

1. How much source and test code have you written? Test code (LOC) vs. Source code (LOC).

Source	Test (Manual + Automated)	Total
4324	68555	72879

Figure 1: LOC distribution (Numbers in the tables do not include original skeleton code, PMD)

The amount of test code LOC is significantly greater than that of our Source Code LOC indicating the amount of effort required in testing in order to ensure the quality of the project. Although the test LOC has been significantly increased by the automatically generated tests; with Randoop alone generating 37,000 lines, that would still result in an estimated 20000+ test LOC, which is 5 times as much as the Source LOC. However, we note that LOC alone is not indicative of the quality of the test cases written and our efforts to write them; these will be explained in further detail below.

2. Give an overview of the testing plans (i.e., timeline), methods and activities you have conducted during the project. What was the most useful method or activity that you employed?

Milestone 1

Milestone 1.1 (Deadline: 22nd Feb, 6pm)

- Application
- Implement Functionalities: BF EF1
- Unit tests for basic and extended functionalities: BF, EF1 (and EF2)

Milestone 1.2 (Deadline: 25th Feb, 6pm)

- Shell
- Implement Functionalities: BF EF1
- Unit tests for basic and extended functionalities: BF, EF1 (and EF2)

Milestone 1.3 (Deadline: 28th Feb, 12.30pm)

- Test cases for all functionalities

During the generation of test cases for milestone 1, our team employed combinatorial testing methods taught in the module. Notably, we used pairwise testing to generate test cases and contain the combinatorial explosion from the different inputs and outputs of each application and catalog testing to identify boundaries in the inputs.

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

Figure 2: Pairwise testing table for Grep Application

Milestone 2

Milestone 2.1 (Deadline: 15th Mar, 6pm)

- TDD of EF2 Applications and shell (Command substitution, Semicolon operator, Uniq, Rm, Paste, Cp)

Milestone 2.2 (Deadline: 19th Mar, 6.30pm)

- Integration and System tests

Milestone 2.3 (Deadline: 20th Mar, 2.30pm)

- EF1, (Evosuite and Randoop) EF2, (Square Test) Automation frameworks implementation

During this milestone, our team conducted TDD with the test cases provided by the teaching team. Additionally, we generated additional test cases for our project using automation tools such as Mockito, Evosuite, DiffBlue Cover, Squaretest and Randoop.

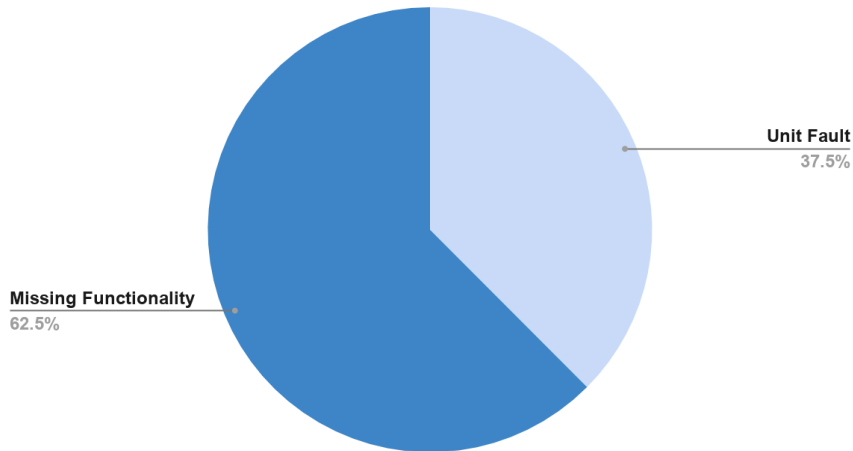
Throughout the entire project, we also conducted regression testing to ensure that there were no additional bugs introduced during our fixes.

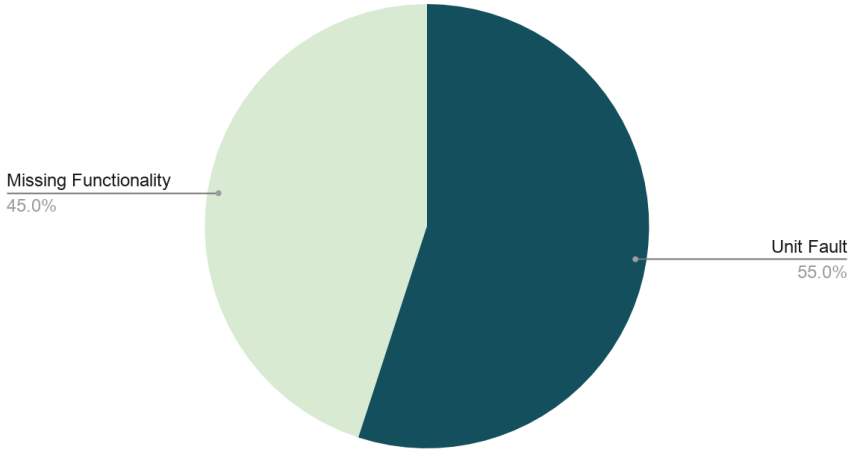
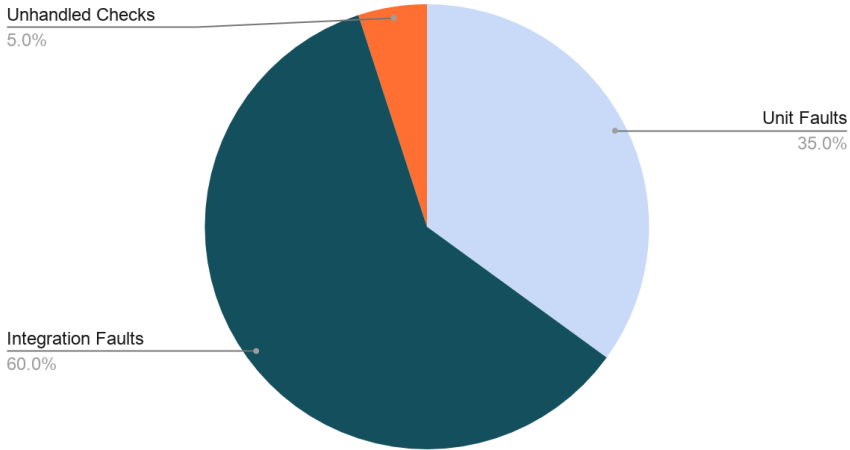
TDD was the most useful method that we employed during the project. To create the test cases for EF2 functionality in Milestone 1, we had to have a clear understanding on the requirements of each functionality. During milestone 2, since the requirements were well understood, implementation of the functionalities came easier as compared to milestone 1. Additionally, the test cases provided by the teaching team made it easier to identify and reduce faults in our project.

