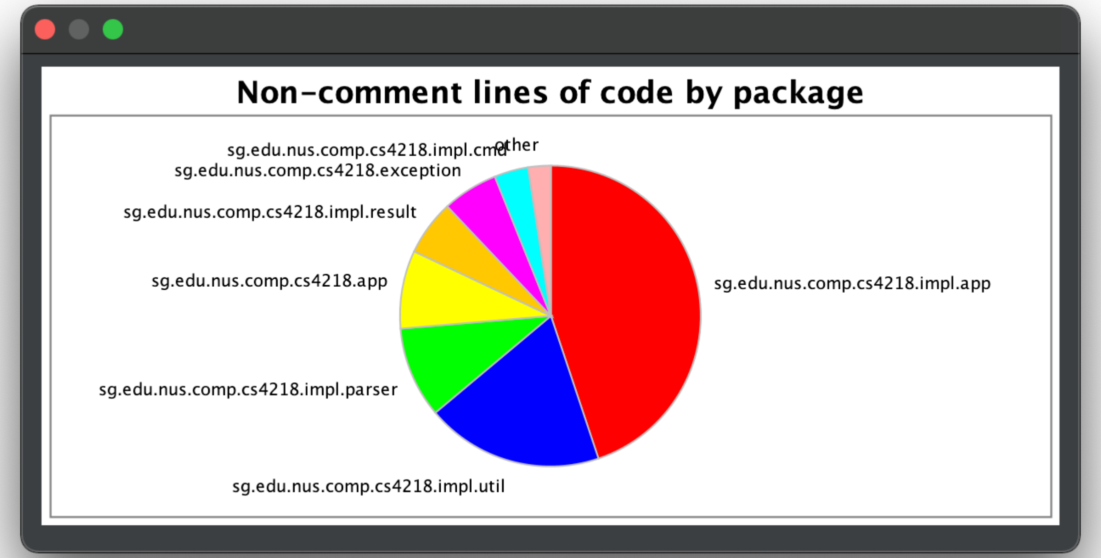
CS4218 Software Testing

Milestone 3

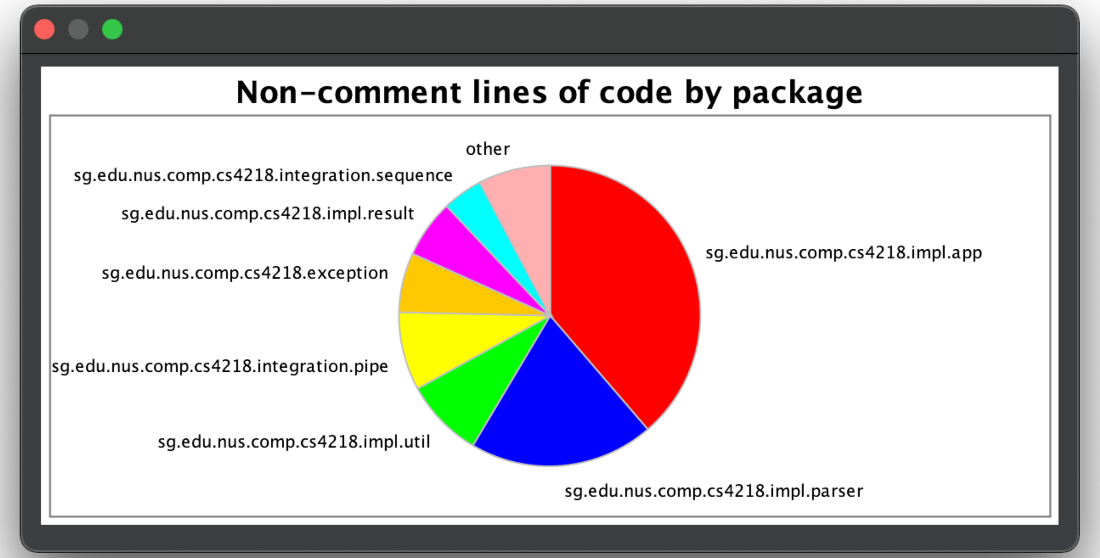
1. Overview

We used the MetricsReloaded Intellij plugin to compute the LOC statistics. Comments, Javadocs and provided TDD test cases are excluded from the analysis.

