

MILESTONE 2 REPORT

Test driven development (TDD) process

We excluded pmd checks for test sources as clarified with the lecturer due to the test cases provided in Milestone 2 contributing to many pmd violations. This was done by going to **IntelliJ Settings > PMD > Options > enable Skip Test sources > Apply**. This is to allow us to focus on code quality for our implementation.

Since our group has completed some form of test driven development during our Milestone 1, where some features which are semi-implemented such as Ls and Find, we extended the test driven development to other implemented features such as CutApplication after receiving the test cases. Our plan was to use the test cases, cross check with our test cases and remove those overlap, and also change the test cases according to the assumptions made by the group. We would also check for the unit tests to ensure that the methods function as intended based on the test cases.

As the group developed EF2, our implementation for EF1 was not completed. Therefore, using the test cases provided and our own test cases that we have written before, by merging the overlapped test cases, we then implemented the features such as globbing using the test cases as reference.

For the test suites provided by other groups, some test cases are different from our assumptions, therefore, we would exclude them from our test cases using Junit 5's `@Disabled` annotation with reason.

For example, in LsApplicationTest, our implementation does not allow the use of -R and -d in a single command, which follows the shell implementation, however, for the test cases provided from other teams allow such commands, therefore, we disabled them as it performed differently from our assumptions.

Overall, the test suites given to us allowed us to use the test case results to implement the feature of EF1. For example, for cp function, we integrated our own cp test and test given to us and it helped us form up our assumptions and implementation. Furthermore, for EF2 cases, the test cases given to use helped us correct some of our assumptions and also improve implementations. The experience with the test cases are generally positive despite differences and they are generally helpful in our test driven development.

Integration testing (Planning and execution)

Planning

We identified three main levels in the module structure:

Top Level (Root) ShellImpl module (Contains the main method hence, we identify this module as module which be undergo System Test)
Middle Level (Main focus for Integration Testing) Consists of Commands and Utils modules ApplicationRunner, CommandBuilder, CallCommand, PipeCommand, SequenceCommand and modules that involve Command Substitution, Globbing, IORedirection and Quoting)
Bottom Level (Leaf) Consist mainly of Application, Arguments and ArgsParser modules such as RmApplication and CdApplication modules together with their corresponding exception and/or Arguments and ArgsParser dependencies.

In Milestone 1, based on the project specification, the focus was on unit testing of Command modules such as CallCommand and PipeCommand as well as Application modules such as RmApplication and their dependencies. In Milestone 2, we adopted a modified version of the bottom up approach where the bottom most layer (leaf) has been fully unit tested and hence, any bugs found in the higher level layers during integration testing is due to the higher level modules under test. This is to minimise the number of stubs required which is often done in practice.

We stated that the approach is modified as it is not a fully bottom up approach due to the project specifications in Milestone 1 where some of the modules in the middle levels such as the command modules identified in the structure above (CallCommand and PipeCommand) have been unit tested. In other words, both the middle layer and the bottom

layer in which we have identified have some modules which we have conducted unit testing as required by Milestone 1 definition.

Furthermore, by definition of bottom up approach, only terminal modules are tested in isolation however, some application modules we identified depend on their corresponding ArgsParser class hence, not all Application modules, by definition of terminal module, are in theory actual terminal modules. 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.

As shown above in the 3 level module structure we identified, the main focus for integration testing would be the modules we identified in the middle layer such as the CallCommand and PipeCommand modules. We also conduct integration testing for modules in the bottom most level who have other dependencies such as RmApplication module's run method which depends on RmArgsParser module. ShellImpl module is identified mainly for System testing as it contains the main method in which the Java Application starts program execution.

As there are many possible combinations of components interactions to test in integration testing and testing all possibilities is not feasible due to potential combinatorial explosion. We decide to adopt the following strategies listed in the table to maximise the effectiveness of the integration testing while avoiding combinatorial explosion.

