

Software Quality Assurance (QA) Report

Analysis across project artefacts and milestones

Write a report covering the following topics and questions. Please provide numbers, plots and explanations with few lines of text (say 1-2 paragraphs) for each question. Note that you should not simply answer each question, but rather create a well-rounded report about your testing efforts during the project. Feel free to add any other information you consider useful regarding your testing efforts.

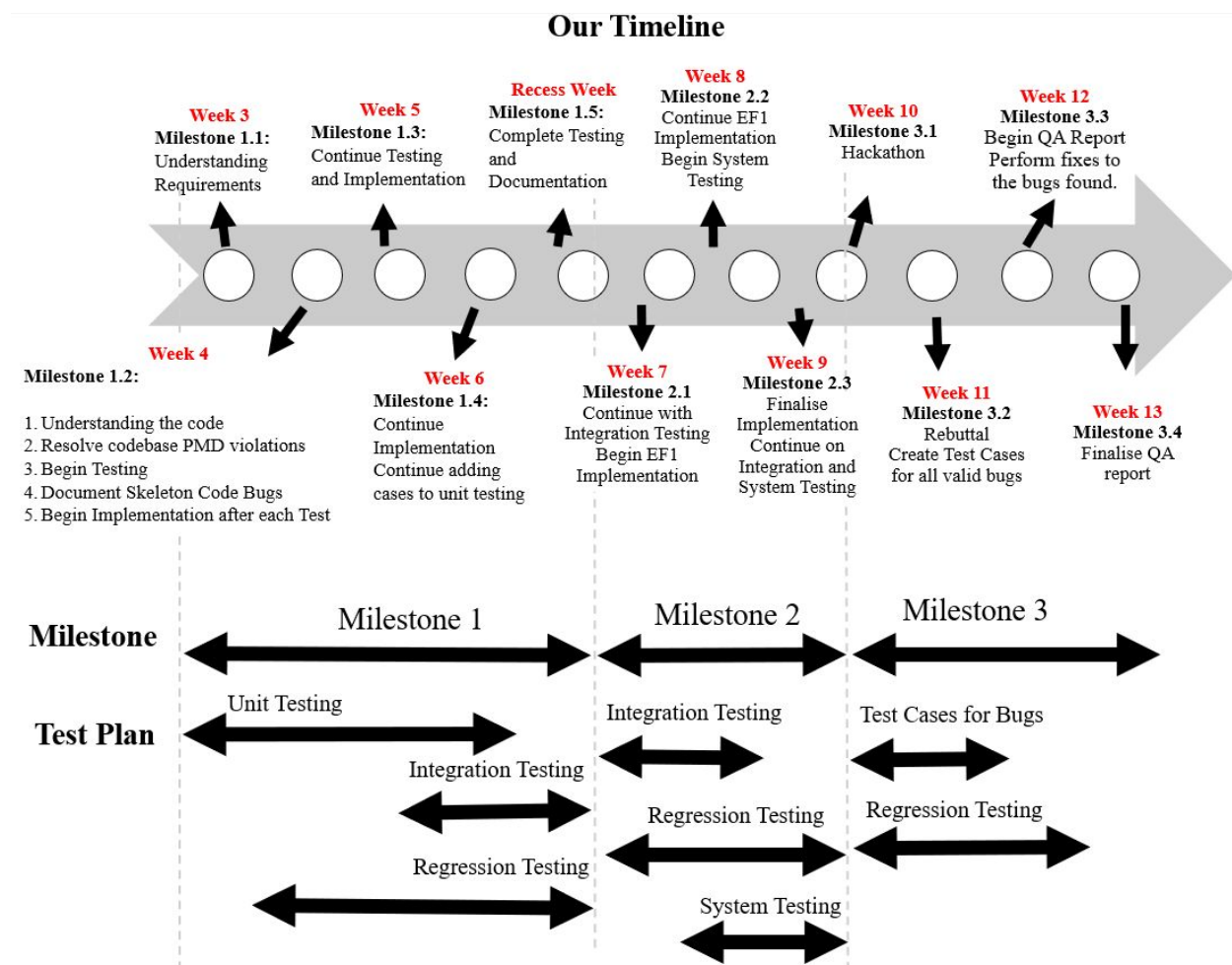
Hint: Read the questions you need to cover in the QA report and plan what information you need to **keep track of and document while working on Milestones 1 & 2, Hackathon, and Rebuttal**. We suggest the use of an issue tracker and additional files to document your testing efforts throughout the semester. You will need such information when preparing your QA report for Milestone 3.

Topics and questions to be covered in the report

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

After finalising our CS4218 module Project source and test code, including bug fixes from the hackathon and rebuttal, we have written a total of 6333 lines of code (LOC) for the source code and 25557 lines of code (LOC) for the test code.

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?



Summary of Test Plan, Methods and Activities

Before we start any form of testing or implementation, our team reviewed the requirements. This involved reading through the project specifications as well as trying out the commands in Linux Terminal. We also asked both the Lecturer and TA for clarification with regards to some of the grey areas of the requirements. This is an important step as we need to define what are the expected behaviours of the features before we can conduct any form of testing.

In Milestone 1, the primary testing activity is unit testing. Due to our test-driven development approach, we strive towards creating test cases before implementation. As such, we created most of our unit test cases before we started our implementation. However, as we implement, we sometimes learn more about the feature, which provided us with more ideas that we used to generate more unit tests cases. Therefore,

you can see that unit testing stretches close towards the end of Milestone 1. In terms of regression testing, whenever we implement our features, we run our tests to ensure that any new code does not break the existing functionalities. This occurs after we begin our implementation, and most of the unit test cases are already up. Towards the end of Milestone 1, we performed some integration testing. We timed integration testing this way, to ensure that we have test cases ready before the features that calls for it are implemented. For example, sequence command, and pipe operator both require integration testing as its commands involve other applications in order for work. As such, we create integration testing cases for sequence command and pipe operator before they are implemented in milestone 1. When both the testing and implementation are done, we perform coverage analysis to see which areas of our testing is not substantive enough.

In Milestone 2, the primary testing activity is integration testing. We continued with our test-driven development approach which we started in Milestone 1. We focused on enhancing our test cases to make it more comprehensive. We do this by combining both the test cases that we have written in EF1 and TDD test cases provided by the teaching team. With the combined test cases, we conducted test driven development to implement EF1 and resolve missing functionalities not considered in implementation of basic functionality and EF2. One example of missing functionalities is the consideration of file permissions cases. We also placed emphasis on ensuring integration between the applications (e.g. sed, grep) and their corresponding subunits (e.g. sedArguments, grepArguments) all integrate properly without breaking. Note that since unit testing was done for all these subunits as well, we are able to first detect which unit-interaction is causing the bug (through integration tests), and pinpoint the specific buggy unit (through unit testing). Regression testing begins at the start of milestone 2, we began implementing features that have already been covered by the integration testing we did in Milestone 1. For system testing, we wrote some key test cases (e.g. corner cases, common error-generating cases) towards the end of milestone 2. As system testing is not automated, it is to facilitate our manual testing of our own shell as well as for the other teams' shells during the hackathon.

In Milestone 3, we have made some fixes to the valid bugs found in the hackathon, and have created test cases for all the valid bugs. While we are making these fixes, we also actively perform regression testing to ensure that all our previous test cases do not fail. In the event where there are failed test cases, we would either update the test case based on the latest changes, or relook at the new code that the fixing process has introduced. Once the fixes and test cases for all bugs are up, we focused on the Quality Assurance Report.

