MILESTONE 1 REPORT

TEST PLAN

Scope

- *Components that will be tested (in-scope):*
    - All Parsers, Applications, and Shell Functionalities.
    - We only unit test the public methods of the modules.
- *Components that will not be tested (out-of-scope):*
    - All Exceptions classes as it only contains constructor and usually in practice, we only unit test public methods. We also do not test ErrorConstants class as it contains only static final fields. In addition, we do not test unused methods such as terminate. We do not remove unused methods, especially those from skeleton for example terminate method.

Approach

We identified three main levels in the module structure:

1) The top level is the ShellImpl module. (The ShellImpl module will be System Test in future milestones)
2) The middle level consists of the Commands and Utils modules such as ApplicationRunner, CommandBuilder CallCommand, PipeCommand and SequenceCommand.
3) The bottom level (leaf) is mainly the Applications such as CdApplication, RmApplication and SedApplication together with some of its smaller dependencies such as its exception classes, its respective Arguments classes as well as ArgsParser classes.

We mainly adopt a modified version of the bottom up approach for integration testing where we mainly unit tests the Application modules as well as its ArgsParser classes. We adopt this approach as by unit testing the bottom most layer followed by conducting integration testing on the higher level layers any bugs found in the integration testing is due to the higher level modules under test as its dependencies/the bottom layer are unit tested. This is to minimise the number of classes that require stubbing which are often done in practice.

We stated the approach is modified as it is not a fully bottom up approach as the project specifications requires the unit testing of the basic functionalities and extended functionalities which includes the Commands modules such as PipeCommand which we identified as the middle layer as mentioned earlier. Hence, modules in both the middle layer and the bottom layer we have identified, have some modules which we conducted unit testing as required by specification.

Furthermore, by definition of bottom up approach, only terminal modules are tested in isolation however, some application modifies we identified depends on their corresponding ArgsParser class hence, not all Application modules, by definition of terminal module, an actual terminal module. We resolved this by unit testing the respective ArgsParser modules as well as the Application modules. Basically, we adopted a hybrid of unit testing and bottom up integration testing approach.

For the unimplemented application modules in the skeleton code, we adopt test driven development approach, in other words, planning and writing the unit tests before implementation to ensure that the methods function as intended based on the test cases.

For some semi implemented application modules in the skeleton code, we write test cases according to behaviour of unix shell first and then run the test cases on our shell to check for wrong outputs and behaviours. With this test driven development method, we are able to more easily identify bugs without having to look through the code to look for bugs.

For example, for LsApplication, a semi implemented application, a bug in method listFolderContent is found. During input of { ls -d folder1} , according to shell in linux, it should return “folder1” instead of the unimplemented shell that returns directories in folder1. Using a test case of input {ls -d folder1}, we are able to pinpoint the bug easily for our implementation by running through the code in IntelliJ’s debugging mode.

The top most layer which we identified, the ShellImpl module, we identified it as being better suited for system test in the future milestone as it is where our shell's Java main method is located and it is the main entry point when executing the shell application.

Generating Test cases

We utilised techniques and tools from functional and structural testing to maximise coverage for individual components and integration of components.

Systematic Testing: Category-Partition Method

Here is an outline of our approach to do functional testing:

1) Decompose project specification into independently testable features as covered in the scope.
2) For each feature, identify all parameters and all possible environment element (if any)
    a) Label them as a category.
    b) Determine the various assumptions for each command.
3) For each category identified in 2), identify all possible representative values
    a) Identification can be done by using:
        i) Boundary value testing
        ii) Erroneous condition testing
4) Introduce possible error or single constraints
5) Generate test case specification(s)
6) Generate test case(s)

We will illustrate the whole process with an example given below using SortApplication.

Firstly, we identify all parameters.
Parameters:

- Flag arguments
- [FILES]

Next, we find the representative values of each category and introduce constraints as below.

Category	Representative Values	Constraints
[-nrf]	n	-
	r	-
	f	-
	nr	-
	nf	-
	rf	-
	nrf	-
	None	-
	Argument other than n, r or f	[ERROR]
[FILES]	Valid single file	-
	Multiple valid distinct files	-
	Multiple valid similar files	-

	Standard input from user	-
	At least 1 file has no read access	[ERROR]
	At least 1 file provided is a directory	[ERROR]
	Single file not found	[ERROR]
	At least 1 file not found	[ERROR]
	Empty standard input	[SINGLE]

After introducing constraints, the total number of test cases has reduced to 37 test cases. This is an improvement from 80 test cases without constraints.