3. Estimate and analyse the distribution of fault types versus project activities.

- Use diagrams and/or explain the distribution of faults over project activities.
- Discuss what activities discovered the most faults. Discuss whether the distribution of fault types matches your expectations.
- Mention if you tracked your bugs or your explanations are based on estimates.
- Analyse the causes of bugs found in your project during Hackathon.
- Is it true that faults tend to accumulate in a few modules? Explain your answer.
- Is it true that some classes of faults predominate? Which ones?

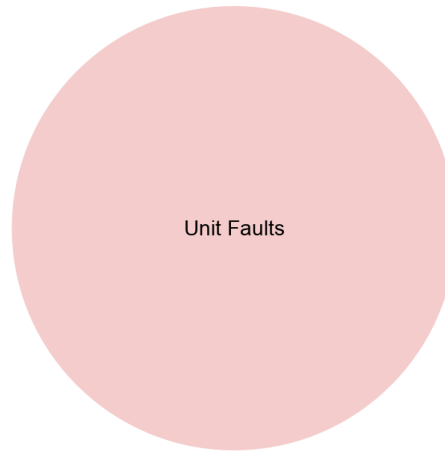
During the course of our project different fault types were identified during different project activities. As we do not have an official documentation tracking our bugs aside from our git commits, we will be discussing this section with estimations. The table below details the main types of faults detected for each activity of our project.

Activities	Main Types of Faults detected						
Requirements review	<p>Requirements Review</p>  <table border="1"><thead><tr><th>Fault Type</th><th>Percentage</th></tr></thead><tbody><tr><td>Missing Functionality</td><td>62.5%</td></tr><tr><td>Unit Fault</td><td>37.5%</td></tr></tbody></table> <p>Missing functionality faults were often detected during requirement reviews. This was especially so during the start of each milestone as our implementation would not be complete due to us not yet fully understanding how the commands should behave for each given input and scenario.</p> <p>Unit faults were also often detected during this activity. This was especially prevalent in Milestone 1, as the teaching team shared clear examples in the Q&A document regarding how the wc command was expected to behave for when the input files are "-".</p>	Fault Type	Percentage	Missing Functionality	62.5%	Unit Fault	37.5%
Fault Type	Percentage						
Missing Functionality	62.5%						
Unit Fault	37.5%						

Unit testing	<p>Unit Testing</p>  <table border="1"> <thead> <tr> <th>Category</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>Unit Fault</td> <td>55.0%</td> </tr> <tr> <td>Missing Functionality</td> <td>45.0%</td> </tr> </tbody> </table>	Category	Percentage	Unit Fault	55.0%	Missing Functionality	45.0%		
Category	Percentage								
Unit Fault	55.0%								
Missing Functionality	45.0%								
Integration testing	<p>Integration testing</p>  <table border="1"> <thead> <tr> <th>Category</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>Integration Faults</td> <td>60.0%</td> </tr> <tr> <td>Unit Faults</td> <td>35.0%</td> </tr> <tr> <td>Unhandled Checks</td> <td>5.0%</td> </tr> </tbody> </table> <p>Integration testing proved to be useful in identifying unit faults. In milestone 2, this was especially helpful in the case of shell specifications. For each application, we wrote integration tests involving different and multiple shell specifications. These included both positive and negative integration test cases.</p> <p>Integration testing also helped us to detect integration faults that we failed to see at the unit testing level. These bugs included issues such as input and output stream handling.</p>	Category	Percentage	Integration Faults	60.0%	Unit Faults	35.0%	Unhandled Checks	5.0%
Category	Percentage								
Integration Faults	60.0%								
Unit Faults	35.0%								
Unhandled Checks	5.0%								