What was the most useful method or activity that you employed?

The most important activity that we did was coverage analysis. By looking at the results of our coverage analysis at both IntelliJ and Jacoco, we are able to pinpoint which part of the code has not been tested. This allows us to adopt a systematic approach by standardising the % coverage that we expect as a group, so that all parts of our implementation are sufficiently covered. We looked into 3 key areas: method, branch and line coverage. Each of these areas serve its own purpose; method coverage highlights to us which method in the classes have not been tested. Branch coverage will tell us whether we tested all branches in conditional statements, while line coverage will indicate to us which part of the code we have not tested. It is through this analysis process that we can have more assurance that our code as a whole has been sufficiently tested and that no substantial part of the code has been left untested which boosts our confidence in the quality of our project which will be elaborated in Question 13.

3. Analyze the distribution of fault types versus project activities:

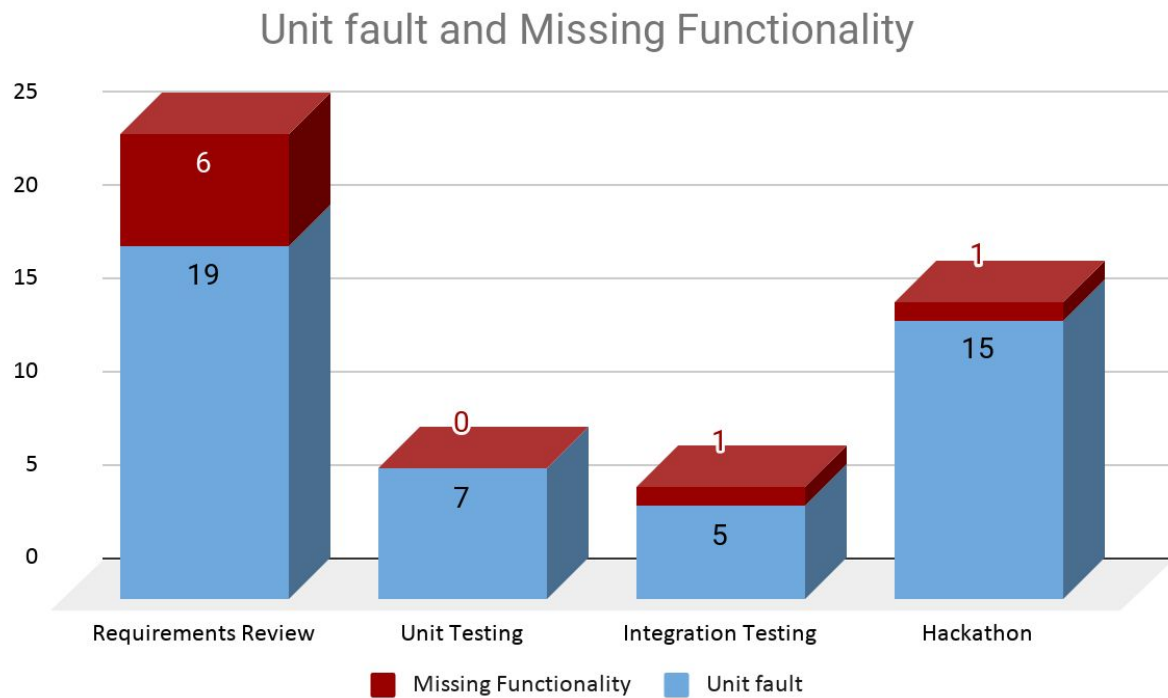
3.1. Plot diagrams with the distribution of faults over project activities.

Types of faults: unit fault (algorithmic fault), integration fault (interface mismatch), missing functionality. Add any other types of faults you might have encountered.

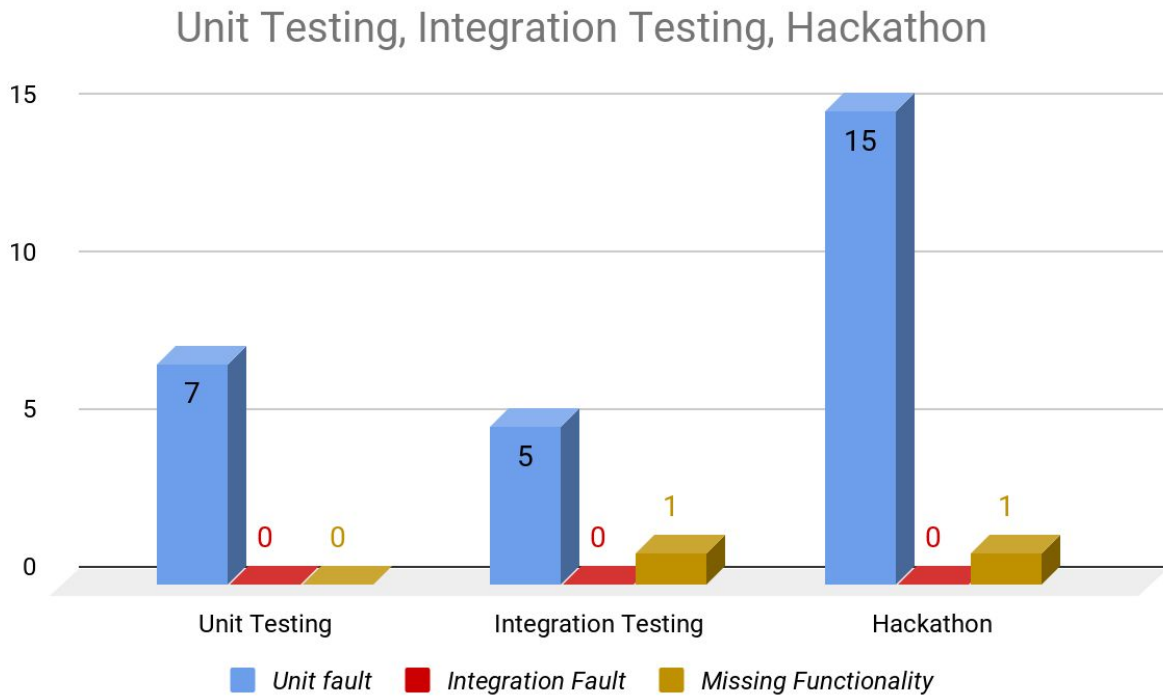
Activity: requirements review, unit testing, integration testing, hackathon, coverage analysis. Add any other activities you have conducted.

Each diagram will have a number of faults for a given fault type vs. different activities.

Discuss what activities discovered the most faults. Discuss whether the distribution of fault types matches your expectations.



The activity that discovered the most faults is requirements review as shown in the graph above where it has a total of 25 faults. This is most likely due to the fact that in the project specifications, certain applications are not well defined and require clarification with the lecturer and comparison with the unix shell. As a result, there is likely a mismatch between the actual implemented behaviour by our shell and the expected behaviour required by the lecturer or by the unix shell.



The distribution of faults type found matches our expectation as we expected that unit faults are the most faults found as shown in the graph above. This is due to the fact that unit fault (algorithmic fault) commonly occurs due to mismatch in expected and actual requirement behaviour as mentioned earlier which resulted in logical faults to occur which is a type of unit fault.

