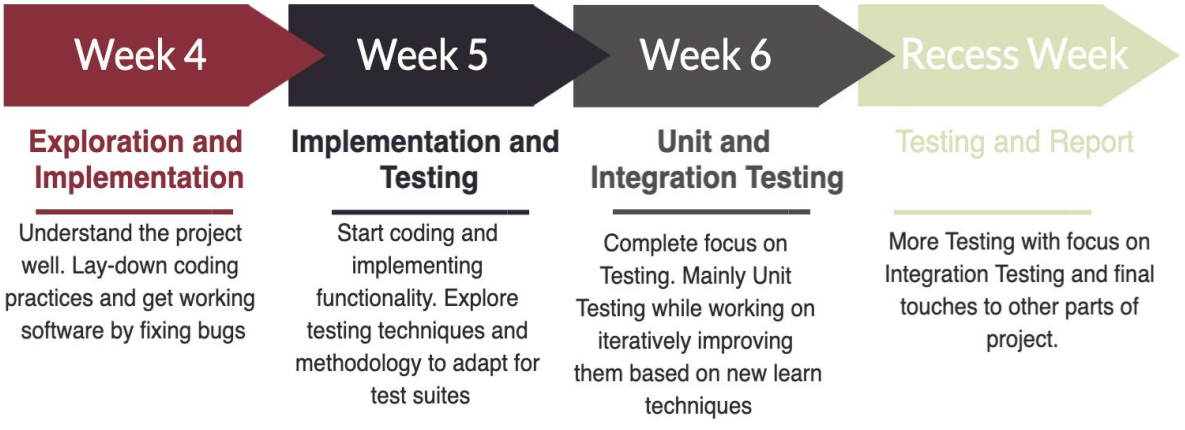
CS4218 - Milestone 1 Report

1. Implementation Plan

- Started by listing down what was there to be implemented so we could have a sense of how much work it would be: `rm` and `paste` for BF, and `cut` and `mv` for EF2.
- In terms of the shell functionality, we noticed that all of the required was already implemented even though there were several bugs.
- Our initial task before starting implementation was to fix the existing bugs in order to get a working software. For this we first tested manually, however switched to a more programmatic way in order to make sure we were not breaking other parts of the implementation and causing regression by fixing tests. These quick tests gave us confidence in our implementation.
- Each member worked on their own in their commands with the others reviewing the pull requests and occasionally discussing solutions. We had come up with certain coding quality standards to follow to lead to a cleaner and similar looking code.
- To clarify the specification, we referred to the man pages of the corresponding GNU coreutils, as well as the original source code. After having a good understanding of what is to be implemented, we created rough diagrams which depicted how a command would behave in different conditions and inputs. This not only helped us in implementation but gave us a good idea of how the test cases for each suite would be generated.
- Whenever it seemed like too much effort to replicate the original functionality, given that the focus in this module is on testing, we made our own adaptations, all of which are explained in the assumptions document.
- We also aimed to fix the implementation of commands which weren not working as expected. In particular, command `ls` was flawed and threw some wrong error messages. Substantial effort was put into fixing the design and making it as close to the GNU shell as possible, within the time constraint.
- In later stages of the project, we were iteratively improving upon implementation slightly based on certain faults and/or functional differences we found while generating tests and testing them against our test suite.



2. Testing Plan

- Overall we aimed to cover as a minimum, all the different argument configurations for each command, which ends up reflecting more or less as method coverage.
- For generating test cases, we started off by trying to fully understand the functionality and brainstorm different possibilities in terms of testing approaches. After this, we looked at the implementation to understand assumptions and structure of code, so we could polish our test cases.
- After writing the tests we tried to roughly apply techniques like MC/DC and branch coverage once we had looked at the code and started doing structural testing.
- The detailed types of test cases we generated can be found in the JavaDoc of each test class and a summary is also present in this report below. We focused more on unit tests as they are low cost and help find faults specifically.
- To choose our inputs we tried as much as possible to cover the basic corner cases, which in this specific domain ended up reflecting mostly as: no arguments, null arguments, single argument, insufficient arguments, multiple arguments, invalid flags, and nonexistent files.
- We made a specific effort to include plenty of negative test cases, where the applications should not succeed in executing but rather terminate with exceptions.
- Additionally, we attempted to add test cases for bugs found in the code provided. However in many of these cases, the bugs were very specific things and it made little sense to introduce a test case for them.
- Most of our tests are not fully unit tests as the modules are not perfectly isolated.This is because in testing the applications, we decided to interact mostly with the `run()` method in a black-box approach, meaning that some external methods such as the argument parser and other utility methods are used in those calls. However, we considered this to be appropriate as isolating everything perfectly would likely be more effort than one can justify, without significantly improving our debugging ability.
- Mocking was still employed once in testing the `CallCommand` class, allowing us to isolate the logic in the command itself from the actual applications being called by stubbing them and verifying that the correct calls were made to run the apps.
- Since most applications deal with and manipulate files, we decided to keep some test assets in a directory (`dummyTestFolder`) such that the application does not need to create and delete them all with each test run. Note that in certain cases, files are still generated and deleted on the fly.
- One big challenge that this approach presented was to keep such directories clean and not leave behind files that are not supposed to be left behind. We tried to deal with these such that the files are deleted on exit, but this has inconsistent behaviour if the JVM does not terminate normally so in most cases we are using the `@AfterAll` method to revert any changes, and in the remaining cases we delete the files at the end of the test method. This is something we would like to keep improving.
- We introduced a `TestUtils` class that we intend to expand, but for now provides just a method to assert that an exception's message contains a specific text, as we observed this was used in most negative test cases.
- In summary, we used tools like IntelliJ, PMD, static analysers like FindBugs, mockito for testing and used techniques like starting with black-box and then moving to white box and eventually testing integrations for creating an effective test suite.