	Source code	**Test code**	
		Unit	Integration & System
LOC	3939	9392	1813
Proportion	26%	74%	

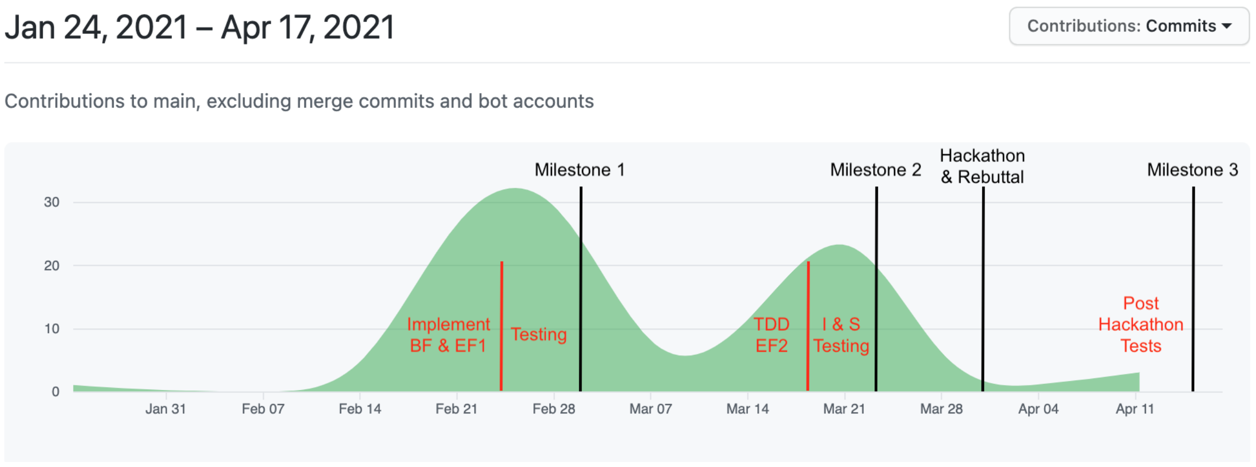


Distribution of source code LOC



Distribution of test code LOC

2. Project Life Cycle



Visual timeline of project plan

Timeline	Methods and activities
Milestone 1	Activities ● Understand the requirements/specifications ● Explore the codebase ● Discover faults within existing code ● Requirements-driven development

	• Coverage analysis • Set up testing tools Methods • Black-box unit testing • White-box unit testing • Black-box integration testing • Category partition testing
Milestone 2	Activities • Test-driven development • Coverage analysis Methods • Integration testing • System testing • Pairwise testing
Hackathon & Rebuttal	Activities • Bug report Methods • Manual testing • Automated testing
Milestone 3	Activities • Debugging and bug fixing Methods • System testing • Manual testing • Unit testing

We used github issues to keep track and coordinate our TODOs - things to implement/test/fix (see https://github.com/nus-cs4218/cs4218-project-ay2021-s2-2021-team12/issues?q=is%3Aissue+is%3Aclosed). The issues would be tagged with the appropriate labels to highlight its nature. Our team members would then each self-assign those issues and this provides a quick and convenient way for us to keep track of each other's progress.

The most useful testing activity is the hackathon, which reveals some edge cases neglected in our implementation. Additionally, we also feel that coverage analysis is rather interesting as we get to visually see which lines/branches of the code are not tested, and we would write test cases to cover them. Seeing the coverage increases makes us happy.

