CS4218 Software Testing

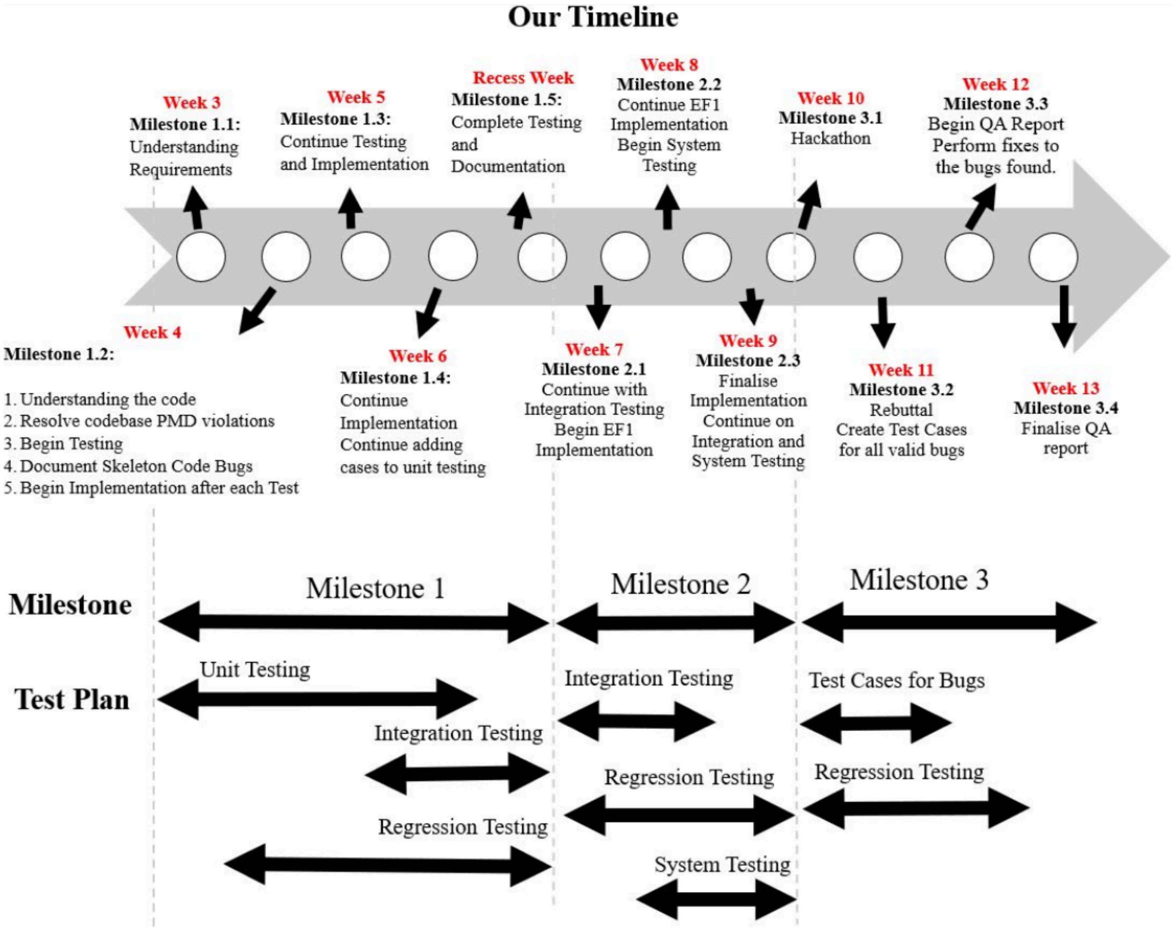
Milestone 3 QA Report

1. Overview

We used the MetricsReloaded Intellij plugin to compute the LOC statistics. Comments and Javadocs are excluded from the analysis. TDD test cases are included in the count under Test Code -> Unit.

	Source Code	Test code	
		Unit (includes TDD)	**Integration & System**
LOC	4686	13086	2132
Proportion	23.5%	65.8%	10.7%

2. Project Life Cycle



Visual timeline of Project Plan/ Activities

Timeline	Activities and Methods
Milestone 1	**Activities:** ● Understand the requirements/specifications ● Before initiating any testing or implementation activities, our team thoroughly reviewed the project requirements. This involved carefully examining the project specifications and experimenting with the commands in the Linux Terminal. Additionally, we sought clarification from both the lecturer and TA to address any ambiguities or uncertainties in the requirements. This initial step was crucial in defining the expected behaviours of the features and setting the foundation for our testing approach. ● Explore the codebase ● Discover faults within existing code ● Requirements-driven development ● Coverage analysis ● Set up testing tools **Methods:** ● Black-box unit testing ● During Milestone 1, our primary testing focus was on unit testing. Following a test-driven development approach, we aimed to create black-box unit test cases before implementing the corresponding features. However, as we progressed with implementation, we gained deeper insights into the features, leading us to generate additional test cases, focusing more on white-box unit test cases. Consequently, unit testing extended into the latter stages of Milestone 1. ● White-box unit testing ● Black-box integration testing ● Towards the end of Milestone 1, we began integration testing. We strategically timed integration testing to align with the implementation of features that required it, such as the sequence command and pipe operator. Test cases for these features were