We identified two main categories in which the shell application modules such as RmApplication and CutApplication can be categorised:

- 1) Application modules involving input streams and/or output streams (Involves IO streams)
- 2) Application modules not involving input streams and output streams (No IO streams involved)

Application Modules Categorisation	
Involves IO streams	No IO streams involved
EchoApplication, PasteApplication, SedApplication, DiffApplication, GrepApplication, WcApplication, CutApplication, LsApplication, SortApplication, FindApplication	RmApplication, ExitApplication, CdApplication, CpApplication, MvApplication

From the above Application Categorisation and characteristics of each middle layer components, we are able to identify the Application modules whose interactions should be the main focus when testing the components in the middle layer such as PipeCommand and SequenceCommand. We have summarised the main focus of integration testing for each components in the middle layer in the table below:

Middle Layer Integration Test Plan	
Middle Layer Component	Main focus of Integration Testing
PipeCommand	Application modules involving IO streams
SequenceCommand	All Application modules (Those that involves IO streams and those that do not involve IO streams)
Command Substitution	Application modules involving IO streams
Globbering	Application modules that accept files or paths as arguments such as LsApplication
Quoting	Application modules involving IO streams using Quotes
CallCommand	All Application modules (Those that involves IO streams and those that do not involve IO streams)
IO Redirection	Application modules involving IO streams with Input < and

In general, middle layer components where IO streams play a significant role such as the PipeCommand module where the pipe operator binds the output of the left part to the input of the right part. The fact that the PipeCommand's main usage generally involves input streams and output streams, Application modules involving IO streams such as LsApplication and GrepApplication should be the main focus of the integration testing. This is due to the fact that PipeCommand interactions between Application modules that do not involve IO streams do not take advantage of the binding of streams by PipeCommand hence, it is not significant and not the main focus of the interactions to be tested during integration testing of the PipeCommand.

However, this does not mean that we do not test Application modules not involving IO streams in PipeCommand. We generally follow the general guidelines below:

- At least 1 negative scenario such as when exceptional case/corner case/error handling)
- At least 10 test cases for positive test cases (for example in PipeCommand with 1 pipe)
- At least 4 test cases for more complex positive test cases (for example in PipeCommand with at least two pipes)
- If Application modules involving IO streams is the main focus, at least 1 test case with Application modules involving no IO streams.

The execution phase

In the execution phase, we focus on meeting the project specifications/ requirements. As such, running all the test cases in the test suite could bog down the system unnecessarily, and slow down the delivery process. This is especially important when there are a substantially large number of test cases in the test suite, and when system resources available are limited (unable to complete running all the test cases within a short period of time).

To reduce the number of test cases being run, we could run tests from top-down. In other words, we start off by running the system tests. This will highlight which requirements in acceptance testing are not met. Next, we could then run the relevant integration tests that are related to the errors pointed out in the system tests. This would narrow down the number of test cases that are running, particularly when the team is focused on debugging. Should there be any error highlighted in the integration tests, we could then run the individual unit tests pertaining to the units that are involved in that particular integration test that is showing error. These unit tests will then help us to pinpoint the exact place where the code has error(s). Running the entire test suite for every regression testing is also not recommended as it is highly inefficient to run test cases that cover classes where changes are not introduced. For regression testing, we would be running the cases from the bottom up. This means that we will run the unit tests for the units that have been recently changed. This is to ensure that the units are running correctly without any errors, and to allow us to spot the error faster. Thereafter, we need to check if other units, that have dependencies on the changed unit, are working the same way. System testing may not be necessary for regression testing, if the unit test and integration test do not show any deviations.