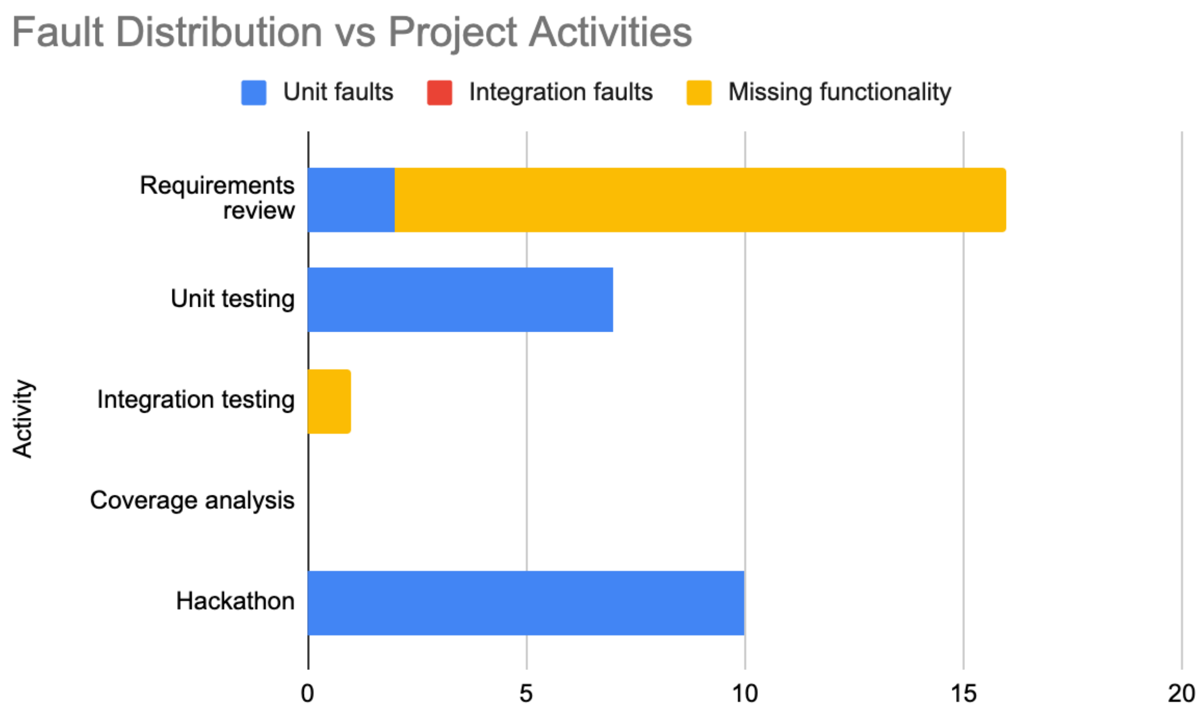
3. Fault Types vs Project Activities Analysis

3.1 Distribution of Faults over Project Activities

The counts are estimates based on our project commit history.

Activity	Distribution of fault
Requirements review	During this stage, we tested the given codebase against the requirements and discovered the following faults present in the initial code via manual testing: **Unit faults: 2** - Shell exits after taking one command (ShellImpl.java) - Call command is never executed (CallCommand.java) **Integration faults: 0** **Missing functionality: 14** - All 14 apps were not working
Unit testing	**Unit faults: 7** - IO error is not handled properly (IOUtils.java) - Errors in StringUtils#isBlank (StringUtils.java) - Errors in handling redirect options (CommandBuilder.java, IORedirectionHandler.java) - Errors in quote unwrapping (ArgumentResolver.java) - Errors in pipe command (PipeCommand.java) - Errors in mv app (MvApplication.java) - Errors in paste app (PasteApplication.java) Unit testing helped discover a lot of unit faults in the original codebase as well as code written by us. **Integration faults: 0** **Missing functionality: 0** Relevant GitHub PRs: - https://github.com/nus-cs4218/cs4218-project-ay2021-s2-2021-team12/commit/f1f1df67c529c95a31f4133bc7aec7f650fe8134 - https://github.com/nus-cs4218/cs4218-project-ay2021-s2-2021-team12/commit/c1384383dfdc884983b0399c55802f261a5becb5 - https://github.com/nus-cs4218/cs4218-project-ay2021-s2-2021-team12/commit/ccedfcf7be3a2af0cbf5f4383b26ca195db82a35 - https://github.com/nus-cs4218/cs4218-project-ay2021-s2-2021-team12/commit/641c6466d34609e677a39756c9d0001f63bce292 - https://github.com/nus-cs4218/cs4218-project-ay2021-s2-2021-team12/commit/fd1efcb07cc2ef481f4ae3a7e368719bf7abce75 - https://github.com/nus-cs4218/cs4218-project-ay2021-s2-2021-team12/commit/c1c9bee77a79fa91321c11c0645475186f1f83f1
Integration testing	**Unit faults: 0** We did not discover any unit fault during integration testing, this is probably because we have done thorough unit testing before integration testing. It is unlikely that we would discover additional unit faults missing from the unit tests. **Integration faults: 0** **Missing functionality: 1**

	- Missing 'paste' app to the ApplicationRunner
Coverage analysis	**Unit faults: 0** **Integration faults: 0** **Missing functionality: 0**
Hackathon	**Unit faults: 10** **Integration faults: 0** **Missing functionality: 0**