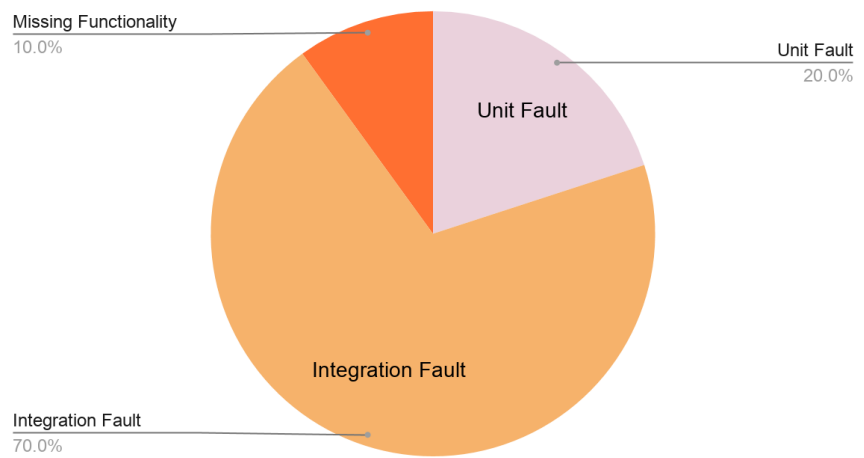
Coverage analysis

Coverage Analysis



Regression testing

Regression Testing



System testing	<p>System testing</p> <p>A pie chart titled 'System testing' showing the distribution of fault types. The chart is divided into two segments: a larger dark red segment representing 'Integration Faults' at 60.0%, and a smaller light red segment representing 'Unit Faults' at 40.0%. Labels with leader lines point to each segment.</p> <table border="1"><thead><tr><th>Fault Type</th><th>Percentage</th></tr></thead><tbody><tr><td>Integration Faults</td><td>60.0%</td></tr><tr><td>Unit Faults</td><td>40.0%</td></tr></tbody></table>	Fault Type	Percentage	Integration Faults	60.0%	Unit Faults	40.0%
Fault Type	Percentage						
Integration Faults	60.0%						
Unit Faults	40.0%						
Automated testing	<p>Automated Testing</p> <p>A pie chart titled 'Automated Testing' showing the distribution of fault types. The chart is a single solid yellow circle representing 'Defensive Faults' at 100%. The label 'Defensive Faults' is centered within the circle.</p> <table border="1"><thead><tr><th>Fault Type</th><th>Percentage</th></tr></thead><tbody><tr><td>Defensive Faults</td><td>100%</td></tr></tbody></table> <p>Our team utilized multiple varying automated testing tools to generate test cases. These testing tools include Randoop, Evosuite, SquareTest, DiffBlue Cover. These test cases proved to be most useful and relevant in the catching of defensive bugs (e.g. checking for null arguments passed in within methods calls).</p>	Fault Type	Percentage	Defensive Faults	100%		
Fault Type	Percentage						
Defensive Faults	100%						

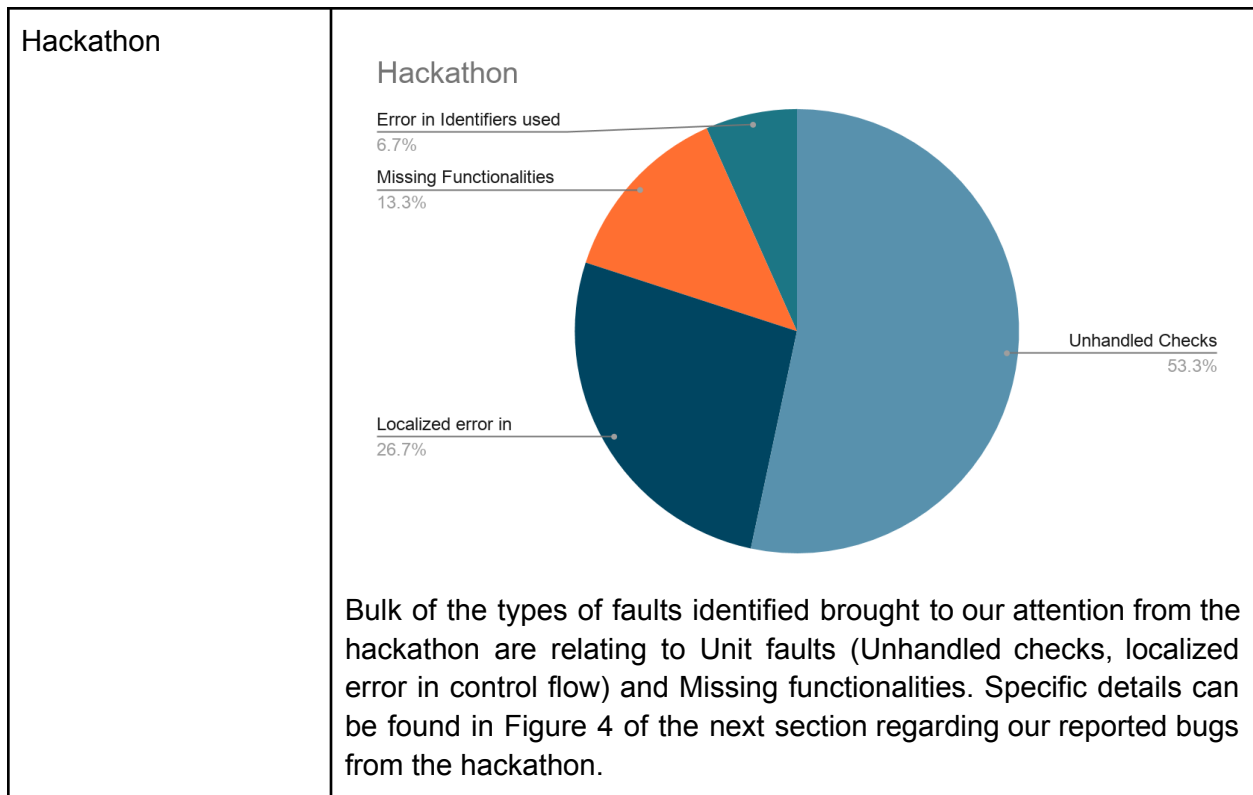


Figure 3: Project Activities With The Fault Types They Detected

Unit testing is the activity that discovered the most faults. This is because we divided the implementation of the applications amongst ourselves, with each class being implemented by a single member as per specification. Hence, we made it a point to vigorously test the respective components we were in charge of on the unit level, as we believed that detecting bugs at that level was crucial to prevent introducing complications at higher levels of testing such as integration and system testing. Additionally, unit testing with the TDD test cases also helped to identify many faults during our implementation as the edge cases were present in the test case. For this reason, we uncovered many unit faults, both for our command applications and shell functionalities such as globbing and command substitution.