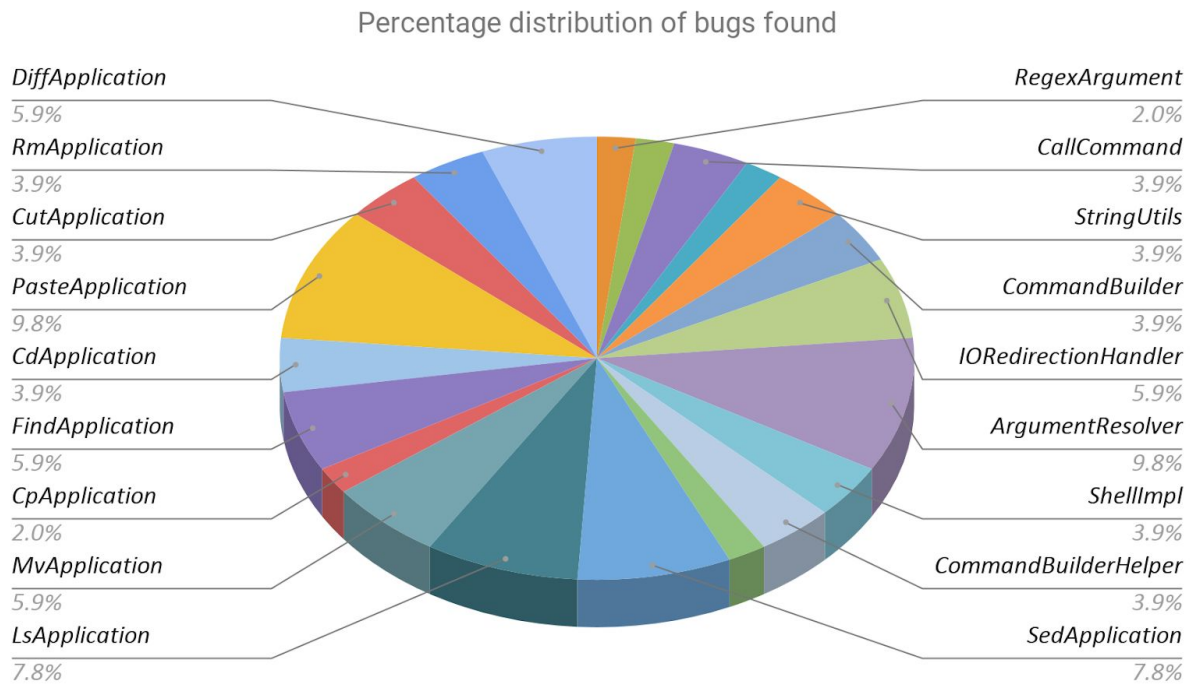
3.2. Plot a diagram for distribution of faults found in basic functionality (old code) during activities on adding extended functionality (new code):

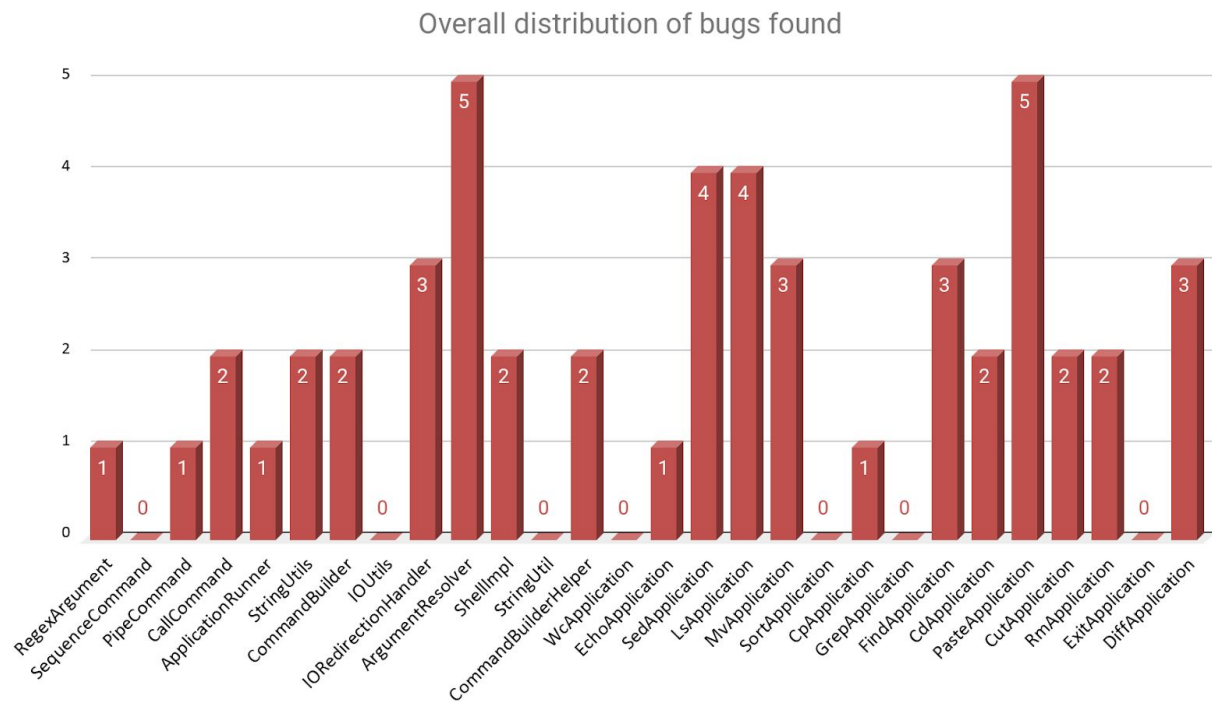
Activity: integration testing, hackathon, coverage analysis. Add any other activities you have conducted.

We plotted a graph based on the distribution of bugs found across the different Java classes for the entire source code of the CS4218 module project as shown in the graph below. This is in line with the fact that the project specification categorises the basic functionalities and extended functionalities based on the classes such as RmApplication in Basic functionality and CpApplication in extended functionality 1.