Random Testing

In addition to functional and structural testing, we have also implemented random testing. This is done by comparing the execution of our commands using our shell alongside with UNIX shell to ensure that implementation made in our shell works similarly with implementation made with UNIX shell.

Structural Testing

For structural testing, we use IntelliJ-Idea code coverage runner tool to aid in generating the necessary metrics on our project. Based on the test case requirements for Milestone 1, our focus will be on statement coverage and branch coverage minimally.

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	86.2% (56/ 65)	88.5% (231/ 261)	86.4% (1534/ 1776)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
sg.edu.nus.comp.cs4218	100% (1/ 1)	50% (1/ 2)	50% (1/ 2)
sg.edu.nus.comp.cs4218.exception	77.3% (17/ 22)	64.3% (18/ 28)	64.3% (18/ 28)
sg.edu.nus.comp.cs4218.impl	0% (0/ 1)	0% (0/ 3)	0% (0/ 23)
sg.edu.nus.comp.cs4218.impl.app	94.1% (16/ 17)	97.2% (104/ 107)	90.3% (967/ 1071)
sg.edu.nus.comp.cs4218.impl.app.args	83.3% (5/ 6)	80% (28/ 35)	75.7% (140/ 185)
sg.edu.nus.comp.cs4218.impl.cmd	100% (3/ 3)	100% (9/ 9)	97.2% (69/ 71)
sg.edu.nus.comp.cs4218.impl.parser	100% (6/ 6)	100% (24/ 24)	100% (59/ 59)
sg.edu.nus.comp.cs4218.impl.util	88.9% (8/ 9)	88.7% (47/ 53)	83.1% (280/ 337)

Figure 1: Screenshot of our project coverage generated by IntelliJ Idea

With the combination of category partition testing, random testing and coverage metrics, this will ensure that our project is well tested and maintained with maximum code coverage.

Resources

We added Maven and Travis continuous integration support to manage build dependencies as well as automate unit testing, integration testing and regression testing.

Note: By Maven standards, unit testing uses **Test.java naming convention as detected by Maven sunfire plugin (Link) whereas integration testing uses **IT.java naming convention as detected by Maven failsafe plugin (Link). As such we follow the naming conventions, **Test.java for unit testing and **IT.java for integration testing.

The testing frameworks used in this project are listed below:

1) JUnit 5 (The main Java unit testing framework which acts as a driver for the module under test).
2) Mockito (A testing framework which supports mocking of objects which is especially useful for creating stubs and unit testing)

We also use GitHub tracker with labelling to keep track of issues such as bug reports, resolved bugs, and implementation progress status.

In addition, IntelliJ PMD Plugin is used to check for good coding practices but this is used as a guide as PMD is a static code analyser hence, it has limitations. For example, if there is a finally block with a custom method which closes the stream, PMD will still consider a violation as it only checks for .close() method but not the custom method which may contain the .close() method.

Moreover, IntelliJ build in coverage tools is used to act as a guide in our unit testing and integration testing to check for statement coverage as well as branch coverage (mainly branch coverage) as in lecture, we learnt that 100 percent statement coverage does not mean 100 percent branch coverage as some branch conditions may not be covered.

Schedule of Testing Activity

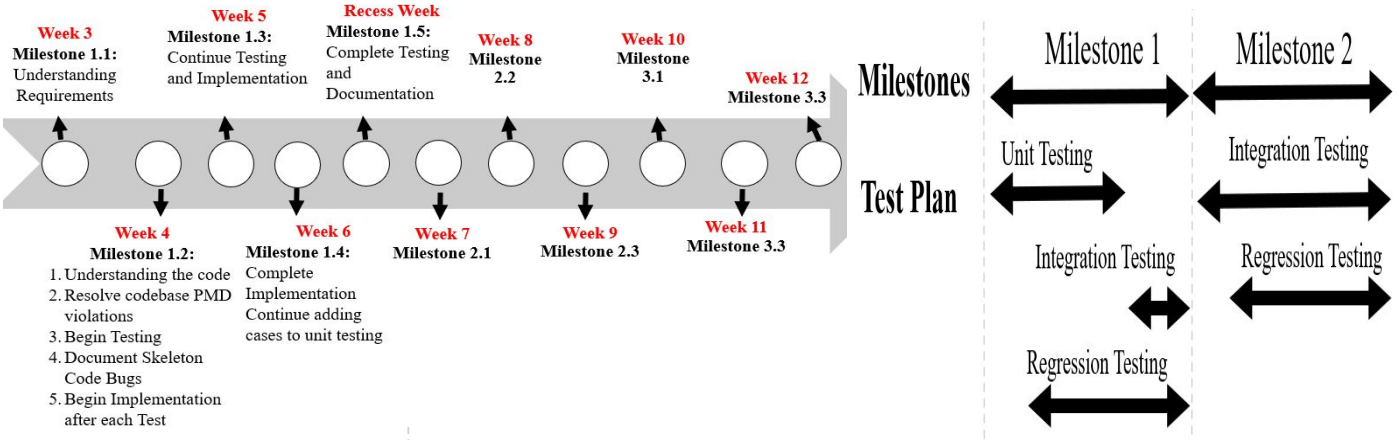

Figure 2: Timeline and Test Plan Illustration

Our team decided to use test-driven development (TDD). Our process is as follows:

1. We start by understanding the requirements.
2. We split up the tasks into Basic Functionality and Extended Functionality.
3. Next, we turn the requirements into test cases. This will ensure that our subsequent implementation is in line with the project specifications,
4. After coming up with the test cases, we create unit tests based on which applications or shell features we would like to implement for the week.
5. After the unit tests are up, we will then proceed to write integration tests.
6. With these tests, we were able to pinpoint the bugs that were in the skeleton code. We debugged, and resolved PMD issues. At this point, we also began documenting these bugs in this report.
7. Thereafter, we begin implementation. As we implemented the shell features and applications, we got more ideas for test cases and added them to our unit and integration tests. These tests also facilitated a continuous debugging process, which allowed us to detect bugs as we implemented the code. Regression testing is frequently done when any changes were made to the tests or the implementation. This is to ensure that any changes will not cause previous working functionalities to stop working.
8. Once testing and implementation has been completed, we focused on the other aspects of documentation.

Summary

In Milestone 1, we have implemented basic and extended functionality 2 of our shell. Furthermore, we have kickstarted our testing effort using techniques and tools taught in CS4218 and beyond.

In Milestone 2, we will be implementing the remaining functionality of our Shell that is under Extended Functionality 1 and continue our testing effort across the project to achieve maximum code coverage.

Appendix

Bug Report for Skeleton code

Bug Report Number	Type	Status	Class Name	Comments
1		Resolved	CdApplication.java	Missing check/throw exception in run method when args is empty to match skeleton assumption of not supporting cd with no arguments. Resolved: By adding if condition check of args.length and throwing ERR_MISSING_ARG if args.length is 0.
2		Resolved	CdApplication.java	Missing check/throw exception in run method when there are more than one args to match skeleton assumption of cd application must take in one arg. Resolved: By adding if condition check of args.length and throwing ERR_TOO_MANY_ARGS if args.length is > 1.
3		Resolved	PipeCommand.java	The if condition logic should be if (i == callCommands.size() - 1) and if (i != callCommands.size() - 1) instead of original skeleton if (i == callCommands.size()) and if (i != callCommands.size()) respectively.
4	PMD	Resolved	SequenceCommand .java	Replace throw new ShellException(e.getMessage()); in skeleton to throw (ShellException) new ShellException(e.getMessage()).initCause(e);
5		Resolved	SedApplication.java	Does not accept standard input. Resolved: Use IO.getLinesFromInputStream to read from stdin and subsequently replace the lines.
6		Resolved	SedApplication.java	Missing method that performs the replacement of string. Does not print any output given a valid command and a valid file. Resolved: Implement replacementHandler, getFileContents and replaceString methods in SedApplication. Skeleton code's output string is empty as nothing is appended into the string.
7	PMD	Resolved	Environment.java	Renamed class from Environment.java to EnvironmentHelper.java. Refactor all references to the original Environment.java and its methods to EnvironmentHelper.java and its methods.
8	PMD	Resolved	EchoApplication.java , GrepApplication.java , LsApplication.java, SequenceCommand .java	Resolved PreserveStackTrace PMD error by using the format for example for ShellException: throw (ShellException) new ShellException(e.getMessage()).initCause(e); Similar format for other exception with ShellException replaced with the appropriate exception such as GrepException, LsException appropriately and the initCause argument passed is the appropriate exception caught in the catch block of try catch statement.