To add on to our testing efforts, we also booted up virtual machines running on Linux to get hold of the shell. This enabled us to compare our outputs with that of Unix to verify our app's behaviour. Admittedly, through this acceptance testing, we have uncovered several bugs, some relating to the priority of error messages to display in the event of multiple flaws in a command.

Overall Distribution of Bug Faults

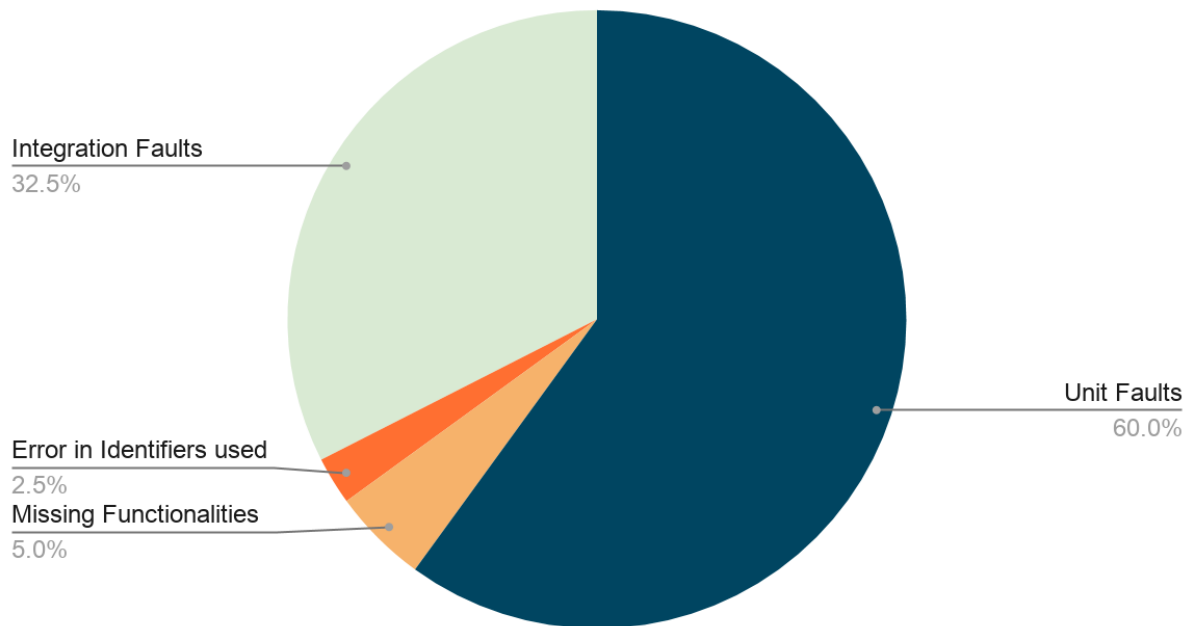


Figure 4: Overall distribution of bug faults detected

The faults discovered from each activity generally matched our expectations for the nature of the activity conducted and our efforts in these activities.

For instance, given the nature of requirements review activity, it was well within our expectation that most of the bugs we would discover would be related to the missing functionality and unit faults. Similarly, since integration testing involves the use of different commands together with shell functionalities, we had expected the faults identified to be those of integration faults and unhandled checks within modules as perhaps some modules did not consider certain inputs, especially those resulting from the execution of another command, this was indeed the case we observed as seen in Figure 3.

However, some activities did not entirely match our expectations as well. The proportion of fault types for the hackathon activity was one such outlier that did not entirely meet our expectations. We did expect unit faults for edge cases we missed, and this expectation was met, however, this was not true for our expectation on integration tests. As mentioned previously, as our team divided the implementation of applications among ourselves, more emphasis was placed in unit testing as opposed to integration testing to ensure that our individual components work as intended. As such, we were expecting integration related faults to dwarf our unit related ones. However, this was not the case. In fact, most faults that were brought to our attention were unit related ones and relatively few integration faults were reported.

Below is a table detailing the valid reported bugs we identified from the hackathon and their details, in total we have 14 unique bugs.

S/N	Bug Description	Module	Classes of fault
Team17_6	Split output file not at current working directory	SplitApplication	Error in identifiers used to create outputstream
Team17_9	Ls - error when no directory or absolute path provided in windows (Duplicate bug as 19_33)	LsApplication	Unhandled checks (did not consider absolute path for Windows)
Team17_12	Mv - unable to give relevant error messages when arguments are invalid	MvApplication	Unhandled Checks (resulting in wrong error message)
Team19_2	Pattern with just white spaces, following unix behavior, will match lines with white spaces rather than printing error message	GrepApplication	Unhandled checks (for empty pattern)
Team19_4	Uniq command with output directory, does not follow unix behavior, not printing "uniq: outputDir: Is a directory"	UniqApplication	Unhandled Checks (resulting in wrong error message)
Team19_6	Removing non-empty directory with both "-r" and "-d" options, following unix behavior, should remove entire directory rather than printing error message	RmApplication	Localized error in control flow
Team19_20	The command echo `echo one two three four wc cat` contains extra spaces compared to unix shell output. Unix shell output: "1 4 20" Application output: " 1 4 20"	Command Substitution	Missing functionality
Team19_21	Is `ls`	LsApplication	Localized error in control flow
Team19_22	The command cp `echo` `echo README.md` `echo README1.md`	Command Substitution	Unhandled checks (for empty string)

	throws an unexpected exception (Duplicate bug as 19_23 to 19_26)		as argument)
Team19_33	ls -R displays error messages instead of listing all files and folders from cwd recursively.	LsApplication	Unhandled checks
Team19_34	Executing "ls -X file.txt file.abc" does not sort the results according to its extension. When running "ls -X file.txt file.abc" in Unix shell, the output prints 'file.abc' before 'file.txt', but in the Java shell, the results is based on the order in the argument provided.	LsApplication	Unhandled checks
Team19_36	When wc command is used on directories, only display directory name in error message, without total count for directory (zeros)	WcApplication	Missing functionality
Team19_42	"cp file1 file1 dir1" does not print a warning message. When running "cp file1 file1 dir1" on Unix shell, warning message indicating that file1 is specified more than once is printed to console. However, on the Java shell, no such message is printed when running the same command.	CpApplication	Unhandled Checks (of the case with same source files)
Team19_46	"grep wo split -l 1" is creating an extra empty file from split.	SplitApplication	Localized error in control flow
Team19_50	When the command echo "echo 'a b'" is entered, a b is returned when expected output is a b.	Quoting	Localized error in control flow