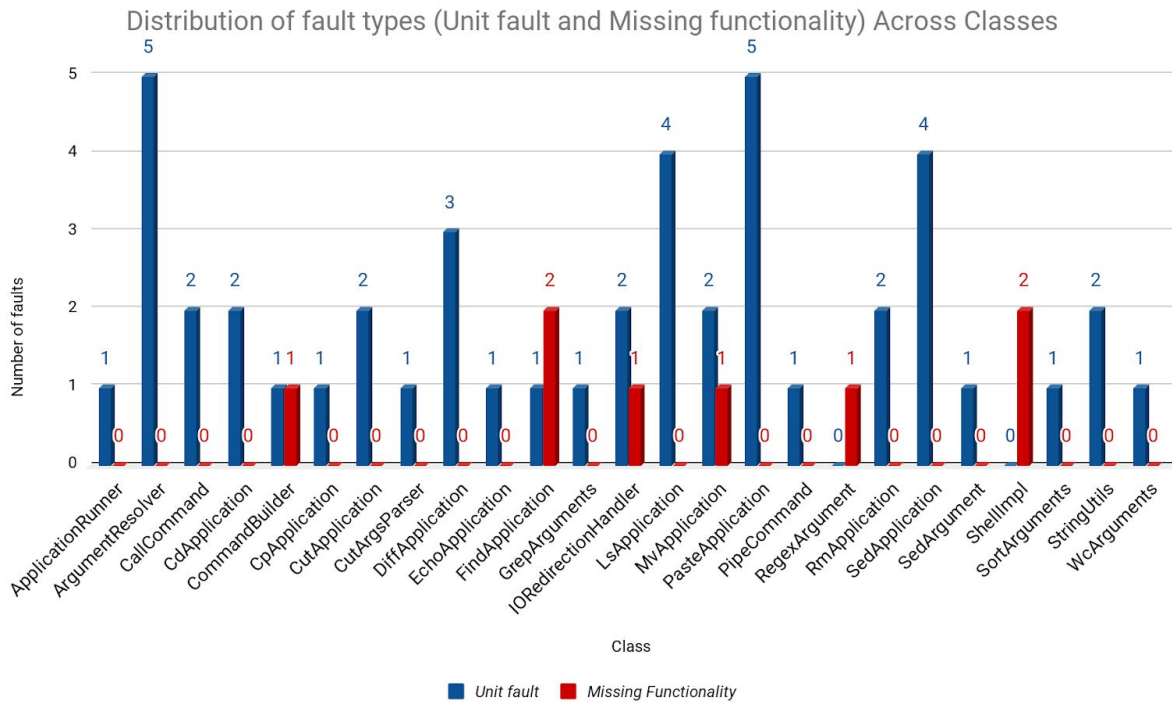
In the graph below, we can see that a larger proportion of the bugs are found in the

ArgumentResolver (9.8 percent) and PasteApplication (9.8 percent). The distribution of other bugs are more evenly spread throughout the program.





Discuss whether the distribution of fault types matches your expectations.



The distribution of fault types (mainly unit fault and missing functionalities) across the CS4218 project source code can be seen in the graph above. The distribution of fault types meets our expectation where in general there are more unit faults (algorithmic or logic fault) across classes compared to missing functionality. This is due to the fact that during the implementation phase we try to implement our shell functionality as close to the unix shell behaviour as well as the project specification as possible. As a result, we expect most if not all functionalities to be implemented hence, fewer missing functionalities related faults. This is compared to unit fault (algorithmic/logic fault) as structural testing is unable to detect code that is not implemented (missing logic) and system testing which is largely based on tester's experience would only be able to detect missing logic if all the different possible cases are considered during the system testing. In other words, there are higher chances that there are cases in which we did not consider such as checking for file permissions resulting in a higher number of unit faults.

3.3. Analyze bugs found in your project according to their type (in all milestones).

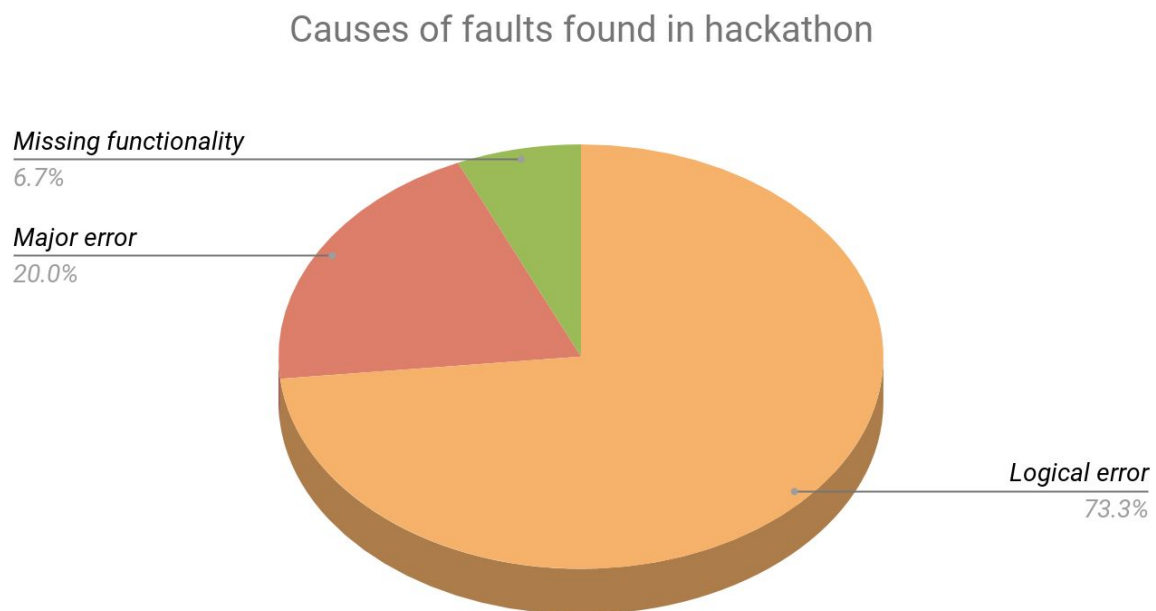
Analyze and plot a distribution of causes for the faults discovered by Hackathon activity.

We grouped the type of errors together into 3 categories for readability.

Logical Error: Error in constants; Error in identifiers; Error in arithmetic (+,-), or relational operators (<, >); Error in logical operators; Localized error in control flow (for instance, mixing up the logic of if-then-else nesting)

Major error: Errors that will cause the application to hang.

Missing functionality: Errors that arise from requirements not being met fully. E.g. Missing logic to allow for multi-dash input in Paste.



Based on the graph above, we can see that most of our causes of faults found during Hackathon is due to logical error with 78.6% compared to missing validation (i.e. missing functionality) with 21.4%. This could indicate that a need for more testing in terms of the logic in the code. Logic bugs, by its nature, are harder to detect. As such, there could be a need to write more system test cases that could potentially detect logic bugs in the code. This could enable us to reduce the amount of logic bugs in our code. In terms of major bugs, we believe that more substantive manual testing could be able to detect such bugs. We could use test cases that test the basic functionality of the

applications to ensure that the application works properly without crashing or exhibit any other abnormal behaviour. We also understand that these major bugs could occur due to new implementation, which means that we need to do more regression testing. Furthermore, although missing functionality bugs make up only 6.7%, there could be a need for us to check through whether all the requirements have been implemented. We could do this by generating test cases for every requirement in the project. This will ensure that all functionalities are present in the final product.

Analyze and plot a distribution of causes for the faults discovered by Randoop, or other tools.

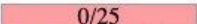

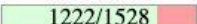
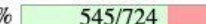
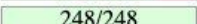
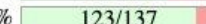
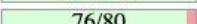
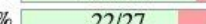
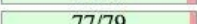
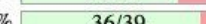
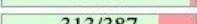
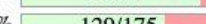
We do not integrate Randoop into our project hence, we do not have data to plot for Randoop. Instead we decide to show the automated Maven PIT generated report which is an additional tool we have integrated into our project.