Tools used to aid our testing

With Maven as our build automation for the project, this introduces us with more tools available to analyse and test our project. Here are the list of some essential testing tools used for our project:

Jacoco

J4Shell

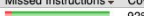
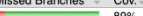














Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
sg.edu.nus.comp.cs4218.impl.app		92%		89%	101	634	111	1,546	3	130	1	17
sg.edu.nus.comp.cs4218.impl.util		86%		81%	30	165	37	379	3	53	1	9
sg.edu.nus.comp.cs4218.exception		69%		n/a	9	28	18	56	9	28	4	22
sg.edu.nus.comp.cs4218.impl		17%		0%	4	6	21	25	1	3	0	1
sg.edu.nus.comp.cs4218.impl.parser		95%		91%	2	36	0	71	0	24	0	6
sg.edu.nus.comp.cs4218.impl.cmd		95%		100%	2	26	4	80	2	12	0	3
sg.edu.nus.comp.cs4218.impl.app.args		99%		93%	12	137	1	250	0	42	0	7
sg.edu.nus.comp.cs4218		100%		n/a	0	1	0	1	0	1	0	1
Total	961 of 11,211	91%	163 of 1,462	88%	160	1,033	192	2,408	18	293	6	66

Figure 1. Our overall Jacoco Code Coverage Report from Maven Site Plugin

Jacoco is a free code coverage tool to provide coverage analysis on our project, mainly line and branch coverage. Coverage analysis can help us identify possible cases that are not covered in our implementation. With the integration of Sunfire report and Maven Site plugin, we are able to provide an auto-generated website with a graphical code coverage

analysis report of our project. This report will show the coverage percentage and also can provide an in-depth code reference for any line or conditions that could be possibly not covered. See above for an overall view of our Jacoco report generated from **Maven Site Plugin**. A more accurate version of Jacoco coverage is in the Appendix from IntelliJ built in Jacoco coverage from Windows.

Note: The coverage report in some applications such as RmApplication is lower due to the minimum check added for file permissions. In our Assumption report we have indicated that we assume file permissions are correct due to issues with file permissions in different operating systems and file systems. There are checks added in implementation and test cases to check the operating system it is being run on or uses operating system specific methods, hence, certain statements and branches would never be executed depending on the operating system hence, affecting code coverage.

Parallel Isolation Tests (PIT)

PIT is an automated mutation testing tool. Mutation testing is when faults or mutations are automatically seeded into your code, then PIT runs our unit tests against automatically seeded versions of your application code. When the application code changes, it should produce different results and cause the unit tests to fail. If a unit test does not fail in this situation, it may indicate an issue with the test suite.

The purpose of having mutation testing is to increase confidence and quality of our test suites by being able to detect faults in our implementation using our test suites. Traditional test coverage (i.e line, statement, branch, etc.) such as Jacoco measures only which code is executed by your tests. It does not check that your tests are actually able to detect faults in the executed code. It is therefore only able to identify code that is definitely not tested. PIT report is auto-generated with Surefire Report plugin and is viewable on the website. See below for an overall view of our PIT report generated from **Maven Site Plugin**.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
40	85% 	81% 

Breakdown by Package

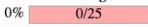
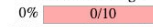
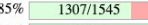
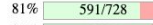
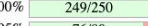
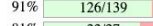
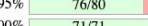
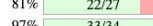
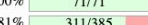
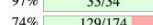
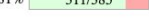
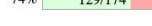
Name	Number of Classes	Line Coverage	Mutation Coverage
sg.edu.nus.comp.cs4218.impl	1	0% 	0% 
sg.edu.nus.comp.cs4218.impl.app	15	85% 	81% 
sg.edu.nus.comp.cs4218.impl.app.args	7	100% 	91% 
sg.edu.nus.comp.cs4218.impl.cmd	3	95% 	81% 
sg.edu.nus.comp.cs4218.impl.parser	6	100% 	97% 
sg.edu.nus.comp.cs4218.impl.util	8	81% 	74% 

Figure 2. Our overall PIT Test Coverage Report from Maven Site Plugin

However, using PIT comes with various drawbacks. One of the main drawbacks is the performance time. There is a huge difference between running our test suites in approximately 1 minute and running PIT with all our test suites in approximately more than 10 minutes. Despite being provided with many tweaks to improve the performance time, the performance time in running PIT is between 3 to 5 mins. This is still ridiculously slow. Furthermore, some tweaks such as limiting the number of mutations and/or scope of test classes provide lesser quality of feedback which might not be beneficial in increasing confidence in our test suites. As such, lesser quality feedback is another main drawback.

