

CS4218 - Milestone 1 Report

1. Implementation Plan

1. In order to implement and test a shell application with the different features, it was crucial for every member to have a good understanding of linux commands and how they work before we created tests and their implementations. Hence we had each member understand the specifications by going through the given project description and consuming online resources on linux commands.
2. The team then spends time to understand the existing code base, figuring out how the code is structured and the rough scope of implementation.
3. Define implementation guidelines and testing guidelines
4. Delegate work among all members and use a git flow to contribute to the project with active communication in the telegram group on progress and if there are any dependencies between units of work.

To begin coding the implementation, we realised there were a few crucial bugs with the shell that needed to be fixed first, so we prioritised the working prototype of the shell application before moving on to the more specific applications like mkdir. By working on the shell and its related logic first, it allowed us to understand how the entire shell application works as a whole.

For any bug fixes on original code, we also documented a table of what kind of bugs we found, how we found it, and how it was resolved.

After implementing the shell functionalities, we moved on to implement all the applications functionality of BF and EF2. For already implemented components, we would develop the tests for these components first, to figure out any discrepancies as according to specification, then based on the failing test cases, we would reverse engineer and figure out what was the broken/missing logic in the code, then implement a solution to fix it.

2. Testing Plan

JUnit 5 was used as our testing framework.

For testing, we split the testing into two phases, unit tests and integration tests. First we worked on creating unit tests for all the components, then tried to integrate two or more units in the integration tests.

Each @Test Method is also named in a structured format: methodName_inputValues_ShouldDoXXX.

2.1 Unit Tests:

The following units were tested:

- All applications in BF, EF1 and EF2
- All shell functionality and shell related applications (e.g. CallCommandTest)
- All utility tools (e.g. IOUtils)

For unit tests, we aimed to cover as many different combinations as possible for each unit, some important criteria we abided by is:

- Aim for 100% Method and Line Coverage
- Boundary cases testing
- Edge cases testing
- MC/DC approach to try different combinations
- Positive and Negative Cases (e.g. null input, incorrect argument etc).

To ensure each unit test is actually tested as a unit by itself, we avoided the use of integrating any other units during the test, and also for any non fully isolated unit, we made use of mockito in order to mock/stub any external unit dependencies.

For the unit tests setup, we also made sure to use JUnit features such as @BeforeEach, @AfterEach to better set up our test cases, reduce duplication and increase readability. For test cases that required reading in files, we could generate these files during the setup process and tear them down once the tests are done.

2.2 Integration Tests:

For integration tests, we used the approach of Pairwise Combinations, where for each application, we would test against all the other valid applications that make sense to test together.

During this process of integrating 2 applications, this often also includes using the Shell and other Shell Functionality such as Pipeline.

This means for the integration tests, we were able to not only test 2 units, but also made use of the actual shell application for the testing, which is really important since it constituted an actual workflow a user would use.

Structural Testing

Another important guideline was line and method coverage. In all tests, we aimed for close to 100% coverage as a good way to know all the logic in the declared classes and methods were at least covered in a path.

Test Class	Method Coverage	Line Coverage
StringUtils	100%	100%
CommandBuilder	100%	86%
IOUtils	100%	96%
RegexArgument	100%	100%
CallCommand	100%	100%
CatApplication	100%	95%
CdApplication	100%	78%
EchoApplication	100%	75%
MkdirApplication	100%	87%
CutApplication	100%	86%
ArgumentResolver	100%	84%
IORedirectionHandler	100%	92%
SequenceCommand	100%	78%
SortApplication	100%	87%
RmApplication	100%	72%
TeeApplication	100%	91%
GrepApplication	100%	69%

PMD

- In addition to our defined guidelines during the implementation and testing plan, we also made use of PMD extensively to verify any logic bugs, bad coding behaviours etc.

- We also used a table to indicate what PMD bugs we fixed and what PMD warnings were ignored with a good justification (e.g. PMD not being able to understand why a stream was not closed, even though the stream was mocked with Mockito).