Figure 5: Details and Breakdown of all Reported Bugs from Hackathon

We do not agree to the claim that faults tend to accumulate in a few modules. This is because even though in general, the LsApplication module has the most bugs, it is not by a significant margin over the other modules. Moreover, every module with bugs has relatively few bugs, which can be seen in Figure 6 below.

S/N	Module	Bug Frequency
1	LsApplication	3
2	SplitApplication	2
3	Command Substitution	2
4	GrepApplication	1
5	UniqApplication	1
6	RmApplication	1
7	MvApplication	1
8	WcApplication	1
9	CpApplication	1
10	Quoting	1

Figure 6: Program Modules With Their Respective Bug Frequency

Amidst the reported bugs from the hackathon, our team observed that some classes of bugs tend to predominate. Most of our bugs tend to fall into either the category of unhandled checks error or localized error in control flow, which can be seen in Figure 7.

We believe that both of these bug types were the most common because we may have overlooked some of the edge cases regarding the possible input values and combinations during our implementation. This is because both of these bug types are a result of a lack of or mistake in the use of conditional branching. These bugs were present not only in the applications we wrote during milestone 1 but also those in milestone 2, where TDD was conducted. This suggests that the edge cases that we overlooked were those that other teams also failed to notice due to its obscurity.

What is worth mentioning, however, is how relatively easy the bugs are to fix once it has been brought to our attention. This applies especially so for the hackathon, since our reviewing teams would give us examples of the unique combinations and input values that trigger these bugs. For instance, it was brought to our attention that the command “rm -r -d nonEmptyDir” was not behaving as expected. Our RmApplication implementation was such that it would check for the -d flag first followed by -r and for each input file, if the -d flag is true, that file would be subjected to the relevant operations within that condition and then continue to the next file, skipping the -r check. As such, the command “rm -d -r nonEmptyDir” would behave similarly to “rm -d nonEmptyDir” instead. This bug was easily fixed by swapping the order of the checks to -r first followed by -d. The bug fix can be seen in pull request [#46](#) of our repository.

S/N	Bug Types	Bug Frequency
1	Unhandled Checks	8
2	Localized error in control flow	4
3	Missing functionality	2
4	Error in identifiers used to create output stream	1
5	Major Errors	0

Figure 7: Bug Types With Their Respective Frequency

4. Provide estimates on the time that you spent on different activities (percentage of total project time):

The following table illustrates the following proportion of time that we spent on various activities:

Activity	Time spent (%)
Requirements analysis and documentation	10
Coding	30
Test development	35
Test execution	20
Others (Fixes for hackathon)	5

Figure 8: Distribution of time spent

As the skeleton contained some of the implementation for BF and EF, it required less time to code. As such, we allocated 30% of the time on coding and focused more on developing tests for unit-testing, integration testing, system testing and TDD testing. As we also did regression testing frequently after our bug fixes, test execution took up a fairly high proportion of our time. The coding process itself also required us to execute the tests and analyze the errors, requiring some time on our end to do so. Overall, we allocated a large portion of our time to testing as we understand the importance of the role it plays on the quality of our project.

5. Test-driven Development (TDD) vs. Requirements-driven Development.

- What are advantages and disadvantages of both based on your project experience?

In our project, as we were expected to conduct test-driven development on EF2 in milestone 1 before focusing on the implementation in milestone 2, it allowed us to properly design and develop test cases for every aspect of the features that are indicated in the requirements. As such, during the implementation stage in milestone 2, we had a clearer view of our edge cases, positive and negative cases that we needed to look out for. As the basic test cases were already in-place, we could simply modify the code when the test fails, reducing the chances of missing functionalities. From earlier, we can see that there were fewer bugs found in the applications implemented during milestone 2; cp, mv, rm, uniq, which supports our experience where TDD was useful in identifying the requirements and edge cases to reduce faults.

Additionally, the time taken on coding was significantly reduced in TDD. While writing the test cases in milestone 1, we were already thinking of ways that we could possibly implement the application. The tests used in TDD also help to provide a certain structure to the logic of the code making it easier to implement. We were also able to quickly get feedback on the code as we were still fresh from implementing the code unlike in requirements driven development.

In contrast, when we were conducting requirements-driven development, we noticed that there is a higher chance of missing out some small functionalities. Admittedly, we would start implementation based on the requirements stated in the project description and our own experience of using the application, causing us to have missing functionalities as we were not aware of all the requirements of the application. In addition, we noticed that we tend to focus more on positive cases while implementing during requirements-driven development. We realized that this may cause us to miss some negative cases and corner cases if we were not meticulous enough.

Applications developed using requirements driven development took longer as we did not have test cases to validate it. Most checks in the initial phases before tests were written were done through manual execution.