	developed before their implementation to ensure comprehensive coverage. ● Category partition testing
Milestone 2	● Integration testing ● Given that unit testing had been conducted for all subunits, integration testing enabled us to detect any bugs arising from interactions between units and pinpoint the specific problematic units through subsequent unit testing. ● System testing ● System testing consisted of a blend of manual and automated testing approaches. Our automated test cases were crafted to primarily involve calling commands through the shell module and verifying the accuracy of the corresponding output. We extended this testing to encompass various operators (such as pipe, sequence, etc.) applied to multiple commands, ensuring smooth interaction between different commands at the system level. ● Pairwise testing ● In the context of pipe command: As there are many possible combinations of component interactions to test in integration testing and testing all possibilities are not feasible due to potential combinatorial explosion. As such, we first identified all possible pairs of commands that are reasonable to be connected with a pipe. Considering a pipe command in the form of “app1 | app2”, a pipe command that is worth testing is where app1 writes to standard out and app2 reads from standard in as this ensures there will be interaction between app1 and app2. On the other hand, it is not so meaningful to extensively test commands like as“cd|exit”as there is no interaction between the two apps. Besides the positive cases, we also tested on negative cases to verify that “app2” will not be executed if “app1” throws an exception.
Hackathon & Rebuttal	**Activities:** ● Bug report

	Methods: ● Manual testing ● Automated testing
Milestone 3	**Activities:** ● Debugging and bug fixing ● During Milestone 3, we addressed valid bugs identified during the hackathon and developed test cases to cover each issue thoroughly. While implementing these fixes, we actively conducted regression testing to verify that our previous test cases remained valid and did not fail. If any test cases did fail, we either updated them to reflect the latest changes or reviewed the new code introduced during the bug-fixing process. Once fixes and test cases for all identified bugs were complete, we shifted our focus to preparing the Quality Assurance Report. **Methods:** ● System testing ● Manual testing ● Unit testing

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

The most valuable activity we employed was **coverage analysis**. By examining the coverage results in IntelliJ, we could identify untested parts of our code. This approach allowed us to establish a group-wide standard for the percentage of coverage we aimed to achieve, ensuring comprehensive testing across all aspects of our implementation.

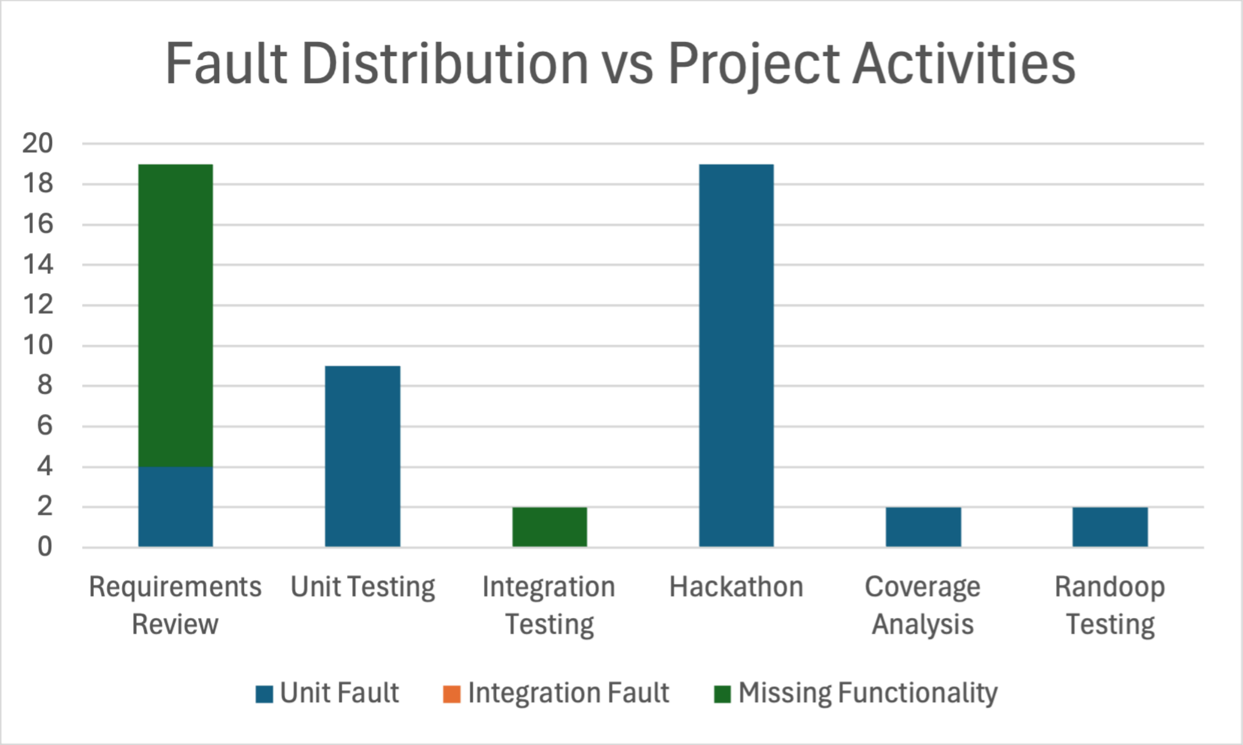
We focused on three key areas: method, branch, and line coverage. Method coverage highlighted untested methods within classes, while branch coverage identified untested branches in conditional statements. Line coverage pinpointed specific sections of code that lacked testing.

High test coverage also helped uncover bugs, particularly in non-happy paths such as exception handling, which are often overlooked during test writing. This process of ensuring comprehensive coverage bolstered our confidence in the quality and robustness of our project.

3. Analysis of Fault Types versus Project Activities

3.1. Faults by Project Activity

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.