How to run the tests

We are using JUnit, the tests can be found in the /public_tests folder, you can run each test.

Note: Certain unit tests and integration tests that involve testing a unit that is not yet implemented in EF1 (since our group is doing only BF and EF2) may fail.

Additional Dependencies

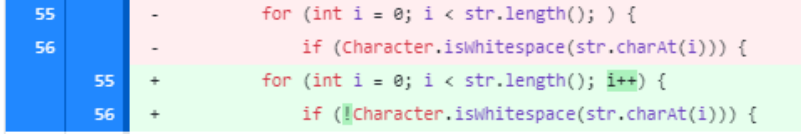
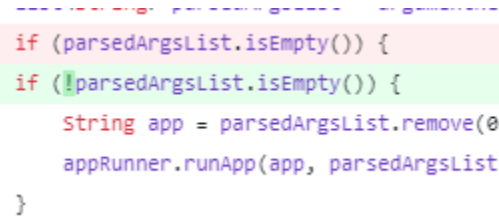
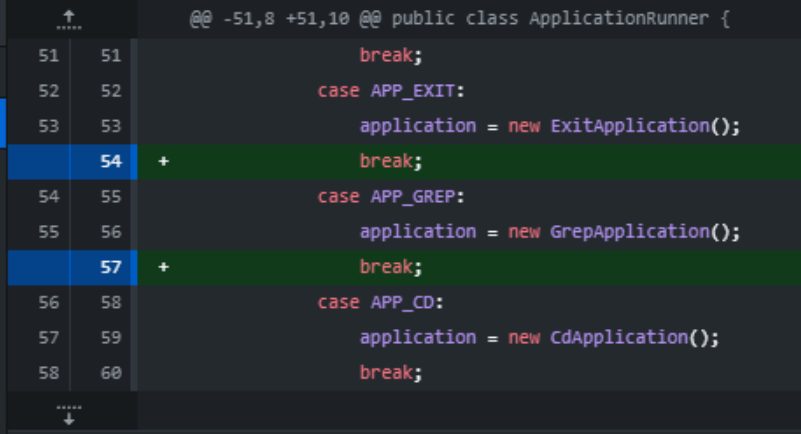
```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>3.0.0</version>
  <scope>test</scope>
</dependency>
```

- Added mockito for mocking classes during JUnit tests.

More information about bug reports and PMD can be found in Appendix below.

Appendix (Bug fixes and PMD Justifications)

Bug Fixes

Related File/Code	Remarks
StringUtils.java isBlank(String str) method	 <p>Missing i++ in for loop and wrong logic for checking white spaces.</p> <p>Discovered this bug while trying to run the shell then typing in a command, but nothing happened. (stuck in a loop)</p>
CallCommand.java evaluate method	 <p>Wrong logic, we should run the app when there are parsed arguments.</p> <p>Discovered bug when trying to debug why nothing happens in shell after running a command.</p>
ApplicationRunner.java runApp method	<p>Discovered "MissingBreakInSwitch" error while running PMD.</p> <p>Fixed error by adding missing breaks.</p> 

RegexArgument.java	<p>Discovered PMD warning where name of attribute is same as one of the methods (isRegex).</p> <p>Renamed isRegex() method to checkRegex() for fix.</p>
ShellImpl.java Main method	<p>Discovered PMD warning of CloseResource indicating my InputStream was not being closed.</p> <p>Added finally block to close the reader at the end of the method.</p>
ShellImpl.java Main method	<p>When running ShellImpl, the app would close after each input. (This is not the correct behaviour since we want to keep prompting the user for new inputs until we receive an empty input).</p> <p>Updated code to handle unlimited input attempts until an exception is thrown or input is empty or user keys exit as input (the ExitException is thrown).</p> <p>Also exits the system with an status code (0) indicating success and 1 indicating some error in execution status.</p>
RegexArgument.java globFiles	<p>GlobFiles method was missing a way to deal with absolute paths.</p> <p>Added check for absolute paths to resolve this problem.</p>
CommandBuilder.java parseCommand method	<pre>switch (firstChar) { case CHAR_REDIR_INPUT: tokens.add(String.valueOf(firstChar));</pre> <p>There was a missing logic for when the first character is '<', hence the fix was to make sure to add it to our token before the break.</p>
CommandBuilder.java parseCommand method	<pre>102 103 // add CallCommand as part of a SequenceC 103 104 cmdsForSequence.add(new CallCommand(tokens 105 + tokens = new LinkedList<>(); 104 106 } else { 105 107 // add CallCommand as part of ongoing Pipe 106 108 callCmdsForPipe.add(new CallCommand(tokens 109 + tokens = new LinkedList<>(); 107 110</pre>