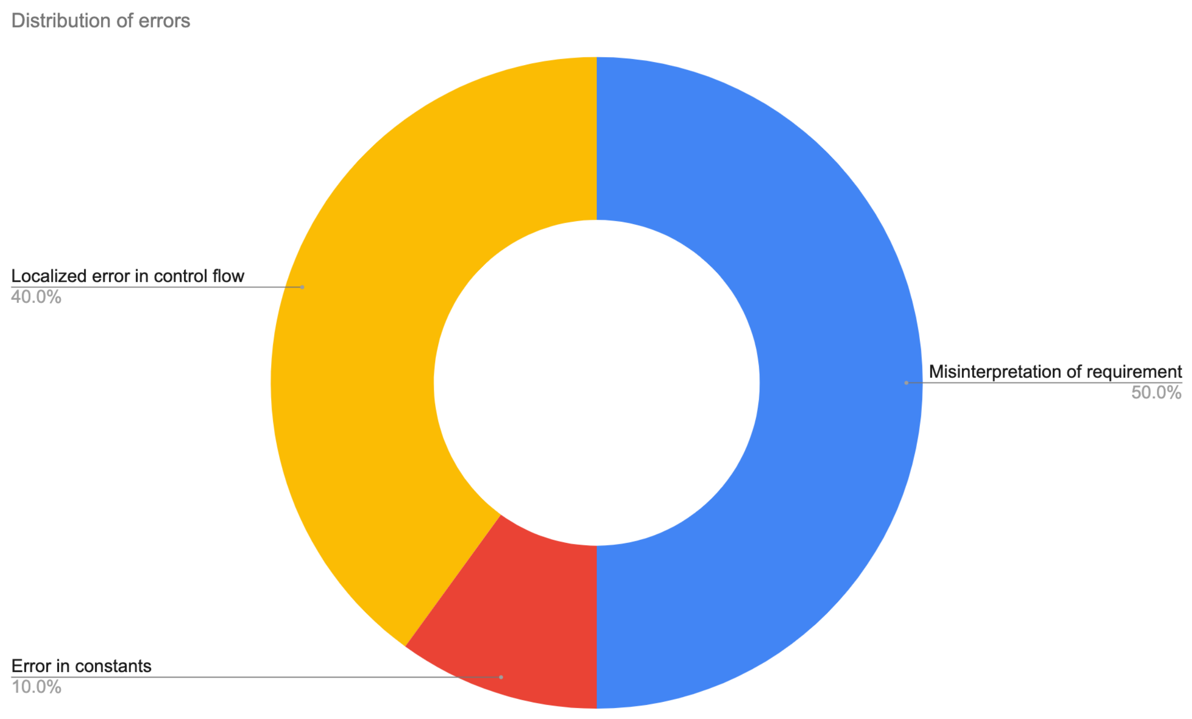
Fault Distribution vs Project Activities

Unit testing and hackathon discovered the most faults. The distribution of fault types largely matches our expectations. During requirements review, we discovered many missing functionalities. For unit testing, since our test cases were quite comprehensive and quite a number of bugs were discovered in the early stage during unit testing, only a few bugs were discovered during integration testing. Similarly, for coverage analysis, the major faults have already been sieved out by the existing tests, instead it only helped us improve overall test coverage. During hackathon, a lot more unit faults were discovered. This is mainly because we misinterpreted some requirements and missed some edge cases.

3.2 Hackathon Bug Analysis

S/N	Bug Report Number	Brief Description	Cause of error
1	from16_2	*cat* a single line file without trailing newline character should not print an additional newline.	Misinterpretation of requirement
2	from16_4	An error message should be shown when copying a folder to another folder without the recursive flag.	Localized error in control flow
3	from16_7	Listing non-folders should not show any error messages.	Misinterpretation of requirement
4	from16_10	When a folder (source folder) is moved to another folder (destination folder) and within the destination folder, there is a folder with the same name as the source folder, this folder should be overwritten.	Localized error in control flow
5	from16_11	When moving multiple files, if an exception is thrown when moving one of the files, the operation should continue.	Misinterpretation of requirement
6	from16_13	Deleting a non-empty directory with -d flag should display the correct error message.	Error in constants
7	from16_14	When deleting multiple files, if an exception is thrown when deleting one of the files, the operation should continue.	Misinterpretation of requirement
8	from16_19	*wc* a single line file without trailing newline character should report 0 line.	Localized error in control flow
9	from16_20	Trailing newline should not be added during file/input redirection.	Misinterpretation of requirement
10	from18_1	Duplicate of #1	-
11	from18_3	*echo hello;* should be executed successfully without error	Localized error in control flow
12	from18_6	Duplicate of #2	-

It is true that faults tend to accumulate in a few modules. For example, bug #5 and bug #7 are from different modules, but the input case is quite similar, i.e. a sequence of files with some invalid files. Since they originate from the same type of fault, the same fix can be applied to address bugs.