Pit Test Coverage Report

Project Summary

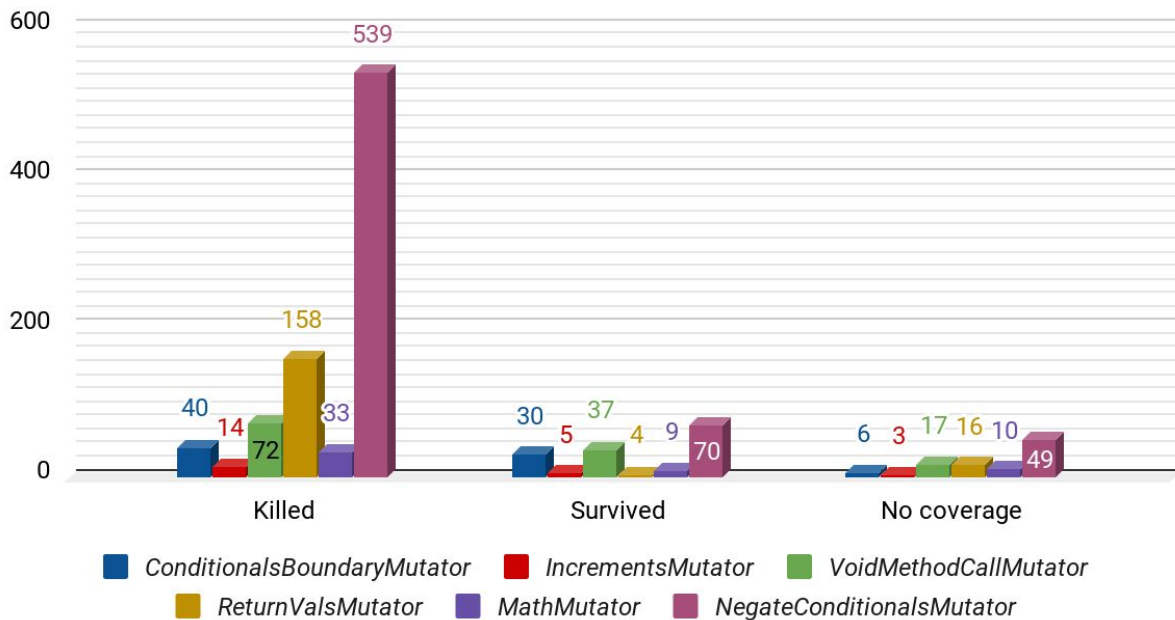
Number of Classes	Line Coverage	Mutation Coverage
40	82% 	77% 

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	80% 	75% 
sg.edu.nus.comp.cs4218.impl.app.args	7	100% 	90% 
sg.edu.nus.comp.cs4218.impl.cmd	3	95% 	81% 
sg.edu.nus.comp.cs4218.impl.parser	6	97% 	92% 
sg.edu.nus.comp.cs4218.impl.util	8	81% 	74% 

Based on the above generated Maven Pit report, our test suite is 77% mutation coverage (855 of 1112 mutants are killed) 257 mutants survived.

Overall distribution of faults in mutation testing



As shown in the above figures, we can see that most of our faults come from the mutation of conditional boundary in terms of percentage ($100\% - 53\% = 47\%$) and mutation of negated conditions in terms of absolute value (70)..

From this we can infer that our test suite is ineffective in exposing simple (artificial) conditional boundary and negation condition mutators hence, it is ineffective at exposing more complex (real) mutators of this nature.

Is it true that faults tend to accumulate in a few modules? Explain your answer.

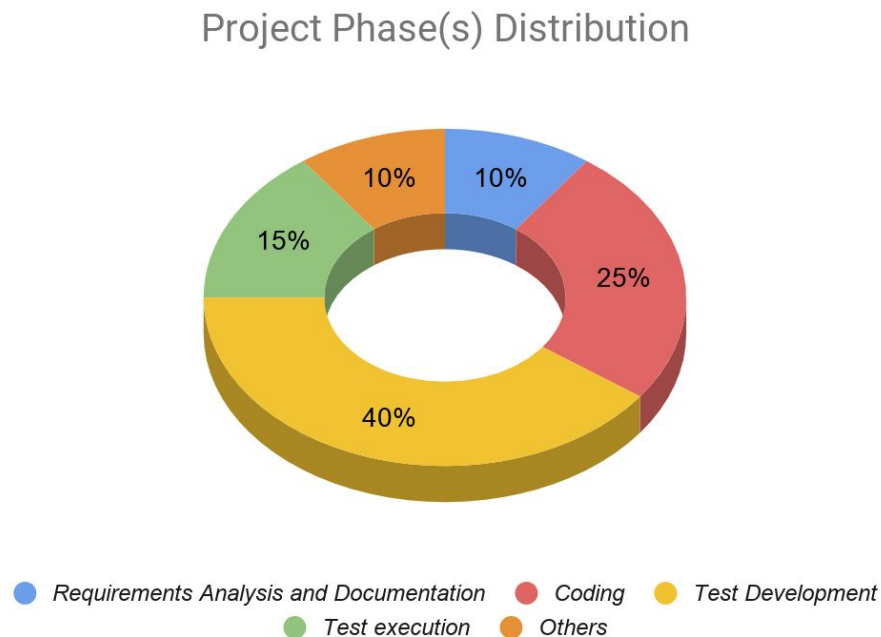
No it is not true that faults tend to accumulate in a few modules. Faults, both logical and missing validations tend to spread across the entire project. So it does not accumulate in one particular part of the program. For example, logical errors can be found in the applications for problems like wrong assignment of destination folders for applications like MvApplication and also missing validation errors such as exceptions not handled in MvApplication for moving a parent folder to a child folder.

Is it true that some classes of faults predominate? Which ones?

It is true, logical faults are more apparent in our program due to wrong logic implementation.

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

Below is the graph to provide distribution on the estimation of on each phase in the project.



Here are a few things to elaborate on certain phase(s) that is not specified in the graph:

- Requirements Analysis and Documentation: The requirements specification is mostly provided except certain specifications require refinement by comparing with unix behaviour.
- Others: Setting up of Maven, Travis Continuous Integration set up for regression testing, GitHub repository set up, Maven PIT for mutation testing, automated Jacoco coverage report generation.

5. Test-driven Development (TDD) vs. Requirements-driven Development. What are advantages and disadvantages of both based on your project experience?

Test-Driven development is more advantageous over requirements-driven development when the program is a brown-field project. For example, for features such as globbing or find applications, the source code given to us was already semi-implemented. In such a situation, by developing or getting test cases first, bugs can be easily identified and helped developers pinpoint where the bugs lie. In cases where lines of code are long, test driven test cases can easily find faults in the program, saving time from trying to read the code and find faults from the code itself with the aid of IntelliJ debugging feature. It also helps us save time from debugging. Furthermore, Test-driven development requires us to plan and design the implementation as well as the outcome we should be getting before we design our test cases.

However, Test-Driven Development might not be as useful as requirements-driven development if the application or feature has not been implemented or new to the system. As there is nothing or little to test yet, for example, the move application in our project, test cases would not be that helpful in helping us design and develop the application.

Requirements-driven development is advantageous where applications are not implemented at all. By using requirement specifications in requirements-driven development, we are then able to come out with our assumptions, and also how a feature should behave. The difference is that we can come out with our assumptions and not be restricted to the test cases provided to us. Furthermore, requirements-driven development also helps us in designing our features according to the requirement. For example, in designing the requirements, we ourselves act as the users of the shell together with the project specification, we design the output using our own expectation of UNIX shell, and this allows us to form our requirements so that we can develop our shell based on the requirements. However, in requirements-driven development, the ambiguity of the applications (how much resemblance to unix shell) can be a challenge for us developers. We are unsure of how much we are required to develop and how we should approach different types of inputs. Furthermore, requirements-driven development can also pose difficulties in integration of our different applications due to differences in assumptions.