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

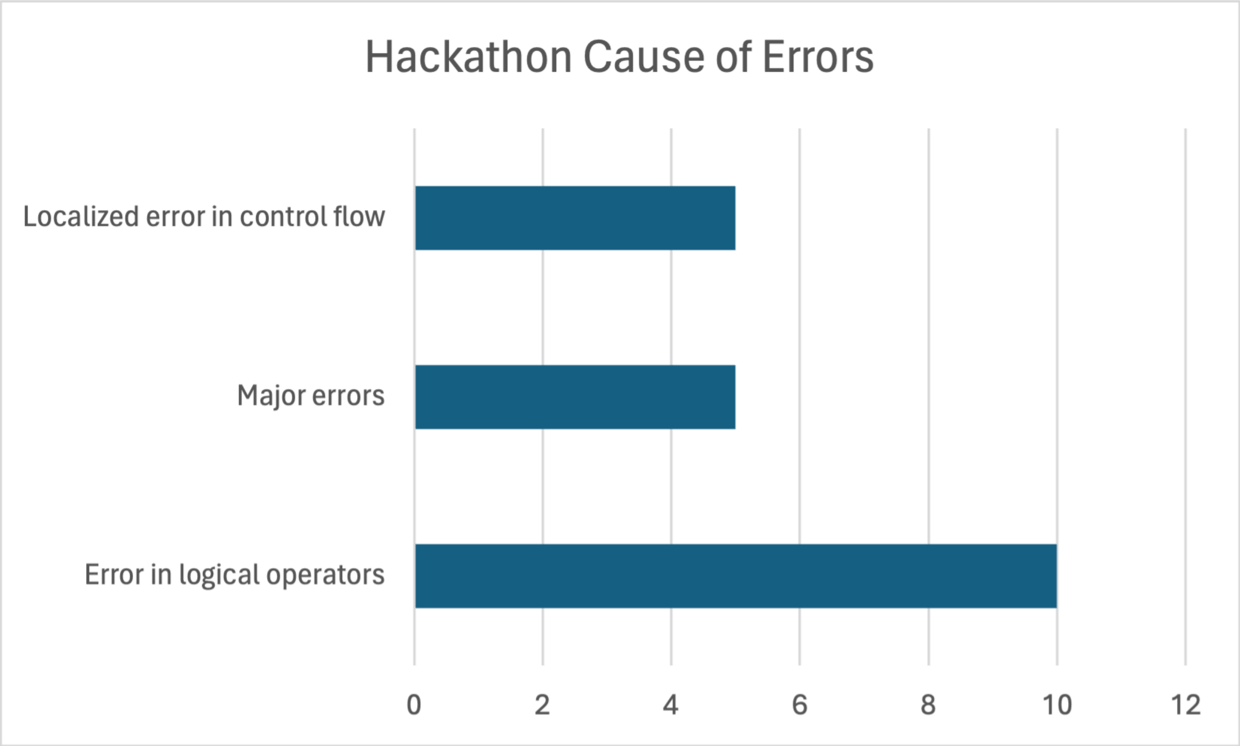
The activity that discovered the most faults were Requirements Review and Hackathon. The distribution of fault type matches our expectations as certain functionality of the application was not implemented yet in the requirements review that we had to implement.

However, the low number of bugs found in integration was unexpected as the bugs found during hackathon was significantly higher than during integration tests.

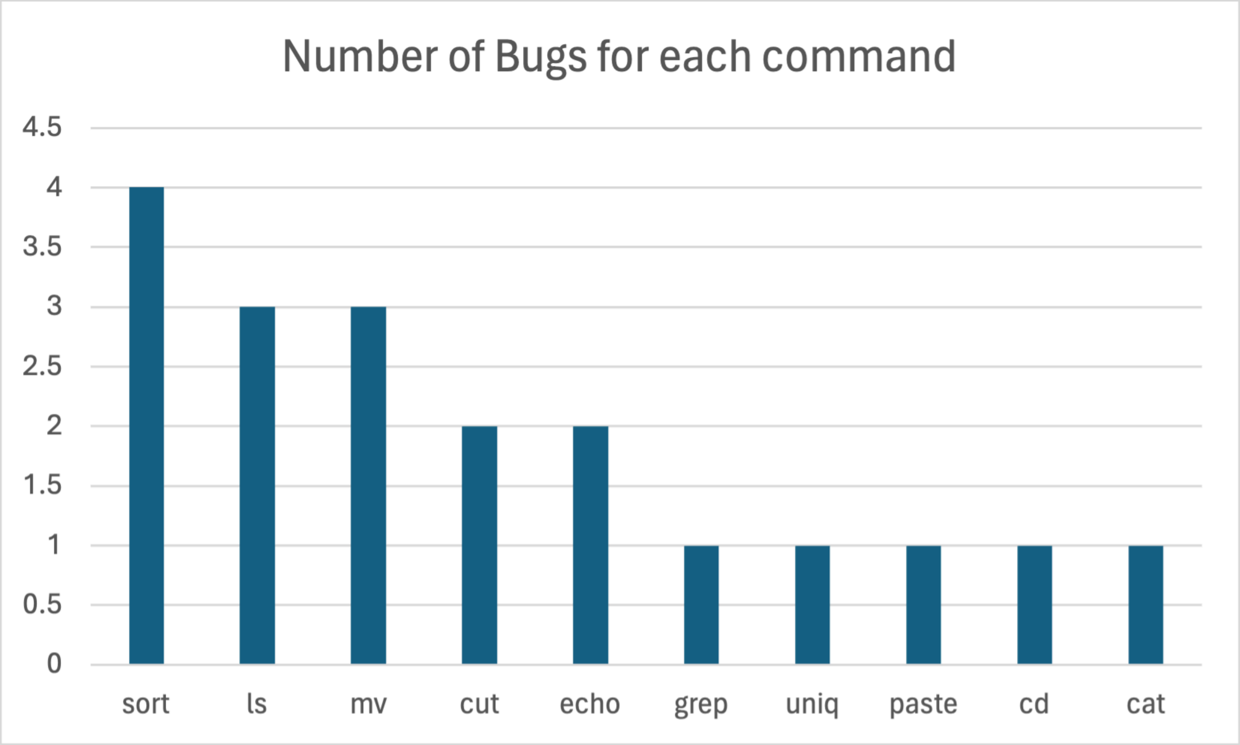
In addition, using coverage %, we were also able to find out many areas of the code that weren't tested and these often turned out to be areas that contained bugs.

