Implementation Plans
For our implementation, we decided to adopt the iterative waterfall SDLC as the requirements for the project were well-defined and would be unlikely to change.

We split the implementation for milestone 1 as documented with deadlines to review the progress.

In accordance with what was discussed during the lab and in our team discussions, we adopted a bottom-up approach. Our team felt that we should complete the implementation of the applications before the shell implementations as we identified dependencies that the shell implementation would have on the completed applications. An example of this would be that in order to test the functionality of quoting, we would at least need the echo application to work. This approach would allow more options to be used with the shell implementation and produce a more robust project.

Project Details and Timeline:

	BF1		EF1		EF2	
	Shell	App	Shell	App	Shell	App
Member 1		exit, wc	Globbing	tee		rm
Member 2		ls, echo	Pipe	mv		paste
Member 3	Quoting	cd		cat	Substitution	cp
Member 4	I/O Redirection	grep		split	Semicolon	uniq

Milestone 1.1 (Deadline: 22/02 evening, 6pm)
Application
Implement Functionalities: BF EF1
Unit tests for basic and extended functionalities: BF, EF1 (and EF2)

Milestone 1.2 (Deadline: 25/02 evening, 6pm)
Shell
Implement Functionalities: BF EF1
Unit tests for basic and extended functionalities: BF, EF1 (and EF2)

Milestone 1.3 (Deadline: 28/02 afternoon, 12.30pm)
Test cases for all functionalities

Milestone 1.4 (Deadline 1/03 morning, 11.00 am)
Finalization of Report (max of 4 pages, 10pt font)
Finalization of Assumptions.pdf
Touch up codebase + code quality, pmd

Testing Plans

Firstly, for BF and EF1, as the implementation needs to be done before testing, we adopted a mixture of white-box and black-box testing. As part of black-box testing, we analysed the application and shell specification before coming up with the range of possible inputs. After which, we partitioned the input space into edge cases, positive, negative cases and tests which are likely to fail based on our intuition. We also included some test cases for random cases to ensure that the feature works as expected.

As part of white-box testing, we aimed to cover the crucial path, statement and method coverage, before expanding to cover condition coverage.

For the integration of our components, we ensured that for every shell feature, we integrated the shell feature with every other possible application and tested their interactions to ensure that it gives the expected behaviour.

For EF2, black-box testing was done based on the feature specification and the method specification indicated in the interface given. After with, we followed a similar approach as above where we partitioned the input space into test cases which are likely to behave similarly.

For all of our relevant test cases, we automated the creation and deletion of sample files which were used for testing to reduce human errors.

For our applications, we conducted pair-wise testing to generate test cases. An example is given for our grep testing:

Flag (0/1/multi/repeated flag)	**inputFormat**			
	stdin		**file**	
	System	**File**	**One file**	**Multi file**
No flag	F	F	T	F
-i	F	T	F	T
-i -c -H (multi + precedence)	F	T	T	F
-H	T	F	F	F
-i -H(multi)	T	F	T	F

Application of Testing Tools and Techniques

As showcased in our lab, we leverage on the Mockito library for spying and mocking to isolate the SUT such that any bugs found can solely be attributed to the SUT - the core essence of unit testing. We have included below a short example to demonstrate the stubbing process, due to limitation space limitation..

```java
@Override
public void changeToDirectory(String path) throws CdException {
    Environment.currentDirectory = getNormalizedAbsolutePath(path);
}
```

Although a simple method like the above which merely assigns a variable typically does not require testing on its own, we did so nevertheless to improve the coverage. In this method, getNormalizedAbsolutePath() is a unit dependency for the changeToDirectory SUT.

```java
@Test
void changeToDirectory_AnyValidPath_ChangesDirectory() throws CdException {
    when(cdApp.getNormalizedAbsolutePath(SRC_LITERAL)).thenReturn(SOURCE_DIRECTORY);
    cdApp.changeToDirectory(SRC_LITERAL);
    assertEquals(SOURCE_DIRECTORY, Environment.currentDirectory);
}
```

As seen above, to isolate the unit during unit testing, we mocked the behaviour of the getNormalizedAbsolutePath() method to return a pre-defined string (SOURCE_DIRECTORY) upon its calling. The Spy class was used to achieve so. This pre-defined string would then be assigned to Environment.currentDirectory in the SUT, which is verified in the assertEquals() of the test.

Furthermore, certain commands such as grep, cat, uniq require outputting the results to System.out during execution. Although we could simulate a human input via System.setIn function, it proved to be difficult to test the correctness of output written to System.out. Upon deliberate research, we found the perfect tool for it on the Maven repository - System-lambda - a collection of functions for testing code which uses java.lang.system.

```java
@Test
public void run_InvalidOptionFileProvided_OutputsErrorMessage() throws Exception {
    String[] args = new String[]{"-a", PATTERN, TEST_FILE1};
    String expectedOutput = PATTERN + NON_EXIST_ERR_MSG;
    String actualOutput = tapSystemOut(() -> {
        grepApp.run(args, System.in, System.out);
    });
    assertEquals(expectedOutput, actualOutput);
}
```

As seen above, when invalid command arguments are passed to the grep Application, the application would output to the shell the exception caught which we are saving into a variable actualOutput. This allows us to verify the contents of actual against expected output.

Libraries Used:

- JUnit
- Mockito (for stubbing)
- System-lambda (for testing of output)

Summary of test cases provided

In generating the test cases, we identified the format of each command and their input, output and expected behaviour

For each of our test cases, we documented 4 fields:

- Description
- Required Files
- Input
- Expected Behaviour

Each application functionality would have a set of:

- Positive test cases
- Negative test cases
- Positive integration test cases with (shell)
- Negative integration test cases with (shell)