PIT is useful to a limited extent when creating a new shell application along with a new test suite. This is to strengthen the new application test suite and uncover behaviours that went untested during the writing of test cases phase. However, in our case, PIT will only serve to increase confidence in all our test suites as we have an ideal amount of code coverage before milestone 2 starts. In addition, we have to wait for a long time to finish executing PIT. As such, PIT does not provide substantial benefit to our cause.

Summary

In Milestone 2, we have implemented all basic and enhanced functionality of our shell while ensuring that all test cases created from milestone 1 passes and maintaining high code coverage.

Prior to Milestone 3, our project will be sent to multiple teams for system testing. Then, we refute bugs from other teams' bug reports on our program.

In Milestone 3, we will be working towards fixing bugs from the rebuttal and doing the closing of this project.

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	<ol style="list-style-type: none"> 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	IORedirectionHandle r.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	IORedirectionHandle r.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: <pre> 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++; } } </pre> 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(), WcApplication.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.
ExcessiveMethodLength in MvApplication	The mv application is rather complicated and it hard to implement without long method.
Avoid if (x != y) ..; else ..; in DiffApplication, and in paste	The diff and paste application has many conditions to consider. I arranged the logic based on what is necessary and anticipated to occur next. Negation rule will make it difficult in a situation where multiple conditions are being considered at the same time.
BufferedReader and File reader close PMD issue	These are explicitly closed (similar to paste), However, PMD does not detect. See above (in paste) for more explanation on this.
Excessive method length in paste	The method name is to make it clearer so that the person using the method, will know what it really does.

IntelliJ Built in Jacoco coverage (More accurate compared to Maven generated version)

Justification: This is more accurate compared to the Maven generated version due to issues running maven in Windows. This coverage is from Run All Tests with Jacoco coverage as specified in lab.

Coverage: All in cs4218-project-ay1920-s2-2020-team22 with cove... ×				
90% classes, 92% lines covered in package 'sg.edu.nus.comp.cs4218'				
Element	Class, %	Method, %	Line, %	Branch, %
app	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
exception	81% (18/22)	67% (19/28)	67% (38/56)	100% (0/0)
impl	95% (41/43)	96% (238/246)	92% (2179/2350)	88% (1291/1462)
EnvironmentHelper	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)

Coverage: All in cs4218-project-ay1920-s2-2020-team22 with cove... x

95% classes, 92% lines covered in package 'sg.edu.nus.comp.cs4218.impl'

Element	Class, %	Method, %	Line, %	Branch, %
app	95% (23/24)	98% (154/157)	93% (1686/1796)	89% (1072/1197)
cmd	100% (3/3)	83% (10/12)	95% (76/80)	100% (28/28)
parser	100% (6/6)	100% (24/24)	100% (71/71)	91% (22/24)
util	88% (8/9)	96% (49/51)	90% (342/378)	81% (169/207)
ShellImpl	100% (1/1)	50% (1/2)	16% (4/25)	0% (0/6)

Coverage: All in cs4218-project-ay1920-s2-2020-team22 with cove... x

88% classes, 90% lines covered in package 'sg.edu.nus.comp.cs4218.impl.util'