In conclusion, both types of development have its advantages and disadvantages and both types were used in our project as mentioned in Question 2 in Milestone 1 and 2,

therefore, from the experience of the project, both when used together can help us design our solution better.

6. Do coverage metrics correspond to your estimations of code quality?

For example, what 10% of classes achieved the most branch coverage? How do they compare to the 10% least covered classes? Provide your opinion on whether the most covered classes are of the highest quality. If not, why?

Element	Class, %	Method, %	Line, %	Branch, %
args	100% (7/7)	100% (42/42)	100% (248/248)	94% (177/188)
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% (16/16)	90% (219/242)	85% (146/170)
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)	95% (132/138)	92% (86/93)
LsApplication	50% (1/2)	88% (15/17)	85% (160/188)	79% (89/112)
MvApplication	100% (1/1)	100% (7/7)	93% (100/107)	90% (47/52)
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% (5/5)	93% (72/77)	88% (46/52)
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)

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)	100% (40/40)	100% (22/22)
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)

100% classes, 95% lines covered in package 'sg.edu.nus.comp.cs4218.impl.cmd'				
Element	Class, %	Method, %	Line, %	Branch, %
CallCommand	100% (1/1)	100% (4/4)	100% (21/21)	100% (6/6)
PipeCommand	100% (1/1)	75% (3/4)	96% (30/31)	100% (14/14)
SequenceCommand	100% (1/1)	75% (3/4)	89% (25/28)	100% (8/8)

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)	97% (35/36)	83% (15/18)
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)

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% (87/87)	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)

Element	Class, %	Method, %	Line, %	Branch, %
app	95% (23/24)	98% (152/155)	94% (1675/1778)	89% (1081/1205)
cmd	100% (3/3)	83% (10/12)	95% (76/80)	100% (28/28)
parser	100% (6/6)	100% (24/24)	98% (78/79)	84% (27/32)
util	88% (8/9)	96% (49/51)	90% (344/380)	81% (169/207)
ShellImpl	100% (1/1)	50% (1/2)	16% (4/25)	0% (0/6)

Element	Class, %	Method, %	Line, %	Branch, %
AbstractApplicationException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
CdException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
CpException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
CutException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
DateException	0% (0/1)	0% (0/2)	0% (0/4)	100% (0/0)
DiffException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
DirectoryNotFoundException	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
EchoException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
ExitException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
FindException	100% (1/1)	100% (2/2)	100% (4/4)	100% (0/0)
GrepException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
InvalidArgsException	100% (1/1)	50% (1/2)	50% (2/4)	100% (0/0)
LsException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
MkdirException	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
MvException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
PasteException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
PwdException	0% (0/1)	0% (0/2)	0% (0/4)	100% (0/0)
RmException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
SedException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
ShellException	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
SortException	100% (1/1)	50% (1/2)	50% (2/4)	100% (0/0)
WcException	100% (1/1)	50% (1/2)	50% (2/4)	100% (0/0)

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% (236/244)	92% (2177/2342)	88% (1305/1478)
EnvironmentHelper	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)

- Total number of classes: 66
 - 10% of classes that achieved the most branch coverage (based on the images above):
 - 7 classes:
 - CallCommand
 - PipeCommand
 - SequenceCommand
 - EchoApplication
 - GrepArguments
 - WcArguments
 - ArgsParser
 - 10% of classes that achieved the least branch coverage (based on the images above):
 - 7 classes:
 - LsApplication
 - RegexArgument
 - GrepArgsParser
 - RmApplication
 - MyPair
 - CutArgsParser
 - ShellImpl

In general, we estimate that our code quality is relatively high (as shown in the images above). This can be seen from Question 4, since we spent roughly 55% of our time on testing. This is also evident from our coverage report, which showed an overall high amount of coverage > 80%.

In terms of branch coverage, we looked at the top 10% classes with the most branch coverage, and the bottom 10% classes with the least branch coverage. We found that low branch coverage may not necessarily lead to poorer code quality. We judge code quality for this case by the number of bugs that were valid in the hackathon. For example, for the RmApplication, although it belongs in the bottom 10% classes in terms for branch coverage, it has no valid bugs reported by the testing team during the

Hackathon. However, with that said, we find that generally, the top 10% classes had better code quality as they had few (if any) valid bugs reported by the testing team during the Hackathon. There are exceptions to this, for example, the GrepArgument, which had one reported bug in the Hackathon. When we weigh all this information together, we can conclude that higher branch coverage could lead to better code quality but a low branch coverage may not necessarily lead to poorer code quality. There could be factors present that affected the branch coverage which lowered the coverage percentage.

In conclusion, our group's opinion is that the most covered classes are not necessarily the classes with highest quality as shown in our argument above.

7. What testing activities triggered you to change the design of your code? Did integration testing help you to discover design problems?

With unit testing, we strive to ensure that every method only has one responsibility (high cohesion). This high cohesion for every method will not only make testing and debugging easier, it will also make the code more organised and readable. At times, when we have methods that are not cohesive and highly coupled with other methods/ or classes, it will be more difficult to unit test the method to ensure that a particular behaviour is implemented correctly due to high coupling involved. It is due to this reason hence, we have to change and improve the design of our code by abstracting the other responsibilities into new separate methods.

On the other hand, integration testing helped us detect discrepancy in codes across the different subunits. For instance, the argument class (e.g. GrepArguments) throws a specific exception message, while the main application class (e.g. GrepApplication) throws a general exception message. Given that the exception message thrown is not specific throughout, changes will have to be made in the main application class design, in order to ensure that the errors thrown do not default to the general exception and are consistently specific.

Integration testing also helped us detect bugs which affect two different applications. For instance, in our sequence command integration testing, we managed to find bugs in paste application with regards to BufferedReader not closed. This caused the stream to remain open and in use hence, the remove command is unable to remove files whose streams are not closed. It allowed us to change the design to ensure that the BufferedReader closed appropriately.

8. Automated test case generation: did automatically generated test cases (using Randoop or other tools) help you to find new bugs?

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

Our group decided to use Maven PIT as our automated generated test cases tool. The purpose of using Maven PIT is to increase the confidence of our test cases. This does not help us to find new bugs. Furthermore, a lot of time is consumed in the process of Maven PIT mutation testing. As such, in terms of cost-benefit analysis, the benefits of Maven PIT may not outweigh its costs (i.e. execution time), particularly in the software engineering industry where software testers often face time constraints.

On the contrary, manual effort for writing a unit test case helps in controlling the flow of execution inside the test cases. In addition, by writing our own test case, we increase the readability of our test cases. Readability is one of the criteria that we want to achieve throughout the project which will be elaborated in Question 12. Other automated test cases generation tools will tend to use unknown variables.

Although automated test case generation helps to reduce the time and effort needed in testing, it also generates unrealistic or infeasible test cases which may not add value to our testing as a whole. Therefore, it is our opinion that manual testing is more effective in allowing us to apply the concepts taught in CS4218, as well as facilitate better planning of our test cases; in other words, we believe that nothing beats an actual human tester to create test cases.

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

The test cases generated by the other team definitely aided in improving the project's quality in terms of correctness which is one of the criteria we have set and elaborated on in Question 12. Furthermore, the test cases generated by the other team are clear and easy to understand.