	<pre> 110 cmdsForSequence.add(new PipeCommand(callCmdsForPipe)); 111 callCmdsForPipe = new LinkedList<>(); 112 } 113 + tokens = new LinkedList(); 114 break; </pre> <p>Missing logic to reset the tokens after processing it.</p>
EchoApplication.java constructResult method	No new line after echo command at the end, fix was to add a new line after constructing the results for the echo command.
IORedirectionHandler.java extractRedirOptions() method	<pre> @@ -35,7 +35,7 @@ public IORedirectionHandler(List<String> argsList, InputStream origInputStream, } public void extractRedirOptions() throws AbstractApplicationException, ShellException, FileNotFoundException { if (argsList == null argsList.isEmpty()) { if (argsList == null argsList.isEmpty()) { throw new ShellException(ERR_SYNTAX); } } </pre>
IORedirectionHandler.java isRedirOperator(str) method	<pre> private boolean isRedirOperator(String str) { return str.equals(String.valueOf(Char.REDIR_INPUT)); return str.equals(String.valueOf(Char.REDIR_INPUT)) str.equals(String.valueOf(Char.REDIR_OUTPUT)); } </pre>
IORedirectionHandler.java extractRedirOptions() method	<pre> public class IORedirectionHandler { private final List<String> argsList; private final ArgumentResolver argumentResolver; private final InputStream origInputStream; private InputStream origInputStream; private final OutputStream origOutputStream; private List<String> noRedirArgsList; private InputStream inputStream; @@ -73,6 +73,7 @@ public void extractRedirOptions() throws AbstractApplicationException, ShellExce throw new ShellException(ERR_MULTIPLE_STREAMS); } inputStream = IOUtils.openInputStream(file); origInputStream = inputStream; } else if (arg.equals(String.valueOf(Char.REDIR_OUTPUT))) { </pre>

PMD Justifications

PMD Bugs Warnings	Justification
CompareObjectsWithEquals IOUtils.closeInputStream() And IOUtils.closeOutputStream()	PMD is suggest we use equals() method to compare the inputStream and System.in instead of using == sign, but in our case, since we're comparing reference and not the actual object itself, so we will ignore this warning.
AvoidDuplicateLiterals pmd violation	There is no getting around this violation since we have to create multiple tests, for example if I were to call echo in different tests, this violation will show up.

CloseResouce PMD Violations in IOUtilsTest	<p>The streams was mocked using mockito, and using mockito we already verified that mockedStream was closed.</p> <p>PMD is not able to detect and understand the mocked streams and hence this violation cannot be avoided.</p>
CloseResouce PMD Violations in CallCommand.evaluate() And PipeCommand.evaluate()	<p>It says the streams are not closed. It is already closed in the logic by calling IOUtils.closeInputStream and closeOutputStream. We think PMD is not able to detect this.</p>
LongVariable warning	<p>PMD is warning us for long variables, but these variables are not extremely long. It is longer because we want to better portray to the coders what they are used for.</p> <p>Hence this warning is ignored.</p>
PreserveStackTrace	<p>Stack traces are not preserved since exceptions are thrown.</p> <p>We are following the implementation of the existing exceptions, hence this warning is ignored.</p>