CS4218 Milestone 1 Report

Implementation and Testing Plans

- Before we could start working on the implementation, we discovered that there were existing bugs in the codebase, which prevented a full run of the Shell program. We managed to resolve the bugs with the help of a debugger and analyzing the stack traces.
- After which, our implementation plan involves writing an application with its associated unit tests together. We split the features amongst us based on an estimated equal workload.
- When a bug has been flagged out by any member, tests will be written to cover the bug scenario, which helps to improve test coverage.
- We also made sure to go through the code review process, just so that we can help each other look out for edge cases in the implementation, as well as in the writing of the test cases.

How Test Cases Are Generated

- Due to the numerous number of possibilities of the argument configurations for each application, enumerating all of them will be a tedious process. Instead, we used the pairwise testing approach, whereby we tested different combinations of arguments for each application.

Tools Used For Testing

- IntelliJ's 'Run Tests With Coverage' feature, which allows us to see at a glance, which of the lines in our implementation are covered and which are not, allowing us to write test cases to achieve higher statement coverage.
- We also use the Mockito library to mock some of the interactions between components, such as mocking IOExceptions or to assert that some function returns us a desired output for several of the tests.
- We also used the static analyzers enabled in IntelliJ, which were helpful in finding dead code, unreachable statements as well as flawed implementation logic.
- We used Github Actions to set up a basic CI flow that runs the tests on a pull request, allowing us to quickly check that no new code additions or fixes break existing functionalities.

Techniques Used For Testing

- We use a mix of systematic partitioning to generate our tests.
- We also try to achieve branch coverage of the implementation code, to explore every possible path of conditional statements, loops and branch constructors at each line.
- We also tested extensively for negative cases, such as cases when the arguments are null or invalid, existence of files etc.
- For applications that did not directly return an observable input that could be verified, we used mocks, and verified the state of it, such as verifying that it leads to another function call with the correct arguments for example (***CallCommand***).

- For applications that require working with reading/creating files, we either create them in the tests resources folder to be used directly, or we create them on the fly, and then clean up these files after the tests are run.
- We have 2 different test classes for each Application, the unit tests and the integration tests. The unit tests focus on ensuring that each implemented method of the interface works as intended and returns the correct results. On the other hand, the integration tests are being run like a black box, ensuring that the correct methods in the class are being invoked with the correct arguments. The unit tests serve to verify that the interfaces are implemented correctly based on its specifications and requirements.
- To ensure us to pinpoint the exact cause of failure, we ensured that whenever an exception was thrown, we verified the cause of the exception with its message, as different portions of the code in the same class could be throwing the same exception. This is essential as the error messages thrown are the feedback returned back to the user.
- For integration tests, since it has a lot of overlaps with the unit tests, our focus is on ensuring the handling of arguments, invalid as well as edge cases, ensuring that the correct arguments are fed to the correct handler methods and that the correct exceptions are thrown at different stages of evaluation.
- For methods that do not produce any output and return `void`, we used mockito to mock the calls, and use `ArgumentCaptors` and `verify` methods to assert the interaction between the caller and callee.

Challenges Faced In Testing

- Cleaning up the tests was one of the difficulties that we faced. For applications that require creating files, we mitigate this by creating a temporary `temp` folder that would be used for the tests, and then deleting this `temp` file after the tests are run. For applications that change the system environment (***CdApplication***), we used a similar approach. We store the original environment path before we run all the tests, and then we restore it after the tests are run.
- Another obstacle we encountered involved ensuring cross-platform compatibility across Linux, macOS, and Windows when comparing strings for the expected path. We addressed this challenge by leveraging the Path library to construct the expected output, allowing us to compare paths accordingly.

Test Cases That Helped Find Initial Faults

#1

```
    @Test
    void extractRedirOptions_nullArgsList_shouldThrowException() {
        IORedirectionHandler redirHandler = new IORedirectionHandler(null, System.in, System.out,
new ArgumentResolver());
```

```
        ShellException exception = assertThrows(ShellException.class,
redirHandler::extractRedirOptions);
        assertEquals(SYNTAX_ERROR_MSG, exception.getMessage());
    }
```

This was one of the test cases that helped us find fault in the condition `if (argsList == null && argsList.isEmpty())` inside `IORedirectionHandler`. Hence, we changed the `&&` condition to `||` condition instead: `if (argsList == null || argsList.isEmpty()).`

#2

```
    @Test
    void isBlank_multiWhitespaces_shouldReturnTrue() {
        assertTrue(StringUtils.isBlank("     "));
    }
```

This was another simple test case that allowed us to discover that the `StringUtils.isBlank()` method was running in an infinite loop, instead of returning true.

Summary of Test Cases

- Negative test cases generally involve validating against incorrect or non-existent files, or null arguments.
- On the other hand, positive test cases encompass various scenarios with valid inputs, including different flags, file paths, standard input/output handling, argument counts, and the use of dashes to represent standard input.
- Application integration tests test the flow from the `run` method called from the `ApplicationRunner`.
- Test Coverage: Class - 85% (51/60), Method - 82% (184/222), Line - 74% (1029/1381)

How To Run All Test Cases

- Build the project using Maven and right click and run all tests on the root `test` folder. `test/tdd` folder contains the TDD tests for EF2, which are currently disabled.