Firstly, the test cases generated by the other team helped us in identifying areas where our project contained bugs and covered cases that we had missed out when designing

our own test cases. A prominent example is in bug report 27 of Hackathon, it was identified as a bug by our shell as the case was not considered during testing and implementation. Therefore, the bug report has aided in helping us to achieve greater accuracy and correctness in our implementation.

Secondly, during the Hackathon's rebuttal session, we were able to clarify what are some implementations we are required to do but were not implemented. An example is having backslash for folders in the FindApplication, where initially we designed it to follow UNIX's shell implementation, only to find out that it was not necessary.

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

In terms of debugging tools, we believe the existing tools taught to us in the labs are sufficient for the scope of this project. We used the debugger in IntelliJ to help us with our debugging. We felt that this was the fastest, most productive way to debug a code due to the wide range of debugging functionalities provided in IntelliJ such as stepping into the code, watching variables in code, setting breakpoints and also by conditions. We do not find any Java debugger Automation Tool to be particularly better than the one provided by IntelliJ. Another debugging method that we find most useful is to trace through the code manually to pinpoint what went wrong. Note that this is not to trace the values, but the logic. This would help us debug logic bugs, and also allow us to better understand our code after the debugging process. For bugs that pertain to values, the IntelliJ debugger and sometimes (although not recommended) a few print statements would do the trick. In addition, manual debugging using IntelliJ has the benefit of allowing us to practice our debugging skills. We also realised that this process tends to be faster when used with past experience in locating the likely source of errors. By knowing where to set the appropriate breakpoints to debug the error, we could speed up the debugging process.

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?

In terms of changing our coding or testing practices, we would consider changing some of our coding or testing practices. We could consider enforcing the categorisation of test cases for all of our test classes. This will enable us to conduct our testing systematically, and allow us to trace bugs by checking through the test cases that passed, using these test cases as a form of checkpoint in the debugging process. This

supplements the manual debugging for logic bugs that we mentioned above, as we could trace through the bugs easier, knowing that some logics have been tested and passed. We could also allocate each category to different developers for debugging. This could further speed up the debugging process as each of us will be more proficient in the type of bugs that we are allocated with.

11. Did you find the static analysis tool (PMD) useful to support the quality of your project?

The static analysis tool (PMD) is useful in helping discover common programming flaws such as unused variables and violation of naming conventions. In the CS4218 module project, a PMD ruleset is provided which has rules already built in based on certain conventions. This helps in meeting our criteria of ease of readability of the source code as well as the maintainability of the source code which we have set and elaborated on in Question 12. The tool allows us to run the analyser to search through the entire source code for any violations of the given rule set/convention, without having to do so manually.

This has the additional benefit of standardising the convention and maximising the code quality across the entire project especially given that the CS4218 module project is being done in a group and individuals may have different coding practices and conventions in which they follow which may or may not involve common programming flaws. The PMD can help standardise the code quality and standards across the entire project even when team members are working on the project at the same time. This minimises the need for individuals to manually check that their coding standards or practices are following the standards. This saves time and as well as cost especially in real world production where time and cost are important criterias to consider.

In addition, with the use of PMD, it will allow other developers reading the source code of the project to have the feeling of consistency in the way the source code is implemented as though it is done by one single individual when in reality it is done by a team. In other words, maximising code quality.

Did it help you to avoid errors or did it distract you from coding well?

As mentioned earlier, it is indeed true that the PMD has helped avoid some programming errors like empty catch blocks and possibly missing break in switch statements. However, being a static analyser, it has its limitations and during our development process in the usage of the PMD there are cases where the PMD act

more of as a distraction especially in the case of lack of flexibility in implementation as PMD checks based on the rule set it was given so anything that violates the rule set is marked as a violation of PMD.

One prominent example is the case of closing of input and/or output streams where the PMD is unable to detect whether the input and/or output streams are indeed closed by the developer through a separate method or a different implementation due to the static nature of the analyser. This results in false positives where input and/or output streams are stated to be not closed and violations are thrown although the implementation itself have ensured that the streams are closed. As shown in this PMD resource link regarding `CloserResource` or streams (https://pmd.github.io/pmd/pmd_rules_java_errorprone.html#closerresource) the PMD checks that the input and/or output streams are closed in the format they specified in the example in the link and even subtle change from the format they have shown such as additional if condition check in the finally block would result in PMD violation. This shows the lack of flexibility of the static analyser due to its static nature. We personally experienced this in our project as during the beginning of the project we spent weeks trying to resolve `CloserResource` PMD violation in the `CallCommand` module etc but was unable to even after clarifying with the lecturers and tutor. After several weeks of trying, we discovered that it is impossible (at least in our view, there might be a way that we might not be aware of) to resolve the PMD violation using the format they have given as it does not allow custom checks such as if the input and/or output streams are `System.in` or `System.out` respectively the streams would not be closed due to the nature of the CS4218 module shell project where the `System.in` and `System.out` needs to remain open to allow the shell to continue accept user input from `System.in` and outputs to `System.out` when necessary which is the nature of shell which is the goal of the project. Without the check, the shell project specification would not be met such as allowing piping of streams in some cases causing bugs in the program. We resolve this violation by suppressing the PMD violation with justification in our report on why we suppressed the violation. This led to a large amount of time and effort wasted which could have been placed in implementation and testing our program. It is due to the above reason we feel that the PMD static analyser has its benefits but at the same time it also distracts us from coding well as its limitations may throw false positives which might give the false impression that our implementation was wrong due to the violation resulting time wasted trying to resolve a potentially non existent problem.

In conclusion, we feel that PMD static analyser is useful in general in avoiding common programming errors ensuring high code quality by checking any violation from the conventions or standard but the violation thrown should be taken as a gauge and not as

an absolute violation meaning the violation need not necessary mean that the coding implementation is wrong as it might be due to its limitation than the implementation hence, not all violations need to be resolved. We feel that the best way to use PMD to improve code quality is to use it to check for violations and any violations should be checked to ensure it is not false positive. Any false positive has to be manually checked by the developer to ensure it is indeed a false positive before suppressing the violation so that the violation would not be thrown in the future allowing us to concentrate on actual violations and resolve them.

Although there is the alternative of modifying the rule set, we believe that it may not be an issue with the rule set but more of a limitation of a static analyser which as the name suggest is unable to dynamically check (hence, static), whether an implementation indeed match coding standards and conventions such as in the case of the streams example given earlier, where it is unable to check that the streams are closed as it does not match their specified format and it may not be feasible to modify the rule set for all possible cases where streams could be closed. The modification of the rule set itself may require time to understand how the rule set is to be crafted and also to ensure it is working correctly which may increase cost in the form of less time to spend on implementation, designing and writing test cases, testing the program and resolving any bugs in the program.

12. 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.

Throughout the whole project, we have used these criterias to evaluate the quality of our project in various phases of the project timeline.

1. Readability

- It is important to ensure high quality documentation and code so that readers and coders are able to understand all of the project documentations and codes with minimal confusion.
- It is also important to make it easier for external users to use our Shell
 - This can be seen during the Hackathon phase where we have to create README to let users set up and use our Shell application quickly and easily with a clear installation guide.

2. Maintainability

- It is to allow future developers to maintain and possibly make improvements to the specification easier. Without maintainability, you will likely have to rewrite the entire application which is very costly.