3.2 Analyse causes of bugs found in our project during Hackathon

S/N	Command	Description	Cause of Error
1	sort	sort -f gives wrong output on file with alphanumeric characters	Error in logical operators
2	sort	sort -n on file with numbers prepended by spaces gives wrong output	Error in logical operators
3	sort	sort -n gives wrong output on large file	Major errors
4	sort	sort -fr -n gives wrong output on large file	Major errors
5	ls	ls cat/* prints error message	Error in logical operators
6	ls	ls */* crashes shell with unhandled NPE	Major errors
7	ls	ls */ prints error message	Error in logical operators
8	mv	mv a a : moving same file is invalid but does not print error message	Error in logical operators
9	mv	mv a.txt b.txt produces wrong result	Error in logical operators
10	mv	mv a.txt b file to directory produces error message	Error in logical operators
11	cut	cut -c on large file produces wrong results	Localized error in control flow
12	cut	cut -c with ranges -2 2- and the like prints error message	Localized error in control flow
13	echo	echo */. crashes shell with unhandled exception	Major Errors
14	echo	echo hello; prints error message	Localized error in control flow
15	grep	grep "" grep.txt prints error message	Localized error in control flow
16	uniq	uniq -D -d -c alice-bob.txt gives wrong output	Error in logical operators
17	paste	paste - A.txt - < B.txt > ab.txt ab.txt has the wrong contents	Major Errors
18	cd	Incorrect behaviour, cd ./~ does not go to home directory	Error in logical operators
19	cat	cat *.txt does not print contents of non directory files	Error in logical operators



The highest number of bugs came from errors in logical operators, indicating a significant gap in handling expected behaviour based on the requirements. These issues likely arose due to cases not considered during testing. This may have occurred due to misinterpretation of requirements, where certain edge cases were not considered. For example, encountering unexpected error messages during the execution of commands, such as the 'cd' command printing an error message instead of navigating to the home directory when given the argument 'cd./~', shows that this particular argument was not taken into consideration during testing.



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

Yes, it is true that faults tend to accumulate in a few modules. For example, in the hackathon, 10 out of 19 of the bugs were found in 3 classes (sort, ls and mv). It is likely that applications like sort contain complicated control flows, therefore, more faults may be accumulated in those applications. ls contains several globbing bugs (e.g. ls */, lsdir/*), due to the more complicated nature of dealing with nested folders.

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

Yes, we have more logical faults due to incorrect logic implementation, especially for more complicated logics such as recursion flag and globbing in ls (i.e. ls -r, ls */).

4. Time Allocation

Project Component	Requirement analysis and documentation	Coding	Test Development	Test Execution	Others
Time	10%	30%	35%	15%	10%

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, GitHub repository set up, PIT and Randoop for mutation testing.

5. Test Driven Development

Test-Driven Development (TDD) was adopted in our project to detect potential flaws and validate the behaviour of the application early. Tests were written to guide the development of features by specifying expected outcomes. This ensured that we can focus on fulfilling the functional requirements of the project.

For Milestone 1, we wrote test cases for most of the functionalities for the application, however, not all test cases are passing as some functions have not been implemented yet. TDD helps us to keep track of functionality that has not been implemented yet. These test cases help us guide the development of the application, by writing tests for intended behaviour that fails first, the test cases can help to keep track of implementing all the functionalities for a particular command command.

For Milestone 2, we were given test cases that represent the intended behaviour of the application that we can decide if we want to include in our project. These test cases give us an indication of the progress of implementing all the required functionality for the application and continue working on missing functionality for test cases that fail.

By adopting TDD in our project, writing tests upfront allows us to think about the intended functionality that is required to implement for the project which may lead to improved code quality.

As TDD tests individual units of codes, it can help to improve integration, testing and future enhancements as making changes in the code or refactoring existing code may introduce new bugs. However, with the TDD test cases, we will be confident of the changes we make to the application without fear of introducing any new bugs.