However, there are some advantages to Requirements-driven Development. As we worked on the implementation before developing the test cases, instead of being focused on modifying the code to pass the test case, we were more attentive to the exception handling and corner cases in our code. As a result, more test cases accounting for those cases were built, reducing the potential for bugs as compared to the test cases developed during test-driven development. Hence, at times, test-driven development may give a false sense of assurance if there are many corner cases in the code that we are prone to miss.

The presence of bugs found during hackathon in applications written in both TDD and requirements driven development suggest that they are both limited in that they are only able to catch bugs that have been accounted for in the tests itself and not missing functionalities that were not considered.

6. Do coverage metrics correlate with the bugs found in your code during hackathon (and not only)?

- What 10% of classes achieved the most branch coverage? How do they compare to the 10% least covered classes in terms of code?
- Provide your opinion on whether the most covered classes are of the highest quality. If not, why?

Coverage: All in cs4218-project-ay2021-s2-2021-team22 ×

100% classes, 93% lines covered in package 'sg.edu.nus.comp.cs4218.impl.app'

Element	Class, %	Method, %	Line, %	Branch, %
args	100% (6/6)	100% (34/34)	94% (167/176)	78% (51/65)
CatApplication	100% (1/1)	100% (6/6)	90% (73/81)	80% (17/21)
CdApplication	100% (1/1)	100% (5/5)	94% (33/35)	81% (13/16)
CpApplication	100% (1/1)	100% (10/10)	96% (90/93)	88% (39/44)
EchoApplication	100% (1/1)	100% (2/2)	100% (16/16)	75% (3/4)
ExitApplication	100% (1/1)	100% (2/2)	100% (3/3)	100% (0/0)
GrepApplication	100% (1/1)	100% (7/7)	89% (115/129)	60% (24/40)
LsApplication	100% (2/2)	85% (12/14)	89% (117/131)	67% (29/43)
MvApplication	100% (1/1)	100% (3/3)	100% (57/57)	92% (13/14)
PasteApplication	100% (1/1)	100% (8/8)	91% (137/150)	78% (36/46)
RmApplication	100% (1/1)	100% (3/3)	92% (46/50)	63% (12/19)
SplitApplication	100% (1/1)	100% (10/10)	93% (207/221)	55% (32/58)
TeeApplication	100% (1/1)	100% (8/8)	90% (76/84)	64% (20/31)
UniqApplication	100% (1/1)	100% (6/6)	96% (116/120)	71% (32/45)
WcApplication	100% (1/1)	100% (8/8)	95% (161/168)	56% (29/51)

Figure 9: Application code coverages

The above showcases our method, line and branch coverages for the Shell application. The top 10% and bottom 10% classes covered are summarized below in a table (ExitApplication is not considered due to its simplicity):

	Method	Line	Branch	Overall
Highest	All Applications (100%)	MvApplication (100%)	MvApplication (92%)	MvApplication (392%)
	except LsApplication	EchoApplication (100%)	CpApplication (88%)	CpApplication (384%)
Lowest	LsApplication (85%)	LsApplication (89%)	WcApplication (56%)	LsApplication (341%)
		GrepApplication (89%)	SplitApplication (55%)	

Figure 10: Analysis of applications of highest and lowest coverage

All the valid bugs discovered during the Hackathon were also analyzed and the following shows how many bugs were related solely to each Shell application class.

S/N	Application	Bug count
1	LsApplication	3
2	UniqApplication	2
3	SplitApplication, MvApplication, GrepApplication, RmApplication, LsApplication, WcApplication, CpApplication	1

Figure 11: Classification of bugs found from hackathon by occurrences

Comparing the coverage metrics with the bug count for every Application class, we can see that LsApplication, having scored the lowest in terms of overall coverage (method, branch, line), also produced the most bugs (3 / 14 bugs discovered). This seems to suggest a negative correlation between test coverage and number of bugs.

However, we note that the applications with the highest overall coverage, namely MvApplication and CpApplication, were not the most bug-free applications as both produced 1 bug each. From the above data observations, we can deduce the following for any non-perfect code base:

- A unit with low test coverage equates to higher presence of bugs since some bugs will not be unveiled by the test.
- A unit with high test coverage does not equate to an absence of bugs.

It is therefore our opinion that the most covered classes are not necessarily of the highest quality. Comprehensive testing does not mean high quality, bug-free code. This is because code coverage only tells us the percentage of code that is executed by the tests; it does not reveal anything about the quality of the class under test and quality of the tests written. If the tests do not consider corner cases and unexpected input, the class may still be buggy even though all lines, methods and branches of the class are covered.

7. What testing activities triggered you to change the design of your code?

- Did integration testing help you to discover design problems?

Our design code was shaped by the testing activities conducted during the project duration. Of the testing activities, regression testing and integration testing contributed to most changes made in the design of our code.

In Milestone 1, we primarily focused on requirements-driven development and unit testing as such we overlooked the correctness of commands when the shell-based applications were to be integrated. Upon integration testing, we discovered several design flaws such as not replacing newlines with spaces when command substitution is integrated.

Usage of global and static variables would also make it difficult to conduct unit testing, as they are generally harder to mock during unit testing. We hence avoided them and only used them for string constants such as exception messages.

Additionally, code quality standards from the PMD rules made us change the design of our code. A case-in-point is the utilization of helper classes to make the code DRY and the adherence to SRP to reduce our “GodClassException” violations. Long methods were also trimmed and abstracted into sub-methods to ensure SLAP. All these were done in consideration that our code will eventually have to be read by someone else, be it the teaching staffs or the hackathon testers.

