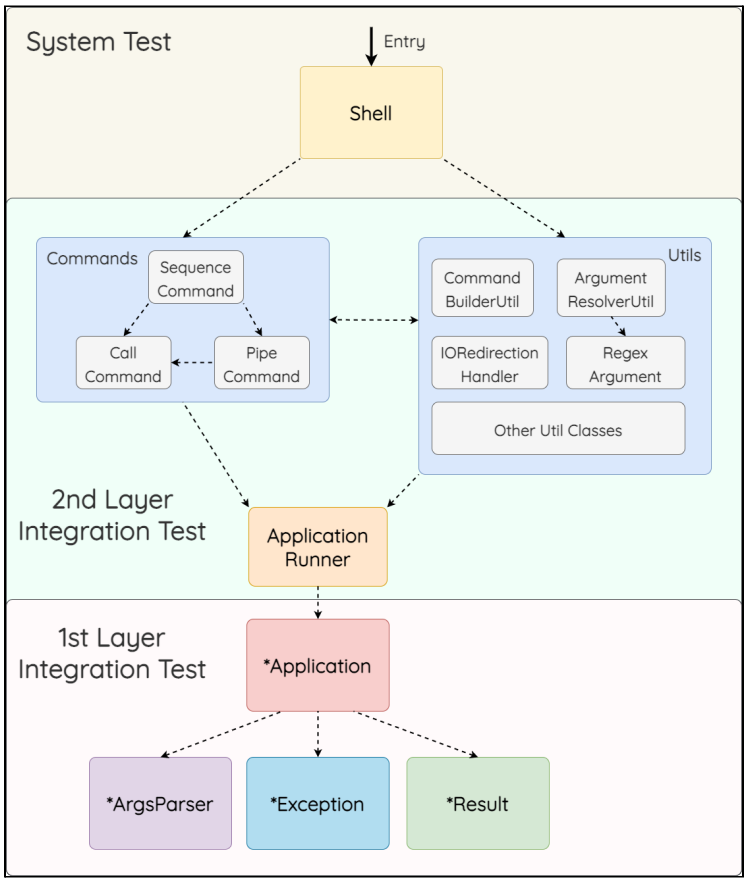
Reflection

- For a start, our team struggled quite a bit to get the TDD process going. In particular, it took some of us quite some time to investigate why our implemented features were failing despite passing our own test cases.
- Additionally, while implementing the leftover features, we had also encountered another issue that could have been avoided if we had first evaluated the provided test cases against the project specifications, before jumping straight into the implementations.
- We had failed to consider the fact that the test cases could be wrong or different assumptions were made instead. As such, while doing TDD, we would sometimes find that the test cases were leading us to implement specifications which were different from what we had understood.
- All in all, we feel that TDD is a very tedious process, especially for those who are trying it for the first time. There is also a need to ensure that the test cases are coded according to the specifications or

we might end up implementing a slightly different functionality. Nonetheless, one of the benefits of TDD is that once we are done with the implementations, the tests are also done :D.

Integration & System Testing

Component Hierarchy



Implementation Plan

- The above diagram shows the dependency graph for the main components.
- The graph sort of has a vertical structure and there are less coupling at the bottom of the graph.
- As such, we have decided to integrate layer by layer from bottom up as we believe that this can better help to isolate/identify the cause of any potential bugs that could surface from the integration.

Testing Plan

- As the unit tests already cover extensively on structural testing/coverage. For integration testing, we decided to focus more on functional testing and ensure that the program aligns with the specification and edge cases are well-handled (doesn't crash the program).
- For test generation:

- Firstly, we experimented with the sample test cases provided in the project specification document, performing integration tests for the components whose dependent modules were previously mocked. E.g. command classes.
- Next, we executed pairwise testing among the applications and command operators.
- Finally, we verify that negative cases are also added as well to test if appropriate error messages are output.

Pairwise testing for pipe command

During the planning stage, we first identified all possible pairs of commands that are reasonable to be connected with a pipe. Considering a pipe command in the form of "app1 | app2", a pipe command that is worth testing is where app1 writes to standard out and app2 reads from standard in as this ensures there will be interaction between app1 and app2. On the other hand, it is not so meaningful to extensively test commands like as "cd | exit" as there is no interaction between the two apps. Besides the positive cases, we also tested on negative cases to verify that "app2" will not be executed if "app1" throws an exception.