2.1 Summary of Unit Tests

- We have summarised the test cases generation here. Our code includes detailed JavaDocs which will have more details of our test cases for each of the functionality.
- We followed the naming convention `test[Feature being tested]` as it worked well with PMD and made it readable.
- Our focus was on building a rigorous test suite for each functionality that would help in detecting bugs and exactly pointing out the fault in functionality as were developing simultaneously while testing.
- In order to make the unit tests as isolated as possible, the abstracted helper methods were also tested for commands instead of the run() function only as this isolated the specific functionality related to command.

Positive Scenarios	Negative Scenarios
- Flags passed individually and in combination if functionally allowed - Combinations of different functional ways a command can be executed. - For inputs we created a rough test catalog for each command type and tried testing them without much overhead. E.g. In `sed`, the replacement rule was tested with multiple replacements, regex and different string content like empty, same etc - Different branches and paths were tested for functionality that was not very convoluted giving us a good branch and path coverage - Some common test inputs (e.g. for files as arguments, whether absolute path and relative path worked, file names with spaces, number of files, use of globbing) were used for commands that would behave differently in different situations. Test Catalogs generated which were specific to GNU shell earlier helped standardize this. - In addition, specific functions related to the unit being tested were also added which tested these in-depth. - We combined different types of inputs for args which gave us different paths or different types of output, this prevented any explosion of test cases and provided good confidence of all scenarios tested.	- Null Arguments or streams into `run()` - Incorrect arguments passed into helper functions - Incorrect flags passed to test the parsing of commands - Exceptional scenarios for functionality where it would fail with the write exception message shown to user when these exceptions are caught. E.g. File not found, File permissions not allowed, Directory expected, etc - All branches and cases when exception would be thrown were tested as we realised this didn't cause any explosion and helped us rigorously test any regression for future implementation changes - Structural testing was done to generate more negative cases and this helped in ironing out the implementation details and

2.2 Integration Tests

- We began integration by creating tests for shell operators like Pipe, Substitution, and IO Redirection as it helped in testing their functionality in isolation and also testing the integration of the commands evaluation.

- We only started integration tests, once we had a good knowledge of all unit tests as it helped us in effectively understanding how they would interact and the possible positive and negative scenarios.
- We then tested some common integrations of two/three commands which depended on each other whether through input, output or in command substitution. This helped in giving confidence that not only are commands working properly on their own but also when integrated with other functionalities.
- Integration Testing was also done to ensure that commands worked well from start to end when going through argument building, command parsing etc and this gave us a good integration of end-to-end in isolation to particular functionality.

3. Future Plans

- We have a statement coverage for 85%. Around half of the untested statements can be covered through System Tests and more Integration Tests and some unused and unreachable code can be further removed.
- Because of the amount of file manipulation, a significant part of the uncovered code also relates to `IOException`s which are often replaced by customized ones. Since we are not simulating IO problems in our testing, these statements often end up without coverage.
- Integration Testing will be done with more focus given to functional combining of commands.
- As of now, paste and cut behave differently when reading from STDIN due to differing assumptions in the development process. We intend to standardize this.
- We want to test the helper functions for unit tests as well as they are also being indirectly tested in other commands unit tests and this will help in making the unit tests more isolated. Moreover, since we have the mocking framework up, we wish to add more isolated unit tests to improve the testing effectiveness in order to find underlying bugs that can be easily localised.
- We also aim to test the integrations of different commands and ensure software works correctly in real world situations when commands are entered one after another.
- We had some suggestions for improving the implementation structure in order for us to allow better testing of software. One example is to make `ApplicationRunner` a static class to avoid repeated instantiation.

4. Appendix

- When JUnit runs multiple test classes, the test methods from different classes are interleaved. Since we are using the Environment class to store the current directory as a static field, this causes several errors to occur, so we have just been running each test class separately to avoid it. For this, we have already set up a configuration `All Test Class Forking` in IntelliJ which can run the tests sequentially.
- Tests for EF1 functionalities like `cp` and `diff`, have been created, though they are expected to fail due to the missing implementations. We also found some bugs related to shell and EF1 functionalities and those tests have been marked as `@Ignored` for now but should pass when implementation is fixed.