Through our testing efforts, we also realized that a method that violates SRP and Law of Demeter would make testing much harder. More specifically, if we were to instantiate objects to be used within an unrelated method, then we would have to mock these objects using spy and mock frameworks such as Mockito during unit testing. However, if we were to pass these objects as method parameters, we can simply mock them using our very own classes. An example is shown below:

```
private void copyFile(Path sourceFile, Path destFile) throws IOException {
    FilesUtil copier = new FilesUtil(Files.readAllBytes(sourceFile));
    copier.copy(destFile);
}
```

Figure 12: Snippet of copyFile method before redesign

The above will be harder to test since FilesUtil is a dependency within the method. If we were to rewrite the method as follows:

```
private void copyFile(Path destFile, FilesUtil copier) {
    copier.copy(destFile);
}
```

Figure 13: Snippet of copyFile method after redesign

Then it would be much easier to conduct unit testing for we can simply instantiate a FilesUtil object in the test code and pass it in as a method parameter. This simplification to testing has thereby triggered a change of design in some part of the code.

8. **Automated test case generation: did automatically generated test cases help you to find new bugs?**

- Compare manual effort for writing a single unit test case vs. generating and analysing results of an automatically generated one(s).

Due to the complexity of the inputs needed to trigger certain execution paths in the project, the effectiveness of the automatically generated test cases were minimal. Of the tools we used, the majority of the test cases generated varied inputs for method calls. There were however exceptional test cases from these that were useful in helping to improve the quality of our code by indicating areas where we could be more defensive.

Below are our comparison criteria for effort spent on manual testing and automatically generating test cases.

Setup:

From CS2103, we were all familiar with JUnit and how to run our test cases so there was little to no overhead when setting up JUnit for our manual testing. As our team had never used any of these tools before, there was temporal overhead from setting up some of these tools. Of the tools we used, SquareTest and Diffblue Cover had little overhead and were easy to set up as they came as plugins in the IntelliJ marketplace and tests could be generated with a few clicks. On the other hand, Evosuite had compatibility issues with versions of IntelliJ and required running from the command line. Luckily the lab provided guidance in using it.

Writing:

Manually writing test cases consumed more time than the automatically generated test case. In comparison, the time taken to manually writing a single test case would be able to run the test automation tools multiple times.

Analysis:

Running both manually written test cases and automatically generated test cases required the same effort. However, analysis of the manually written test cases required less effort as being the ones who wrote them, we were familiar with the expected result from the given input. However, for the automatically generated test cases, the input generated by the tools were generic and not specific enough to the project specification. In Figure 13, it can be seen that the input generated by the tools were simply "content" and string array with "value" when broken standard output for rmApplication. The test case was to test the effect a broken standard output would have on our application. However, the inputs themselves would not be valid unless a file named value was present in our directory causing an exception to be thrown if not present. In that scenario, it would then be possible for the assertThrows call to catch a file not found exception rather than a broken standard output exception.

```

@Test
void testRun_BrokenStdout_throwsException() {
    // Setup
    final InputStream stdin = new ByteArrayInputStream("content".getBytes());
    final OutputStream stdout = new BrokenOutputStream();
    rmAppUnderTest = new RmApplication();
    // Run the test
    assertThrows(AbstractApplicationException.class, () -> rmAppUnderTest.run(new String[]{"value"}, stdin, stdout));
}

```

Figure 14: Test case generated by Squaretest

Additionally, the quantity of automatically generated test cases were significantly greater than our manually written test cases. Randoop alone generated 35,000 LOC worth of test cases to analyze.

We also experienced some issues with the execution of test cases generated by the tools. We were unable to run the test cases generated using EvoSuite as we encountered dependency issues which we spent hours trying to resolve but to no avail. In the end, we manually compared the inputs and outputs of the generated test cases.

To summarise, the automation tools are able to generate test cases faster and in greater quantities. However, they required more effort to analyze the test cases input and output. Additionally, the quality of the test cases generated by the tools were lower as the tools did not generate the test cases with the context of the project. To achieve greater efficiency in testing, we could have used the tools to generate test cases for catalog inputs such as broken streams, null values, etc, while writing manual testing for the project requirements.

9. Hackathon experience: did test cases generated by the other team for your project helped you to improve its quality?

Error Messages	Functionality
3	11

Figure 15: Number of bugs per classification

From the classification of valid bugs provided by the other teams, we can see that approximately 20% (3/14) of the bugs were related to defensive programming where the error messages printed by our Java Shell did not match that of Unix. Those were helpful in identifying areas in our code which faults could possibly occur and helped increase our project quality.

The other 80% (12/14) of bugs found consisted of missed edge cases and incorrect formatting of results that lead to errors in the outputs. These test cases identified the missing and incorrect implementation of functionalities in our project.

In modifying our project code to pass these test cases, we were able to increase defensiveness of our code and produce a more functionally complete Shell. Through the hackathon, we were also able to affirm the quality of our own test cases against other teams' projects which also helps to affirm the quality of our project.

10. Debugging experience: What kind of automation would be most useful over and above the IntelliJ debugger you used – specifically for the bugs/debugging you encountered in the project?

- Would you change any coding or testing practices based on the bugs/debugging you encountered in the CS4218 project? Did you use any tools to help in debugging?

Many bugs that we discovered ourselves prior to the Hackathon perplexed us greatly as seemingly similar commands would lead to a rather different output. A good example is the usage of single and double quotes with command substitution. Another is the output difference between `echo `cat file`` and `cat `echo file``, notwithstanding the expected replacement of new lines with spaces in the former. A substantial amount of time was consequently invested in the comparison of execution paths for similar commands.

For the aforementioned bugs, it would have been most useful to automate the process of test-based fault localization against executed statements. However, this will only work if one of the commands is already producing the right output. We can employ the trace alignment technique to isolate statement differences between the two, effectively narrowing our scope of debugging in the code (i.e. fault localization).