9	PMD	Resolved	ApplicationRunner.java	Added missing break statement in the case APP_SED: application = new SedApplication(); break; Similarly add break statement in APP_EXIT, APP_GREP cases respectively.
10		Resolved	EchoApplication.java	Added missing String += STRING_NEWLINE; after the result = String.join(“ “, args); statement in the else statement of constructResult method. As original skeleton code the output printed by echo when with arguments do not have new line character at the end.
11		Resolved	ExitApplication.java	Replaced System.exit(0) in terminateExecution() method with throw new ExitException(“0”); to take advantage of the ExitException.java available which is not used in the skeleton code. The ShellImpl will then catch the ExitException and get the exit code and check if it is 0. If it is zero, ShellImpl will call System.exit(0) to terminate the shell.
12		Resolved	CallCommand.java	Replace if logic check for if (parsedArgsList.isEmpty()) to if (!parsedArgsList.isEmpty())
13		Resolved	CommandBuilder.java	Added missing tokens = new LinkedList<>(); statement after cmdsForSequence.add(new CallCommand(tokens, appRunner, argumentResolver)); statement.
14		Resolved	IORedirectionHandler.java	Replaced if (argsList == null && argsList.isEmpty()) with if (argsList == null || argsList.isEmpty())
15		Resolved	StringUtils.java	Replaced for (int i = 0; i < str.length();) with for (int i = 0; i < str.length(); i++) due to missing i++ in skeleton code which cause an infinite loop.
16		Resolved	StringUtils.java	Replaced if (Character.isWhitespace(str.charAt(i))) with if (!Character.isWhitespace(str.charAt(i)))
17		Resolved	IORedirectionHandler.java	Replaced return str.equals(String.valueOf(CHAR_REDIR_INPUT)); in isRedirOperator() method with return str.equals(String.valueOf(CHAR_REDIR_INPUT)) || str.equals(String.valueOf(CHAR_REDIR_OUTPUT)); As the original skeleton code does not detect > as a redirection operator
18		Resolved	LsApplication.java	Replaced if(directory.charAt(0) == '/') with if(file.isAbsolute()) for determining absolute path as it is not consistent across different platforms. Windows uses backslash (\) for file/path separator while unix based use forward slash (/)
19		Resolved	LsApplication.java	ListFolderContent Algorithm implementation changes. Ls folder should list the folder content if its in current folder and if it’s in current directory.
20		Resolved	ArgumentResolver.java	1. The skeleton code handles only the command but does not handle any sub command. I have added all subOutputSegment into the regex argument linked list

				2. Line 92 - 93: Remove from parsedArgsSegment instead of subOutputSegment and add another check to see if parsedArgsSegment is empty.
21		Reserved for debugging in milestone 2	GrepApplication	Flags in arguments are not checked thoroughly to ensure only the correct character is provided. For example “-ii”, “-ic” would still work.
22		Resolved	IORedirectionHandler.java	Missing handling of double “>>” direction, to throw ERR_SYNTAX error.
23		Resolved	CommandBuilder.java	Line 79: case CHAR_REDIR_INPUT: Break; Missing tokens.add(String.valueOf(firstChar)); causing IO redirect to not detect “<”.
24	PMD	Resolved	IORedirectionHandler.java	Resolved broken null check pmd violation by replacing skeleton code if (argsList == null && argsList.isEmpty()) with if (argsList == null || argsList.isEmpty()). Basically, the skeleton code uses && logical operator in the if condition instead of || logical operator.
25	PMD	Resolved	RegexArgument.java	Resolved AvoidFieldNameMatchingMethodName method by refactoring the method declaration name public boolean isRegex() to public boolean hasRegex()
26		Resolved	ShellImpl	Resolved issue where shell does not keep prompting for new user input after a command is executed as required by project specification. This is resolved by adding a do while loop with the String currentDirectory = EnvironmentHelper.currentDirectory; System.out.print(currentDirectory + “> “); commandString = reader.readLine() method execution together with shell.parseAndEvaluate method execution in the body of the do while loop so as to keep prompting user for input.

27	PMD	Resolved	GrepApplication.java	Resolved excessive method length PMD violation for GrepApplication.grepResultsFromFiles() method by using refactoring through extract method in build in IntelliJ refactoring tools from skeleton’s:
				count = 0;
				while ((line = reader.readLine()) != null) {
				Matcher matcher = compiledPattern.matcher(line);
				if (matcher.find()) { // match
				if (isSingleFile) {
				lineResults.add(line);
				} else {
				lineResults.add(f + ": " + line);
				}
				count++;
				}
				}
				Extracted to a newly created separate method known as addMatchedPatternToLineResult. The full method declaration is private int