Pairwise testing for sequence command

For sequence commands, we focused more on testing combinations of commands that are not extensively covered in the pairwise tests for the pipe command (i.e. cd, mv, rm, cp, ls). For these commands, there is usually little or IO interaction among them. However, their orderings in the sequence command can significantly affect the shell output. Besides positive cases, we have also tested negative cases such as invalid commands like "rm file1.txt; cat file1.txt" and verify that the correct error message is output.

Testing Tools

Mockito

- Mockito greatly simplifies the process of creating stubs an.d makes some complicated test cases more human-readable.
- Below is an example from PipeCommandTest.java. We would like to test that PipeCommand stops executing the remaining commands when an exception is thrown in the currently executing command.

```
@Test
public void evaluate_AppExceptionThrown_RestTerminated() throws AbstractApplicationException, ShellException {
    CallCommand command1 = mock(CallCommand.class);
    CallCommand command2 = mock(CallCommand.class);
    doThrow(CatException.class)
            .when(command1)
            .evaluate(eq(stdin), any());

    PipeCommand command = new PipeCommand(List.of(command1, command2));

    assertThrows(CatException.class, () -> {
        command.evaluate(stdin, stdout);
        verify(command1).evaluate(eq(stdin), any());
        verify(command2, never()).evaluate(any(), eq(stdout));
    });
}
```

- In order to test this behaviour independent of the implementation of various CallCommand, without Mockito, we would have to manually create a CallCommand stub class which is only used

once in this method. That means, it is possible that we have to create several stubs of the same class to fulfil the need of different test cases.

- With Mockito, we can use methods like mock(), doThrow() to define a stub with simple behavior with only two lines. We can use verify() to check whether the expected method calls are made.
- We have also used other Mockito features like spies and argument capturing in other test cases of the project.

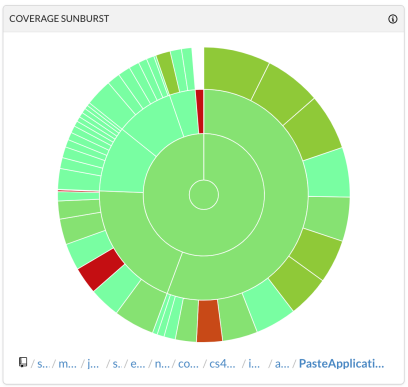
Gradle Build Scan

- Using Gradle, we have automated the PMD static code analyzer to run during tests or builds. As such, whenever there are any PMD warnings, tests/builds will fail and we will be alerted.
- Additionally, we have included Gradle Build Scan that will automatically generate a test report to provide insights to our testing (see https://gradle.com/s/b5vkl4yzrdyyc). This has proven to be extremely useful as some of our team members have sometimes relied on the test report to locate bugs.
- In particular, we have successfully identified the following bug using this tool:

Report	Bug	Cause	Buggy code fragment	Fix
Link	Test cases on listing files/directories with ls passed when run locally. However, the tests **sometimes** failed when run on other/remote platforms (E.g. Travis).	The build scan report revealed that the order of the listed files is not always deterministic. When run locally, the listed files are of a particular order and thus it passes the test cases. Yet, when run on Travis, a different order is output and thus the test cases failed.	See	Output files are sorted in lexicographical order to improve testability. See

Codecov + Jacoco

- Finally, our team has also included Codecov for a deeper analysis of our code coverage. The code coverage provided by this platform is much more comprehensive (based on Jacoco) than the one provided in Intellij. See https://app.codecov.io/gh/nus-cs4218/cs4218-project-ay2021-s2-2021-team12 (may need admin access to cs4218 repo).
- Codecov's coverage graph helped to facilitate structural testing as it highlights the coverage density of our program.



Appendix

PMD Violations Justifications

Violated Rule	Files	Justification
GodClass	Various *Application classes.	Implementation logic could be complicated and long. However, we have tried to abstract the logic into various smaller helper methods.
CloseResource	When declaring input/output streams.	Streams are closed in nested functions or sometimes the rule might return false positives. (see https://github.com/pmd/pmd/issues/1911)
AbstractNaming	AbstractApplicationExceptionTest class.	This is the driver test class for AbstractApplicationException. As such, it should not be declared abstract.
ExcessiveMethodLength	Some test/impl classes.	Multiple assert clauses + closures belonging to the same test category increase the method length significantly. Methods with many switch/conditional cases.
LongVariable	Entire project.	Variable names should be meaningful and reveal intention.
AvoidStringBufferField	RegexArgument class.	StringBuilder is highly recommended for efficient concatenation of multiple strings.