Another automation which might have been handy is that of dynamic slicing. There were multiple times we struggled to identify the code that modified a variable, given the long and repetitive function calls which encouraged simply stepping over and hence missing such code. Quick knowledge of dynamic data and control dependencies could have shaved off hours spent on locating the code responsible for the additional spaces in the command output. Although we did not use any debugging tools, had we had more time, we would certainly have researched effective slicing tools and learnt to apply them for fault localization.

11. Propose and explain a few criteria to evaluate the quality of your project, except for using test cases to assess the correctness of the execution.

Apart from the coverage criterias taught in class, the following 3 are some criterias which can be utilised to gauge the quality of our project:

1. Average Percentage of Faults Detected

Average percentage of faults detected (APFD) is a code quality metric that measures the rate of bugs found relative to the proportion of test cases being executed. This metric enables us to evaluate the efficiency and effectiveness of our test cases and how sufficient our testing is. It

also allows us to focus our testing and implementation efforts on areas where testing is not sufficient enough.

2. Cyclomatic Complexity

Cyclomatic Complexity measures the structural complexity of the project. It is a quantitative measure of the sum of linearly independent paths through the source code. Higher cyclomatic complexity indicates that the code is more complex and hence it would be more difficult to test. As a result, it is more likely to result in faults. So, lower cyclomatic complexity would mean better code structure and quality as it maintains a readable codebase.

3. Method Cohesion

Method cohesion metric determines how well the methods of a class are related to each other. It measures the proportion of methods within a class that uses a particular attribute or property of that class. A low value means that not a lot of methods are using those attributes, so it may indicate that the methods are performing unrelated functions. A high method cohesion means that attributes are effectively used by the methods, thus indicating better encapsulation.

12. Which of the above answers are counter-intuitive to you?

Cyclomatic complexity is rather counterintuitive in the assessment of our project quality.

This is because cyclomatic complexity in essence measures how structurally complex our project is, suggesting that the more complex it is, the more likely it is to have faults. However, this is not a fair representation of our project quality given that a complex structure is essential as the shell's intended behaviour is one that consists of much recursion and conditional branching, e.g. `shellImpl.java main()` method employs a while loop, and throughout the application a lot of conditional branching is involved, such as in `ArgumentResolver` class, to check for the command inputs of users to dictate how the program should behave.

Moreover, cyclomatic complexity is actually most useful in identifying the unique paths of our program and by extension the lower bound of the number of test cases we need to write to test effectively our program to ensure we cover path coverage. However, even in the hypothetical scenario where all possible infinitely many paths are tested, we still cannot imply that no bugs are present and that our project is perfect and bug-free. In fact, for our project, testing a good number of possible values and their combinations is more crucial.

Hence, we believe that out of the three proposed criteria, cyclomatic complexity does not help to evaluate our project's quality as much and is hence counter-intuitive.

13. Describe one important reflection on software testing or software quality that you have learnt through CS4218 project in particular, and CS4218 in general.

One crucial lesson that we had learnt from the project is that statistics and coverage criteria only help to improve the proportion of code / path covered by tests and in doing so gives the inaccurate perception that the SUT is functional and more bug-free.

The reality is that such figures do not correlate with test quality and sufficiency. Corner cases and special inputs may have been overlooked, especially with regards to integration between the systems. This was also apparent from our experience using automated testing tools. Despite the large number of test cases and LOC produced, we realised through the hackathon exercise that we had overlooked some of the said complex scenarios and edge cases. Therefore, it is essential that we do not refer solely to coverage criteria to assess the comprehensiveness of our test cases.

For CS4218 in general, we learnt that Software Testing is much more than simply unit, integration, system and acceptance testing as we had perceived from past modules and internships.

Prior to this module, our knowledge of testing was rudimentary and was limited to the above types of testing acquired through CS2103T. CS4218 has opened our eyes to a new realm by exposing us to different testing strategies. We have also learnt to appreciate the systematic process of test case derivation from specification, as well as the resolution of frequently-encountered combinatorial explosion through key combinatorial testing concepts. Moreover, our knowledge boundary is further transcended by the revelation of fault localization through static and dynamic slicing using data/control dependencies. We were also equipped with the orthogonal approach to object-oriented testing, both intra and inter class, empowering ourselves to tackle the intricacies of OO testing. Another concept which was nothing short of astonishing is the invariant/ conditional-based assertions and oracle generation via contracts. This module also imparted in us performance and distributed systems testing, both of which were extremely relevant in designing scalable systems, but admittedly often overlooked.

14. We have designed the CS4218 project so that you are exposed to industrial practices such as personnel leaving a company, taking ownership of other's code, geographically distributed software development, and so on. Please suggest a new topic for the project that would bring similar or more benefits to the students

Our genuine opinion is that the project is already rather comprehensive and provides sufficient exposure to many testing concepts that will be relevant in the industry. We are highly satisfied with what the CS4218 teaching team has excellently curated for us.

Regrettably, as the Java Shell is inherently command-line based without a GUI, students missed out on the opportunity to apply GUI testing approaches such as Manual/Model-based testing

and Record & Replay. Considering the importance of UI/UX and the extensive use of GUI and GUI test frameworks in contemporary industry-level products for end-users, it would be beneficial if students could be exposed to a practical tinge of GUI testing.

For this reason, we suggest the integration of a fully-functional window that acts as the GUI of the shell. The window may resemble that of command prompt (in Windows) or Bash (in Linux) with customization and personalization features. When issuing commands to the existing Java shell, one huge inconvenience we students faced was the incapability to verify the current working directory (i.e. the lack of 'pwd' command). The window may therefore also display the current working directory beside the prompt for user command, much like Command prompt and Bash. An additional GUI testing requirement can then be incorporated into the project description. Understandably, this would introduce more workload for the students, so we also propose a reduction in the number of applications to be implemented.

===== **END OF QA REPORT** =====