Distribution of errors

As seen from the chart above, the most predominant class of faults is misinterpretation of requirement. As such, this result reinforces the idea that having a full understanding of the requirements is crucial as any misinterpretation could result in bugs (unintended behaviour) in the program. Overtime, as severity or number of the misinterpretations accumulates, the number of unintended behaviours would also further increase.

4. Time Estimates on Project Components

Project Component	Requirement analysis and documentation	Coding	Test development	Test execution	Others (learn/set up testing tools)
Time	15%	30%	40%	7.5%	7.5%

5. Requirement-Driven Development vs Test-Driven Development

	Advantages	**Disadvantages**
RDD	In RDD, the functionalities are coded and implemented directly following the specifications. This means that following RDD, the required features can be implemented in the quickest time	For RDD, the code is written before the test cases. As a result, the initial faults in the code could accumulate and become much harder to fix in the later stage of development. Drastic changes might be required if the bug stems from

	possible. As such, implementation-wise, RDD is more time efficient as compared to TDD. For instance, in milestone 1, we managed to finish implementing BF and EF1 in around 3-4 days. Thereafter, we started on writing the test cases for those functionalities. With RDD, code may be better-designed as developers may aim to cover requirements comprehensively instead of just trying to pass all test cases in TDD. Additionally, we feel that RDD is more suited while integrating the different components. During integration, we find that it is better to finish writing the integration code first rather than writing the code bit by bit, trying to progressively pass the integration tests. This is because sometimes when an integration test fails, there are much more factors involved that could have caused the failure as compared to when a unit test fails. As such, development progress might be too slow if we try to do TDD and hence RDD is preferred.	fundamental issues in the architecture. For large scale projects, RDD could be less efficient than TDD as a lot of time might be spent on tracing and bug fixing. With RDD, code may take longer to write and have to be rewritten several times as developers may not have a clear and comprehensive idea of the expected behaviour while writing code before test cases.
TDD	Following TDD, as long as there are any tests that fail, we would immediately re-write the implementation code in order to pass the failing tests. This helps us to sieve out the potential bugs as early as possible and prevent the bugs from accumulating and becoming much harder to fix at the later stages of the project. Since we repeatedly re-run the test cases during each iteration, we have more confidence that the changes in each iteration do not lead to regression or break the existing functionality. Even when there is regression error, it is also much easier to locate the faults and debug using the failing TDD test cases. Moreover, the test cases are very helpful during code refactoring as well.	Since we write just enough code to pass the test cases, it is extremely important for the test cases to be comprehensive. Otherwise, some requirements or edge cases might be neglected during the development process. Passing all tests may bring a false sense of correctness as the tests reflect the developers' possible misinterpretation of requirements. The TDD process is slow and tedious due to its iterative workflow. Before starting to write actual code, a significant amount of time is needed to set up the test environment and develop the test cases. If there are changes in the requirements, the corresponding TDD test cases need to be updated as well. For example, in milestone 2, several interfaces were updated. Some test cases written in milestone 1 were not applicable anymore. As such, a significant amount of time and effort could be required to constantly update and maintain the TDD test cases. This could be detrimental to the progress of the