- Hence, we write shorter and simpler codes to make it easier to test by using the PMD static code analyser and ensuring that there are no PMD violations.
3. Testability
- Each specification or functionality needs to be able to be tested by us easily. With more high-quality tests being created often, we will be able to detect more bugs within our application. This will allow us to provide a more robust Shell application
 - We ensure high testability across the project by
 - Plan test cases based on both the project specification and the behaviour of the Mac Terminal and Windows command prompt.
 - Add more test cases based on the code and code coverage where Jacoco will highlight any lines or branches that have not been tested.
4. Correctness
- It is important to ensure correctness of the entire program as this will save us time and be able to maintain a high quality application throughout the project timeline. This is especially critical if this Shell application is going to be deployed to production.
 - Also, our program code needs to match with a combination of the project specifications, our assumptions, our test cases and the behaviour of the unix prompt.
5. Adaptability
- This is important as the project requirements are ever-changing. So, our application must be able to handle any changes that could be from the project specifications.
 - This can be evidently seen in the module forums and frequently asked questions (FAQ) file where Prof Cristina makes clarifications that are made by other teams.

13. What gives you the most confidence in the quality of your project? Justify your answer.

The high coverage metrics such as the statement and branch coverages, which are required to be maximised based on project specification, gives the most confidence in the quality of the project as the coverage metrics allows us to identify what areas such as statement or branches of the source code are not covered by our testing which we can then improve our testing to maximise the coverages as mentioned in Question 2. By maximising the coverage to the best of our ability, the coverage measurements

provided by tools such as Jacoco report or IntelliJ built in coverage report, allows us to have some form of measurements in the form of percentages or numbers to act as progress indicators when we conduct testing in our project.

However, admittedly the coverage metrics itself does not necessarily mean that the project is of top quality or the test suite are effective as elaborated on earlier in Question 6. It has to be coupled with effective system testing and effective test suite to ensure that the project specifications are met and all possible areas in which bugs might occur such as boundary cases are addressed as coverage metrics which is part of structural testing is unable to test for missing logic faults as structural testing does not focus on code that is not implemented or present. Hence, it has to be coupled with functional based testing such as system testing to ensure that the specifications are met and potential missing logics are addressed.

Despite functional based testing such as system testing also plays a significant role in improving confidence and quality of the project, it does not have a measurable value to gauge and it is highly dependent on what cases were considered during system testing, hence, having a measurable value which coverage metrics provides, gives the most confidence in the quality of the project especially when the value is high, as it provides us a number to rely on.

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

As elaborated in Question 11, PMD being a static analysis tool was initially thought to be useful in ensuring good coding practices however, after using the tool we realised that it was not flexible as it throws many false positives due to its static nature which distracts from coding well. This was unexpected and counter intuitive given that the tool was initially thought to automate checks in code quality minimising the effort to improve code quality and also aid us in standardising our code practices when in fact, our group end up spending more time trying to resolve PMD violations instead.

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

One important reflection on software testing that we have learnt is the different approaches to integration testing (Top down approach, bottom up approach and sandwich approach), specifically the approach that we found most useful in the CS4218 module project in particular and CS4218 module in general is the bottom up approach integration testing.

By analysing how the bottom up approach works, we find it intuitive and we observed that it has the advantage of conducting effective testing of a program without having to unit test every single component in the program which may be very expensive and costly in real world production use where time and cost are important. This is due to the fact that many stubs have to be created especially for modules in the upper layers (not the leaf layers) to isolate the module from the rest of the components in the program to conduct unit testing.

With the bottom up approach, we can first categorise the components of the program to different layers based on their dependencies such as the root, intermediate and leaf layers. The components identified in the leaf layer will be unit tested. The rest of the layers (not including the leaf layers) will undergo integration testing. This method works well assuming the components in the leaf layer are well unit tested. In this way, any bugs found in the components in the upper layer during integration testing are more likely to be bugs in the component undergoing integration testing and less likely to be related to its dependencies as based on the bottom up approach, the lower layers (its dependencies) are assumed to be well tested.

As mentioned, the bottom up approach works well assuming the lower layers are well tested especially the leaf layers which have to be well unit tested to minimise the change of bugs found when testing components in the upper layer not due to its dependencies (lower layers such as the leaf layer). This means that more time has to be spent unit testing the lower layers, especially the leaf layer.

We feel that this bottom up approach which is part of the incremental approach would be very useful when we graduate and start working in companies as testing is an important part of the software development process and understanding this approach can allow for a more efficient and less costly testing process.

Notably, the adoption of this approach depends on the situation as different situations may call for different testing approaches. In the case of bottom up approach, it is more suitable for software where there are less likely to be changes to the lower layers and other approaches can be adopted such as the top down approach where it is suitable for softwares there are less likely to be changes in the upper layer compared to the lower layers. Hence, the adoption of the different approaches depend on the situation and which approach is best suited for the situation.

In the case of the CS4218 module project, we believe it is well suited for the bottom up

approach as the majority of the project specification (besides the operators such as the sequence command, pipe command, etc) involves the lower layers especially the leaf layer such as the specific application commands like rm command. The need to implement these commands to meet the specifications means that when starting the project we attempt to ensure that the application commands are well implemented such that it meets the requirement specification and at the same time most if not all boundary cases are considered. This means that the application commands have to be well unit tested and since the application commands which are in the leaf layer are to be well unit tested, when unit testing of the application commands in the leaf layer are completed, we can move to the upper layers to conduct integration testing using the bottom up approach to minimise the need to create stubs given the limited time constraint of the project while at the same time ensuring that our program is well tested.

It is important to note that to be more accurate, the CS4218 module project approach we adopted was not a fully bottom up approach if we based on bottom up approach definition due to the requirement in the project specification milestone to require unit testing in components that are not necessary in the leaf layer such as the SequenceCommand and PipeCommand modules. Hence, given that upper layer modules such as SequenceCommand and PipeCommand modules are required to be unit tested based on project specification, the approach was more of a hybrid bottom up approach than a purely bottom up approach as some non terminal modules are unit tested not just the terminal modules (modules in the leaf layer).

16. 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 try to suggest an improvement to the project structure which would improve your testing experience in the project?

Suggestion #1: One improvement that can be made to improve testing experience is to either choose an integration tool that allows testing on multiple operating systems (OS) or add an additional continuous integration tool to execute alongside other operating systems (OS) besides Linux .

Limitations: Currently, our Travis is capable to execute tests automatically. However, Travis only works for Linux as Windows is still in beta-mode and Travis in Windows only supports a very limited set of software products. Coincidentally, we make the assumption that our Shell can support Windows platform only. In practice, applications will be developed across multiple OS. It will not make sense to use continuous

integration tool(s) that cannot be tested across multiple platforms.

Benefits: In addition to the benefits of having continuous integration, it would be exponentially better to allow our continuous integration tool to be able to build, compile and execute tests across various popular OS. This will allow our Shell to be platform independent across various OS.

Suggestion #2: One of the improvements can be done is to be clear about the requirements regarding resemblance to the shell implementation. Initially, some of the implementations such as having “/” for folders in shell are thought to be important, however, it was concluded during Hackathon rebuttal as explained in question 9, that it should not be considered a bug as the contents listed is correct.

Limitations: It can be clearer for groups so that we do not have to make unnecessary assumptions and use too much time debugging a “bug” that is not considered a bug in the first place.

Benefits: Being clear in categorising what bug would be considered a “bug” can allow groups to spend more time on correct implementation and testing.