In summary:

For **TDD**, the advantages are:

- Improved Test Coverage: We write tests first that need to pass after implementation of the functionality
- Early Bug Detection: Allows us to find bugs early and fix them if the tests do not pass
- Allows us to track our progress of functionality implementation
- Flexibility and confidence in refactoring code

While the disadvantages for **TDD** are:

- Might need more time to complete project as we need to write tests as well as functionality

For **Requirements-driven Development**, the advantages are:

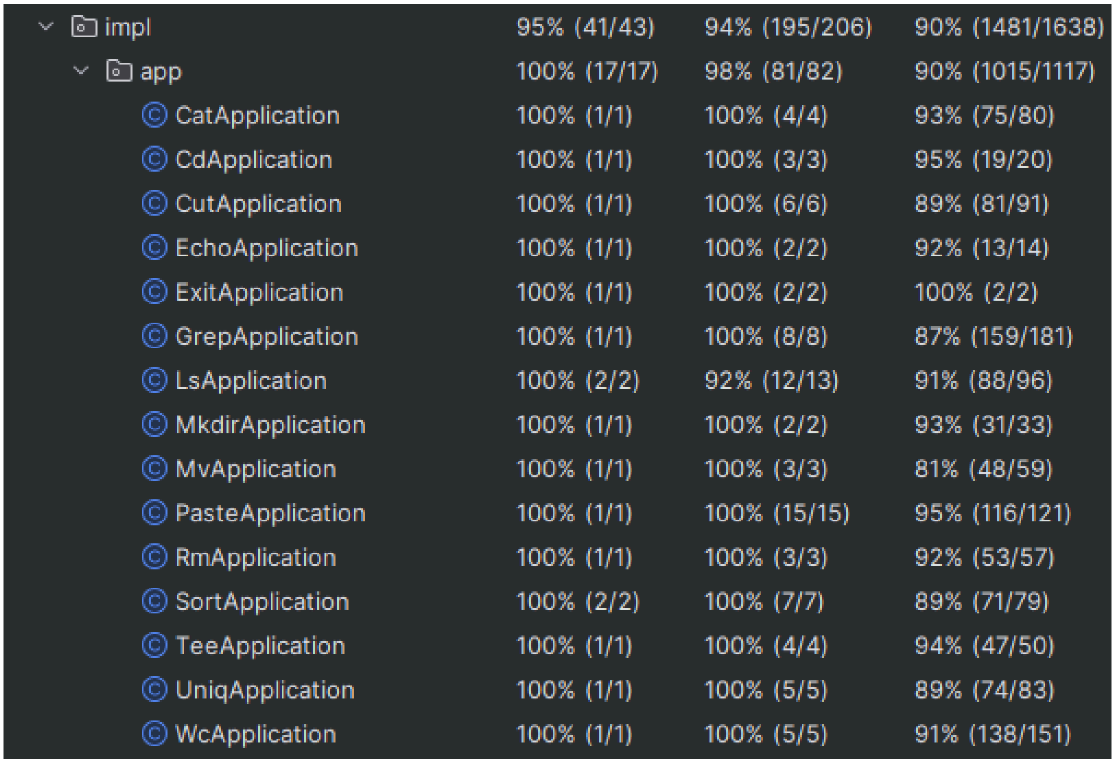
- Ensure that functionality meets requirements
- Less overhead in writing and maintaining tests as compared to TDD

While the disadvantages for **Requirements-driven Development** are:

- Incomplete testing: Might not have full test coverage
- Not much flexibility and confidence when refactoring

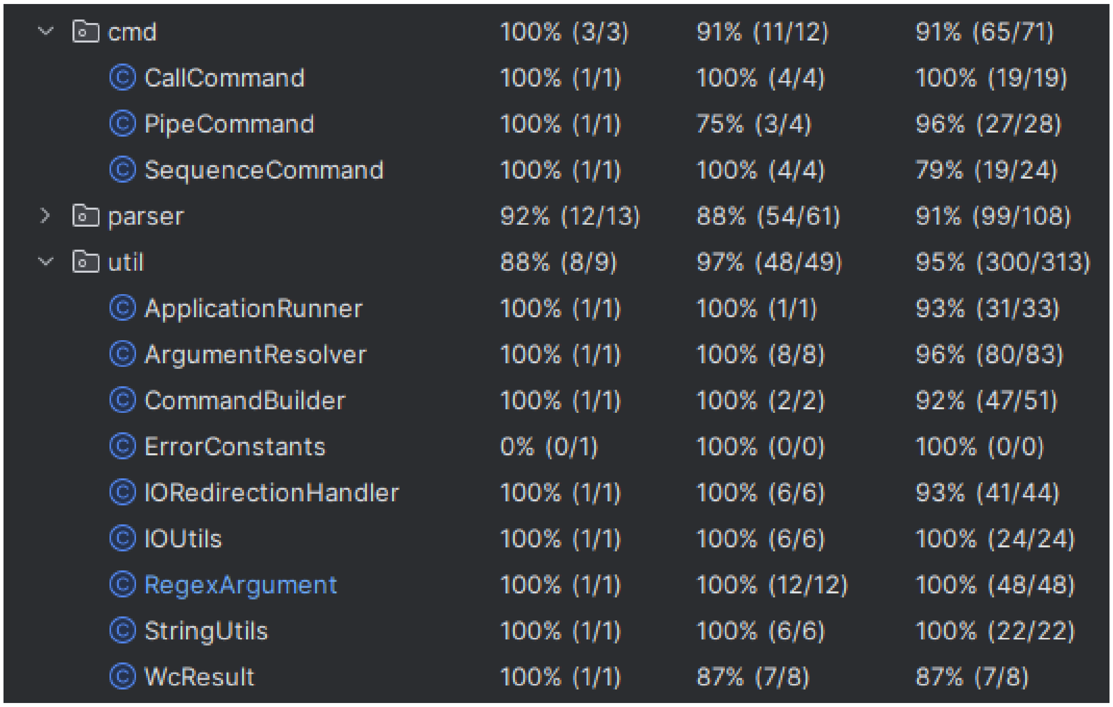
6. Coverage Metrics and Code Quality

Application Code Coverage [Class Coverage %, Method Coverage %, Line Coverage %]:



impl	95% (41/43)	94% (195/206)	90% (1481/1638)
app	100% (17/17)	98% (81/82)	90% (1015/1117)
CatApplication	100% (1/1)	100% (4/4)	93% (75/80)
CdApplication	100% (1/1)	100% (3/3)	95% (19/20)
CutApplication	100% (1/1)	100% (6/6)	89% (81/91)
EchoApplication	100% (1/1)	100% (2/2)	92% (13/14)
ExitApplication	100% (1/1)	100% (2/2)	100% (2/2)
GrepApplication	100% (1/1)	100% (8/8)	87% (159/181)
LsApplication	100% (2/2)	92% (12/13)	91% (88/96)
MkdirApplication	100% (1/1)	100% (2/2)	93% (31/33)
MvApplication	100% (1/1)	100% (3/3)	81% (48/59)
PasteApplication	100% (1/1)	100% (15/15)	95% (116/121)
RmApplication	100% (1/1)	100% (3/3)	92% (53/57)
SortApplication	100% (2/2)	100% (7/7)	89% (71/79)
TeeApplication	100% (1/1)	100% (4/4)	94% (47/50)
UniqApplication	100% (1/1)	100% (5/5)	89% (74/83)
WcApplication	100% (1/1)	100% (5/5)	91% (138/151)

Shell and Utils Code Coverage [Class Coverage %, Method Coverage %, Line Coverage %]:



cmd	100% (3/3)	91% (11/12)	91% (65/71)
CallCommand	100% (1/1)	100% (4/4)	100% (19/19)
PipeCommand	100% (1/1)	75% (3/4)	96% (27/28)
SequenceCommand	100% (1/1)	100% (4/4)	79% (19/24)
parser	92% (12/13)	88% (54/61)	91% (99/108)
util	88% (8/9)	97% (48/49)	95% (300/313)
ApplicationRunner	100% (1/1)	100% (1/1)	93% (31/33)
ArgumentResolver	100% (1/1)	100% (8/8)	96% (80/83)
CommandBuilder	100% (1/1)	100% (2/2)	92% (47/51)
ErrorConstants	0% (0/1)	100% (0/0)	100% (0/0)
IORedirectionHandler	100% (1/1)	100% (6/6)	93% (41/44)
IOUtils	100% (1/1)	100% (6/6)	100% (24/24)
RegexArgument	100% (1/1)	100% (12/12)	100% (48/48)
StringUtils	100% (1/1)	100% (6/6)	100% (22/22)
WcResult	100% (1/1)	87% (7/8)	87% (7/8)

Table that contains issues filed on our github during hackathon:

Application	Issue Count	Actual Line Coverage %
Sort	4	89%
Mv	3	81%
Ls	3	91%
Echo	2	92%
Cut	2	89%
Cat	1	93%
Cd	1	95%
Paste	1	95%
Uniq	1	89%
Grep	1	87%

Based on the table above, we can see that it is not immediately noticeable whether less line coverage results in more issues or bugs found. In general, we noticed that when the line coverage is on the higher percentages (i.e. 85% and higher), it is difficult to predict whether a class with 100% line coverage will have fewer bugs than a class with 90% line coverage. However, during our development and testing phase, we realised that if a class had low coverage (below 50%), then there were definitely more bugs present that we were not aware of, and had to fix once we added more test cases that detected those bugs.

Our opinion is that we do not think that covered class necessarily represents whether a class is of the highest quality, but it serves as a good benchmark of confidence and makes it obvious to detect whether a class needs more test cases or not based on the coverage percentage. When we look at our actual bugs that were filed during the Hackathon, the realisation was many bugs were actually a result of the interaction between multiple units. For example, all 3 ls application bug were actually related to the globbing method (e.g. ls */*), but if we were to look at the argumentResolver and regexArgument line coverage, they are 96% and 100% respectively. Hence, the conclusion is coverage does not mean a class is of the highest quality, but it serves as a good metric to tell when a class is of low quality and not tested sufficiently.

7. Code Design Review

7.1 Unit Tests:

During the initial stage where we implemented unit tests, where each unit is tested independently of each other, this meant that there were two ways to deal with scenarios where one unit component depended on another.

1. Create mocks
2. Ensure minimal dependency between unit classes

Good systems are often designed with the Single Responsibility Principle in mind, where we try to ensure each unit only deals with its own responsibility which also greatly helps to reduce decoupling. During unit tests, we were able to really understand how much responsibility each unit had since we often had to mock or have very long unit tests if the unit itself had a lot of responsibility. Through this in-depth understanding of each unit's responsibility, it affected our code design choices and we designed code that are less dependent on each other and to code with the single responsibility principle in mind.

7.2 Integration Tests:

During the integration testing, it involved testing multiple units together. During the creation of integration tests, we had to understand the interaction patterns between different units, and this prompted us to rethink and refactor our interaction patterns.

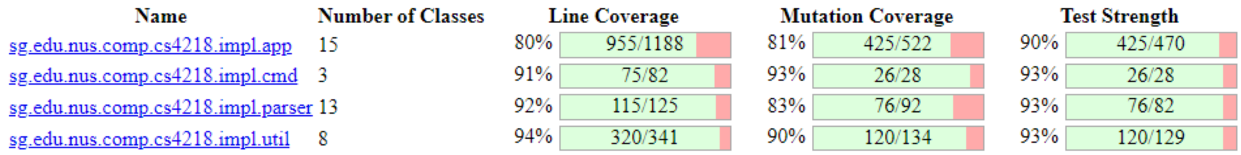
In addition, integration tests helped us discover design problems that occurred between units. For example, during integration tests where we involved multiple units (e.g. cat piped to sort), we realised some units were not able figure out whether to read the input from the inputStream from pipe or the usual way of reading inputs through arguments.