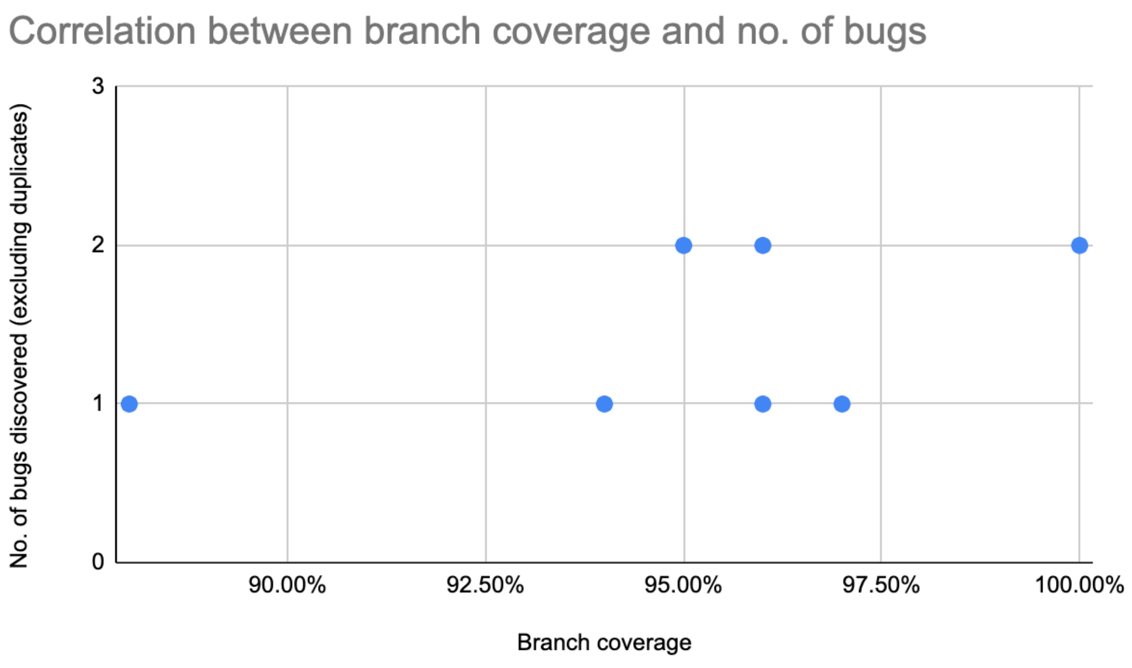
		project if the requirements are frequently changed.

6. Coverage Metrics vs Bugs Found

Interestingly, coverage metrics do not really correlate with the bugs found in our code during hackathon. For the classes related to the bugs discovered, all of them have rather high code coverage. For example, for MvApplication, although it has 100% line coverage and code coverage, there are two bugs present.

Class	Line coverage	Branch coverage	No. of bugs discovered (excluding duplicates)
MvApplication	100%	100%	2
LsApplication	95%	97%	1
WcApplication	92%	96%	2
CommandBuilderUtil	97%	96%	1
RmApplication	98%	95%	2
CatApplication	91%	94%	1
CpApplication	94%	88%	1

Note: The statistics above is taken from the code before bug fixes.



Correlation between branch coverage and no. of bugs

For some bugs discovered in hackathon (e.g. bugs for *rm*, *mv* applications), we did consider similar input cases in our test cases. However, due to the differences in the interpretation of the requirements, the test cases themselves were wrong in the first place. In this case, such bugs are not correlated with code coverage.

For some other edge cases highlighted by the tester teams in hackathon, we observed that the relevant lines/branches were actually covered by some other test cases. For example, when *wc* a single line file with no trailing newline character, the branches taken are no different from the branches taken if we *wc* a single file with trailing newline character. In this case, such bugs cannot not be surfaced even if we have some test cases that cover the relevant lines/branches.

We achieved 100% branch coverage for classes in the following packages:

- sg.edu.nus.comp.cs4218.impl.cmd
- sg.edu.nus.comp.cs4218.impl.parser
- sg.edu.nus.comp.cs4218.impl.result
- sg.edu.nus.comp.cs4218.impl.exception
- sg.edu.nus.comp.cs4218

In addition, we achieved 100% branch coverage for

- CdApplication
- EchoApplication
- ExitApplication
- IOUtils
- StringUtils
- CollectionUtils

The least covered classes are:

- TeeApplication (85%)
- RegexArgument (88%)
- CpApplication (90%)
- SplitApplication (90%)
- ArgumentResolverUtil (91%)
- IORedirectionHandler (93%)
- ApplicationRunner (93%)
- CatApplication (94%)

In terms of code, the classes that achieved the most branch coverage are usually less complex compared to the least covered applications.

Generally the most covered classes are more likely to be of better quality compared to the less covered classes as the uncovered lines/branches in those less covered classes might contain faults. However, this is not always the case. Sometimes the most covered classes may not be of the highest quality. For example, a class with 100% code coverage might have faults like missing functionalities and such faults can only be revealed by functional testing.

7. Code Design Review

Fortunately for our team, at the start, we did foresee some of the problems that might arise when performing unit and integration testing with the current skeleton code architecture. In particular, each of the app classes is very large and handles multiple different logic. This is quite bad for testing as we would be testing various different logic on a single class and since the methods are long and convoluted, this would hinder the isolation of bugs and could possibly make it difficult for us to identify the cause/root of the bugs.

