Milestone 1:

Plans for implementation and testing:

- Implementation:
    - Each command application's class will implement its own interface, ensuring adherence to a standardized set of methods and behaviors. This promotes modularity and facilitates the interchangeability of command implementations.
    - Use the methods provided to override and implement the function, each command application class will override and implement the required functionality. This approach ensures consistency in method signatures and behavior across different command applications.
    - Error and exception handling mechanisms will be implemented to handle exceptional scenarios gracefully. This includes the creation of new exception classes when necessary to encapsulate specific error conditions and provide meaningful error messages to users.
    - An argument parser class will be added to the parser folder to parse command-line arguments and identify flag options provided by the user. This facilitates the extraction of relevant options and parameters required for command execution.
- Testing:
    - Unit Testing:
        - Covered different methods of each application class
        - Ensure individual components and methods perform as expected under various conditions
        - Assertions such as assertTrue, assertFalse, and assertEquals will be used to validate method outputs against expected results, ensuring correctness and reliability of the codebase
        - Edge cases such as empty file arrays and null inputs will be tested to verify that the application handles such scenarios gracefully without crashing or producing unexpected behavior
        - Test cases will be added to cover uncovered code segments, ensuring that all code paths are exercised during testing to uncover potential bugs or logic errors
        - Test cases will be designed to cover different conditional branches within the code, ensuring that each choice within conditional statements is exercised at least once during testing
    - Integration Testing:
        - Integration tests will check for the correct handling of different input combinations for each application class. This includes testing scenarios where multiple options or parameters are provided in different combinations to ensure consistent behavior across different usage scenarios

How test cases were generated:

1. Identification of Test Scenarios:
    a. Identified different scenarios based on the functionality of each application class

    b. Test scenarios consider various cases such as no options specified, different combinations of options, and boundary cases
2. Boundary cases considerations:
    a. Include cases with empty input stream, null input stream, and single or multiple lines of inputs, empty files, multiple files
3. Exploratory Testing:
    a. In addition to formal test cases, exploratory testing is conducted to uncover additional issues or scenarios not captured by predefined test cases. This allows for the discovery of hidden bugs, usability issues, and edge cases that may have been overlooked during formal testing
    b. For instance, during the initial testing of the **sort** command, we encountered an issue where negative numbers were not being sorted correctly. Recognizing this discrepancy, we proceeded to create specific test cases involving negative numbers to validate our observation. Subsequently, we promptly iterated on the **sortApplication** implementation to address and rectify this sorting error

Tools and techniques used for testing:

1. **JUnit Framework**: The JUnit framework is utilized for writing and executing unit tests, providing a standardized approach to test-driven development and automated testing.
2. **Assertion Libraries**: Different assertions from the JUnit library, such as assertTrue, assertFalse, and assertEquals, are employed for result validation, ensuring that expected outcomes match actual results during testing.
3. **Exception Verification**: The assertThrows method is used to verify that exceptions are thrown as expected for error conditions, ensuring that the application handles exceptional scenarios correctly.
4. **Input Simulation**: ByteArrayInputStream is utilized to simulate input streams for various test case scenarios, allowing for the testing of input-dependent functionality without relying on external resources.
5. **Code Quality Tools**: IntelliJ PMD plugin is used to ensure clean code and adherence to coding standards, helping maintain code readability, maintainability, and scalability.
6. **Mockito**: Mockito is used for stubbing and mocking dependencies, enabling isolation of components for unit testing and facilitating controlled behavior of dependencies during testing scenarios.

Faults found:

Code	Fault	Test cases that found faults
ApplicationRunner.runApp()	Missing break statements	ApplicationRunnerTest
StringUtils.isBlank()	Incomplete while loop and incorrect condition statement	StringUtilsTest.isBlank_Whitespace_ReturnsTrue
CdApplication.run()	Doesn’t handle empty/null parameters	CdApplicationTest.run_EmptyArgs_ThrowsCdException

		CdApplicationTest.run_NullStdin_ThrowsCdException CdApplicationTest.run_NullStdout_ThrowsCdException
SortApplication.sortInputString()	Not able to sort negative numbers correctly	SortApplicationTest.sortFromFiles_NegativeNumbersInFile_ReturnsSortedOutput
IORedirectionHandler.extractRedirOptions()	Shell exception not thrown when argument list is either null or empty	IORedirectionHandlerTest.parseCommand_nullCommand_ThrowShellException IORedirectionHandlerTest.parseCommand_emptyCommand_ThrowShellException
CommandBuilder.parseCommand()	Tokens used did not reset	CommandBuilderTest.parseCommand_twoCommandsWithSemicolon_ThrowNothing CommandBuilderTest.parseCommand_PipeCommand_ThrowNothing
PipeCommand.evaluate()	FileNotFound exception not handled	PipeCommandTest.evaluate_FirstCommandError_ThrowFileNotFoundException PipeCommandTest.evaluate_SecondCommandError_ThrowFileNotFoundException

The following test cases are expected to fail as they have yet to be implemented (EF1):

- PipeCommandTest
- GlobbingTest
- LsApplicationTest
- PasteApplicationTest
- UniqApplicationTest
- MvApplicationTest

Summary of test cases provided :

- Test cases cover various scenarios including individual component testing and integration testing, ensuring that the entire application behaves as expected under different usage scenarios.
- Test scenarios are derived from requirements and specifications, ensuring that all specified functionality is thoroughly tested.
- Coverage metrics such as statement coverage, branch coverage, and path coverage are used to ensure comprehensive testing and identify areas that require additional attention.

90% classes, 84% lines covered in package 'sg.edu.nus.comp.cs4218'

Element	Class, %	Method, %	Line, %
app	100% (0/0)	100% (0/0)	100% (0/0)
exception	73% (14/19)	63% (14/22)	63% (14/22)
impl	97% (40/41)	93% (156/167)	85% (1068/1...
Application	100% (0/0)	100% (0/0)	100% (0/0)
Command	100% (0/0)	100% (0/0)	100% (0/0)
Environment	100% (1/1)	100% (1/1)	100% (1/1)
Shell	100% (0/0)	100% (0/0)	100% (0/0)

100% classes, 85% lines covered in package 'sg.edu.nus.comp.cs4218.impl.app'

Element	Class, %	Method, %	Line, %
CatApplication	100% (1/1)	100% (4/4)	77% (45/58)
CdApplication	100% (1/1)	100% (3/3)	100% (22/22)
CutApplication	100% (1/1)	75% (3/4)	51% (40/78)
EchoApplication	100% (1/1)	100% (2/2)	100% (15/15)
ExitApplication	100% (1/1)	100% (2/2)	100% (3/3)
GrepApplication	100% (1/1)	100% (4/4)	97% (70/72)
LsApplication	100% (2/2)	91% (11/12)	90% (79/87)
MkdirApplication	100% (1/1)	100% (4/4)	92% (48/52)
MvApplication	100% (1/1)	0% (0/2)	33% (1/3)
PasteApplication	100% (1/1)	66% (2/3)	75% (3/4)
RmApplication	100% (1/1)	100% (4/4)	100% (39/39)
SortApplication	100% (2/2)	100% (7/7)	79% (73/92)
TeeApplication	100% (1/1)	100% (3/3)	87% (34/39)
UniqApplication	100% (1/1)	100% (2/2)	100% (3/3)
WcApplication	100% (1/1)	100% (5/5)	92% (100/108)

- Regression testing strategy is employed to ensure existing functionality is not affected by new changes.
- Exploratory testing is conducted to uncover additional issues or scenarios not captured by formal test cases, enhancing the overall quality and robustness of the software.