Another design problem was the propagation of errors, where previously we were only handling the errors in each unit, but now due to the integration of multiple units where one unit could call each other, integration tests helped revealed how our errors were propagated through the system, and in some scenarios, we realised certain errors were not handled properly.

8. Automated Testing Tools

In our project, we made use of mutation tests using PIT Test and automated tests generated by Randoop.

8.1 Using PIT Tests:



Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
sg.edu.nus.comp.cs4218.impl.app	15	80%	955/1188	81%	425/522	90%	425/470
sg.edu.nus.comp.cs4218.impl.cmd	3	91%	75/82	93%	26/28	93%	26/28
sg.edu.nus.comp.cs4218.impl.parser	13	92%	115/125	83%	76/92	93%	76/82
sg.edu.nus.comp.cs4218.impl.util	8	94%	320/341	90%	120/134	93%	120/129

Based on the results of PIT Tests, we were able to identify test cases that did not adequately cover the mutations and also the different test coverage of our program. By looking into classes with low mutation coverage, we were able uncover edge cases and conditions that were not covered by our existing tests (bugs/faults) and subsequently fix them. Based on the PIT Test result, we tried to improve and create more test cases that can catch these mutations and iteratively ran the PIT Test to try to improve our scores/results.

PIT Tests were extremely helpful as it did not require any additional effort on our end to find potential mutations. As compared to having to think about what other logic that were not tested in our test cases, which were often something that was difficult to figure out since there are many possible combinations and edge cases.

8.2 Using Randoop:

Randoop is an open-source automated testing tool designed to generate unit tests for Java programs.

One challenge we had was to create enough tests to achieve comprehensive test coverage due to the time constraint given, resulting in some classes being less tested as compared to others. To achieve comprehensive testing, we employed Randoop with the strategic aim of enhancing the testing suite for classes that previously exhibited low coverage metrics. A significant achievement through the use of Randoop was the identification of errors within our code that had eluded traditional testing methods. These errors, once rectified, contributed to the overall integrity and performance of the applications.

The culmination of our efforts with Randoop was a notable increase in test coverage. By addressing undercovered classes and incorporating the insights gained from the generated tests, we significantly expanded the scope of our testing suite.

The testing effort of using randoop to generate tests was also extremely straightforward and easy to use. As compared to the efforts of having to write a single unit tests for a complex application unit that could often involve having to mock many objects, create resource folders and etc, using randoop to generate tests to supplement our less covered applications were much quicker, and can easily scale to creating many test cases for many different units. However, one thing to take note of is, the test cases created by Randoop are often not very readable, which may result in difficulty in debugging the test cases.

Overall, automated tests were a great tool to help supplement our existing manual testing efforts, but they are not a replacement.

9. Hackathon/Rebuttal experience:

Participating in the hackathon and engaging in the rebuttal experience was instrumental in enhancing our project's quality. Testing other teams' projects allowed us to identify faults that were easier to spot from an external viewpoint, prompting us to reflect on and inspect our own code for similar issues. This comparative analysis helped in reinforcing our codebase against common errors. Additionally, observing different coding approaches widened our perspective, enabling us to adopt more robust solutions and refine our methods.

The feedback from other teams during the hackathon highlighted both subjective disagreements based on different assumptions about operational conditions and objectively identifiable deficiencies in our project. This underscored the importance of clear specifications before development and helped us prioritise bug fixes and enhancements, especially in handling edge cases and optimising performance.

10. Debugging experience

A powerful addition could be the use of more advanced static analysis tools, beyond basic linting, to detect deeper issues such as thread safety, memory leaks, and other concurrency issues which are common in systems programming. Tools like FindBugs or Checkstyle can provide more in-depth analysis and help enforce coding standards that prevent error-prone code practices.

For enhancing the debugging experience in our Java shell project, beyond what was achievable with the IntelliJ debugger, several additional tools could be highly beneficial. Firstly, incorporating a tool designed specifically to read and compare output files would have greatly assisted in identifying discrepancies between expected and actual outputs, thereby pinpointing bugs more efficiently. This could involve automated diff tools that highlight differences without manual checking, speeding up the debugging process.

Additionally, integrating Continuous Integration (CI) testing tools would have provided significant benefits. CI systems could automatically run tests on every commit, ensuring that new code integrates smoothly with the existing codebase and that bugs are caught early in the development process. Tools like Jenkins or GitHub Actions could automate these testing workflows, providing immediate feedback on the impact of new changes.

Regarding changes in our coding or testing strategies based on the debugging and challenges encountered, a more coordinated approach would be beneficial. Developing a comprehensive testing plan collectively at the start of the project would ensure that all team members are aligned with the testing objectives and methodologies. Agreeing on common structures and formats for code would facilitate easier integration and less confusion, reducing bugs related to inconsistent coding practices. Additionally, explicitly stating and validating assumptions early in the development process within the team would help avoid misunderstandings and ensure that all functionalities are built on a clear, agreed-upon foundation. This approach would not only streamline development but also enhance the overall robustness and reliability of the shell implementation.