				addMatchedPatternToLineResult(boolean isSingleFile, BufferedReader reader, Pattern compiledPattern, String fileName, StringJoiner lineResults) throws IOException On the other hand the original count = 0 GrepApplication.grepResultsFromFiles() method is replaced with count = addMatchedPatternToLineResult(isSingleFile, reader, compiledPattern, f, lineResults);
28		Resolved	ShellImpl.java	In the try catch block of commandString = reader.readLine() where catching IOException, the skeleton code has a break statement in the catch block which causes compiler error. It is resolved by removing the break statement and in the catch block and replacing it with printing the exception message to console.
29		Resolved	CallCommand.java	Resolved closing stream bug where stream not closed lead to commands like paste A.txt B.txt > AB.txt; rm AB.txt to throw IOException as streams are not closed in the above example stream using AB.txt is not closed hence file cannot be removed. This is resolved by appending IOUtils.closeInputStream(inputStream); IOUtils.closeOutputStream(outputStream); to the last line of the evaluate method in CallCommand.java. This does not resolve the pmd violation of CloseResource for callcommand.java due to limitation of pmd as explained in the justification above.
30		Resolved	ShellImpl.java	Resolved variable commandString might not have been initialised error by replacing String commandString; with String commandString = “”; in other words, initialising it to an empty string.

31		Resolved	IOUtils.java	Resolved java.io.FileNotFoundException as the exception is not catch in the openOutputStream() method. This is resolved by enclosing fileOutputStream = new FileOutputStream(new File(resolvedFileName)); in a try catch block catching FileNotFoundException or IOException in general such as by:
				
				try {
				fileOutputStream = new FileOutputStream(new File(resolvedFileName));
				} catch (IOException e) {
				throw new ShellException(ERR_FILE_NOT_FOUND);
				}

32		Resolved	ShellImpl.java	Missing printing of currentDirectory variable as part of shell prompting similar to in unix. This is resolved by adding the two lines shown below: String currentDirectory = EnvironmentHelper.currentDirectory; System.out.print(currentDirectory + "> ");
33	PMD	Resolved	ShellImpl.main(), SedApplication.replaceSubStringInFiles(), WcApplicaiton.countFromFiles()	Resolved pmd closeresource violation by wrapping stream in try with resources so that stream will be closed.
34		Resolved	ArgumentResolver.java (in relation to the CallCommand.java class)	First argument of the command provided by the user cannot be preceded by whitespace, or tab. To resolve this, I added trim() to the first argument to remove all the whitespaces. This is similar to Linux's behaviour.
35		Resolved	SortArguments.java	SortArguments should throw an exception for illegal flag options instead of ignoring it. I resolved this by replacing the block of code that skips invalid flags with throw new Exception() to show illegal flag messages.
36		Resolved	FindApplication.java	Resolved not catching no arguments input.

Justification for unresolved PMD violation

PMD Violation that are unresolved	Justification
Excessive Method Length in ApplicationRunner.runApp()	With more command Applications being implemented and added to the shell, new case statements in the runApp() method's switch statement is required to allow supporting of the newly implemented command Applications. It is an unavoidable pmd error and to maintain code consistency in using the switch as the way to determine which application to run, we decided to not resolve it.
CloseResource pmd violation in CallCommand.evaluate()	This is actually resolved by appending IOUtils.closeInputStream(inputStream); IOUtils.closeOutputStream(outputStream); to the last line of CallCommand.evaluate() method however, this is not detected by pmd as pmd is a static code analyser and does not check that the streams are closed using an external method in this case IOUtils.closeInputStream and IOUtils.closeOutputStream methods.
CloseResource pmd violation in IOredirectionHandler	The streams need to stay open to pass into the applications so cannot be closed in this case.
CloseResource pmd violation in IOredirectionIT	The resource is actually closed in writer.close() however, PMD are not able to detect the closing of resource.
CloseResource pmd violation in PipeCommand.evaluate()	This was actually resolved by appending IOUtils.closeOutputStream(nextOutputStream); after each

	iteration of the for loop. However, similar to in CallCommand, the limitation of pmd in not being able to detect if resources are closed by external method lead to pmd violation.
PasteApplication.mergeFile, PasteApplication.mergeFileAndStdin, Paste.Application.paste()	For BufferedReaders, as an array of BufferedReaders is used, a for loop is used to close each individual BufferedReader. PMD however, requires us to use try catch to implicitly close the Reader. This form of closing, is restricted to try (BufferedReader br = new BufferedReader(new FileReader(file)) which is not suitable for this case as an array of buffered readers is used. Also, similarly, FileReader has been explicitly closed. PMD again does not detect this, and thus should be unresolved as it cannot be resolved.
AvoidDuplicateLiterals pmd violation	As clarified with Ms Wang Zi, CS4218 module tutor, we do not have to resolve AvoidDuplicateLiterals pmd violation.
GodClass, Excessive Method Length in LsApplication	Due to the specifications of the shell command, PMD flagged a god class stating the class is doing too many things, however, it is the entire class by itself so unable to resolve the violation.
LongVariable pmd violation for test cases	As clarified with Dr Cristina, module coordinator, we do not have to resolve LongVariable pmd violation for test cases.