Test Evaluation Result

Source / Component	Program (mostly according to default unix behaviour)	Provided test cases	Resolution
App error output	All error messages are written to stderr.	Some error messages are written to stdout.	Follow default unix behaviour
cat app	Prefixed line numbers will reset when cat multiple files. E.g. <file1> 1 … 2 … … <file2> 1 … 2 …	Prefixed line numbers do not reset when cat multiple files. E.g. <file1> 1 … 2 … … <file2> 49 … 50 …	Follow default unix behaviour

	...	...	
ls app	No output when ls on a single nested empty directory.	Output the nested directory's name but with empty files.	Follow default unix behaviour
ls app	Output files are not sorted in any particular order. Non-deterministic output order.	Output files are sorted in lexicographical order. Deterministic output order on different platforms (improves testability).	Follow provided test cases.
ls app	When given absolute paths, output shows absolute paths.	When given absolute paths, output shows relativized paths.	Follow default unix behaviour
wc app	Uses a single tab character as delimiter. Not default unix behaviour but the output looks neater.	Uses multiple whitespaces as delimiter.	Use tab character as delimiter
wc app	Output filename label when read from stdin is empty.	Output filename label is "-" when read from stdin.	Follow default unix behaviour
split app	As all error messages are written to stderr, output stream is not required for this app and no exception will be thrown.	Output stream is required or an exception will be thrown.	No exception thrown if output stream is not given.

Pairwise test creation

After identifying important combinations of commands to test, we made the following table to guide our test creation for pipe commands.

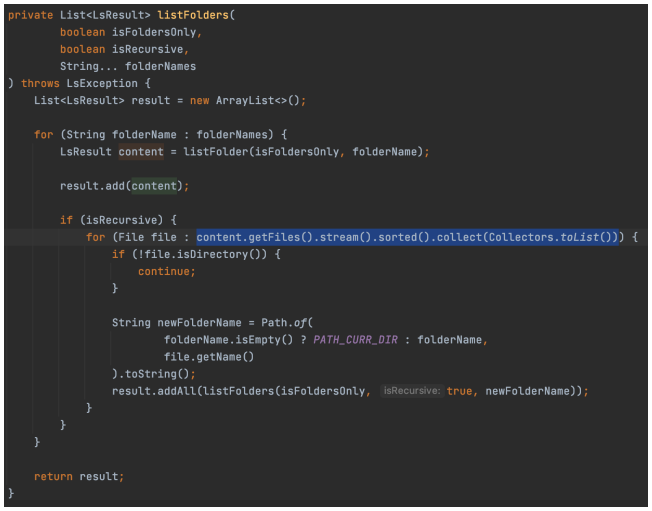
Application	Write to stdout?	Read from stdin?
Echo	Y	
Ls	Y	
Wc	Y	Y
Cat	Y	Y
Grep	Y	Y
Exit		
Tee	Y	Y
Cd		
Split		Y

Mv		
Uniq	Y	Y
Rm		
Paste	Y	Y
Cp		

Buggy Code Fragment and Fix

```
private List<LsResult> listFolders(
        boolean isFoldersOnly,
        boolean isRecursive,
        String... folderNames
) throws LsException {
    List<LsResult> result = new ArrayList<>();

    for (String folderName : folderNames) {
        LsResult content = listFolder(isFoldersOnly, folderName);

        result.add(content);

        if (isRecursive) {
            for (File file : content.getFiles()) {
                if (!file.isDirectory()) {
                    continue;
                }

                String newFolderName = Path.of(
                        folderName.isEmpty() ? PATH_CURR_DIR : folderName,
                        file.getName()
                ).toString();
                result.addAll(listFolders(isFoldersOnly,  isRecursive: true, newFolderName));
            }
        }
    }

    return result;
}
```

Buggy code



```
private List<LsResult> listFolders(
        boolean isFoldersOnly,
        boolean isRecursive,
        String... folderNames
) throws LsException {
    List<LsResult> result = new ArrayList<>();

    for (String folderName : folderNames) {
        LsResult content = listFolder(isFoldersOnly, folderName);

        result.add(content);

        if (isRecursive) {
            for (File file : content.getFiles().stream().sorted().collect(Collectors.toList())) {
                if (!file.isDirectory()) {
                    continue;
                }

                String newFolderName = Path.of(
                        folderName.isEmpty() ? PATH_CURR_DIR : folderName,
                        file.getName()
                ).toString();
                result.addAll(listFolders(isFoldersOnly,  isRecursive: true, newFolderName));
            }
        }
    }

    return result;
}
```

Fix