11. Evaluating quality

Compatibility and Portability: Given that the shell aims to emulate UNIX-like behaviour, its ability to run consistently across different Java-supported platforms (like Windows, macOS, and Linux) is essential. Testing in different environments or using a cross-platform compatibility testing tool can help ensure that the shell behaves as expected regardless of the underlying operating system.

Maintainability: This involves assessing how easily the shell can be modified or extended. Factors such as code modularity, adherence to coding standards, and documentation quality are important. Tools that analyse code complexity and adherence to best practices, like SonarQube, can provide quantitative measures of maintainability.

Error Handling: Proper management of command errors and unexpected input is crucial for a reliable shell. Evaluating the quality of the project should also include how well the system handles failures, such as gracefully catching exceptions, providing informative error messages, and recovering from errors without crashing.

12. Counterintuitive answers

The counter-intuitive discovery from the integration testing phase of the CS4218 project was the significance of interaction patterns between different units of code. Initially, one might assume that if each unit works flawlessly on its own, their combination should naturally result in a well-functioning system. However, the integration tests revealed that this is not necessarily the case. The intricacies of how individual units communicate can lead to unforeseen issues, such as improper error propagation or conflicts in data handling. This experience underscored the often overlooked truth that the way components interact can be just as critical as their individual correctness. It taught us that a system's architecture needs careful consideration of the interfaces between units, not just the functionality within them. This revelation was enlightening, as it shifted the focus from just verifying unit behaviour to a broader perspective that also scrutinises the connective tissue of the application.

13. Reflection

Software testing is a crucial component of the software development lifecycle that ensures the quality, functionality, and security of the product before it reaches the end user. Reflecting on the experiences of the CS4218 project and the principles of CS4218 in general can provide valuable insights into the practice and theory of software testing.

a. Reflection on CS4218 Project:
During the CS4218 project, one significant realisation is the essential role of automated testing in maintaining the robustness of software. Automated tests are repeatable, faster to execute, and more reliable in catching regressions than manual testing. They provide a safety net that allows developers to make changes with confidence, knowing that any adverse effects will likely be caught by the test suite. The project reinforced the idea that writing good tests is not just about covering the "happy path" but also about considering edge cases, unexpected inputs, and how the software behaves under failure conditions. Effective testing requires thinking like both a developer and a malicious user, as this dual perspective helps to ensure that the software is not only functionally correct but also resilient against misuse.

b. Reflection on CS4218 in General:
In a broader sense, CS4218 highlights the importance of integrating testing into the earliest stages of software development. Traditionally, testing was often an afterthought, performed after the software had been fully developed. This approach can lead to a higher cost of fixing bugs, as issues discovered late in the development cycle are often more complex to resolve. Learning from CS4218, it becomes clear that a shift towards a test-driven development (TDD) methodology can lead to cleaner designs and more maintainable code. TDD encourages small,

incremental changes, and the tests become documentation that reflects what the code is supposed to do. Moreover, the course underscores the critical understanding that software quality is not merely the absence of bugs but also encompasses performance, usability, accessibility, and user satisfaction. It stresses the importance of non-functional testing, such as performance and load testing, to ensure that the software not only works correctly under normal conditions but also maintains its performance under stress.

In both the project and the broader course, there's a recurring theme: the quality of the test cases is as important as the quality of the software itself. Good test cases are those that offer a high level of coverage, including both white-box (internal structure driven) and black-box (functionality driven) testing approaches, to ensure that every part of the code is examined for correctness. Tests need to be designed to be repeatable, isolatable, and should offer clear indications of failure. The reflection on CS4218 solidifies the understanding that testing is an ongoing process; as software evolves, so too should its tests. This continuous cycle of development and testing ensures that improvements can be made while maintaining the integrity of the existing functionalities.

Lastly, both the project and the CS4218 course emphasise the criticality of test coverage metrics in understanding the effectiveness of a test suite. Throughout the project, the use of coverage tools was not simply a means to achieve a percentage target but a method to identify untested paths and critical logic that could lead to defects. This exposure to coverage analysis instilled the concept that high coverage numbers don't necessarily equate to high-quality testing. Instead, intelligent testing strategies that include boundary testing, equivalence partitioning, and state transition tests, among others, are necessary to ensure that the coverage is meaningful. This approach fosters a deeper comprehension of the codebase and the assurance that the test cases developed are providing the necessary safeguards against future changes, thus enhancing the overall software quality.

14. Suggestions

An improvement to the CS4218 project structure that could enhance the testing experience might involve these key aspects:

Continuous Integration/Continuous Deployment (CI/CD) Workflows: Introducing students to CI/CD principles and having them set up pipelines would simulate an important aspect of industry practices. This would allow students to experience how automated tests are run in a professional environment and understand the importance of quick feedback loops.

Clear and More Detailed Requirements: Adding a bit more clarity to project requirements could further enhance the testing experience. While the somewhat open-ended requirements mirror the realities of the industry where specifications can often be broad, a little more specificity could help in aligning team efforts more effectively. This would allow testers to create more precise test cases and minimise the time spent on interpreting requirements, thus streamlining the development process.

Integration of Comprehensive Bug Tracking Systems: Encouraging the use of robust bug tracking software beyond the basic functionalities of GitHub Issues could simulate real-world software development scenarios more effectively. Tools like Jira, Bugzilla, or Redmine offer detailed tracking capabilities, including time tracking, priority management, and more comprehensive workflow configurations. Utilising these tools within the project could provide practical experience with the systems that are prevalent in the industry, thereby enhancing the learning curve and preparing students for professional software development environments.