Element	Class, %	Method, %	Line, %	Branch, %
ApplicationRunner	100% (1/1)	100% (1/1)	100% (35/35)	100% (16/16)
ArgumentResolver	100% (1/1)	100% (10/10)	100% (85/85)	97% (47/48)
CommandBuilder	100% (1/1)	100% (2/2)	85% (42/49)	84% (21/25)
ErrorConstants	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
IORedirectionHandler	100% (1/1)	100% (6/6)	100% (49/49)	89% (25/28)
IOUtils	100% (1/1)	100% (6/6)	89% (33/37)	85% (12/14)
MyPair	100% (1/1)	100% (6/6)	100% (19/19)	80% (8/10)
RegexArgument	100% (1/1)	85% (12/14)	70% (57/81)	52% (26/50)
StringUtils	100% (1/1)	100% (6/6)	95% (22/23)	87% (14/16)

Coverage: All in cs4218-project-ay1920-s2-2020-team22 with cove... x

95% classes, 93% lines covered in package 'sg.edu.nus.comp.cs4218.impl.app'

Element	Class, %	Method, %	Line, %	Branch, %
args	100% (7/7)	100% (42/42)	99% (249/250)	93% (178/190)
CdApplication	100% (1/1)	100% (3/3)	96% (24/25)	95% (19/20)
CpApplication	100% (1/1)	100% (6/6)	88% (59/67)	87% (42/48)
CutApplication	100% (1/1)	100% (8/8)	99% (155/156)	93% (116/124)
DiffApplication	100% (1/1)	100% (17/17)	90% (228/253)	83% (141/168)
EchoApplication	100% (1/1)	100% (2/2)	100% (16/16)	100% (6/6)
ExitApplication	100% (1/1)	50% (1/2)	50% (2/4)	100% (0/0)
FindApplication	100% (1/1)	100% (12/12)	96% (122/126)	88% (62/70)
GrepApplication	100% (1/1)	100% (9/9)	94% (132/140)	90% (86/95)
LsApplication	50% (1/2)	88% (15/17)	85% (159/187)	79% (87/110)
MvApplication	100% (1/1)	100% (7/7)	91% (93/102)	89% (43/48)
PasteApplication	100% (1/1)	100% (6/6)	95% (116/121)	86% (80/92)
RmApplication	100% (1/1)	100% (9/9)	93% (84/90)	82% (53/64)
SedApplication	100% (1/1)	100% (6/6)	94% (81/86)	97% (47/48)
SortApplication	100% (2/2)	100% (7/7)	95% (67/70)	97% (45/46)
WcApplication	100% (1/1)	100% (4/4)	96% (99/103)	98% (67/68)

Coverage: All in cs4218-project-ay1920-s2-2020-team22 with cove... ×

100% classes, 99% lines covered in package 'sg.edu.nus.comp.cs4218.impl.app.args'

Element	Class, %	Method, %	Line, %	Branch, %
DiffArguments	100% (1/1)	100% (7/7)	100% (47/47)	85% (41/48)
FindArguments	100% (1/1)	100% (4/4)	100% (24/24)	93% (15/16)
GrepArguments	100% (1/1)	100% (7/7)	97% (41/42)	95% (23/24)
LsArguments	100% (1/1)	100% (5/5)	100% (23/23)	93% (15/16)
SedArguments	100% (1/1)	100% (7/7)	100% (42/42)	95% (23/24)
SortArguments	100% (1/1)	100% (6/6)	100% (34/34)	100% (32/32)
WcArguments	100% (1/1)	100% (6/6)	100% (38/38)	96% (29/30)

Coverage: All in cs4218-project-ay1920-s2-2020-team22 with cove... ×

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

Element	Class, %	Method, %	Line, %	Branch, %
ArgsParser	100% (1/1)	100% (3/3)	100% (18/18)	100% (10/10)
CutArgsParser	100% (1/1)	100% (6/6)	100% (28/28)	100% (10/10)
GrepArgsParser	100% (1/1)	100% (4/4)	100% (6/6)	50% (2/4)
LsArgsParser	100% (1/1)	100% (4/4)	100% (7/7)	100% (0/0)
MvArgsParser	100% (1/1)	100% (3/3)	100% (5/5)	100% (0/0)
RmArgsParser	100% (1/1)	100% (4/4)	100% (7/7)	100% (0/0)