As such, for milestone 1, we had already begun to refactor and redesign the architecture to make it more modular. For instance, we have split each app into 3 distinct components, one for the app logic which is handled by the main app class, one for the parsing of app-specific input format which managed by the parser class, and the last one is for encapsulating and formatting the app result which is handled by the result class. Hence, with this new improved architecture, each component is in charge of a specific functionality and unit testing can be effectively and rigorously carried out to test a single functionality on each of the individual components.

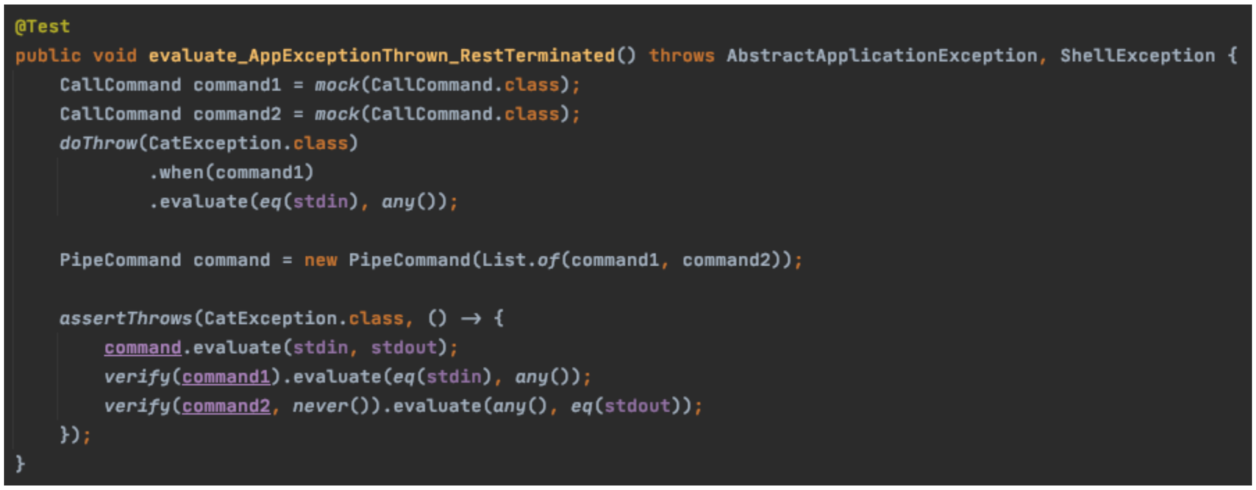
Additionally, since these components are very modular, if there are any new bugs found from integration tests but our rigorous and highly-specific unit tests revealed no bugs, then there is a high chance that the bugs originated from the integration logic instead of from the individual component logic. In a way, our improved architecture helps us to better isolate and identify the cause of bugs.

8. Testing Tool Review

Our team did not use any automated test generation tool. Instead, we have explored other tools that have facilitated us in producing quality test cases.

8.1 Mockito

- Mockito greatly simplifies the process of creating stubs and makes some complicated test cases more human-readable.
- Below is an example from PipeCommandTest.java. We would like to test that PipeCommand stops executing the remaining commands when an exception is thrown in the currently executing command.



```
@Test
public void evaluate_AppExceptionThrown_RestTerminated() throws AbstractApplicationException, ShellException {
    CallCommand command1 = mock(CallCommand.class);
    CallCommand command2 = mock(CallCommand.class);
    doThrow(CatException.class)
            .when(command1)
            .evaluate(eq(stdin), any());

    PipeCommand command = new PipeCommand(List.of(command1, command2));

    assertThrows(CatException.class, () → {
        command.evaluate(stdin, stdout);
        verify(command1).evaluate(eq(stdin), any());
        verify(command2, never()).evaluate(any(), eq(stdout));
    });
}
```

- In order to test this behaviour independent of the implementation of various CallCommand, without Mockito, we would have to manually create a CallCommand stub class which is only used once in this method. That means, it is possible that we have to create several stubs of the same class to fulfil the need of different test cases.
- With Mockito, we can use methods like mock(), doThrow() to define a stub with simple behavior with only two lines. We can use verify() to check whether the expected method calls are made.
- We have also used other Mockito features like spies and argument capturing in other test cases of the project.

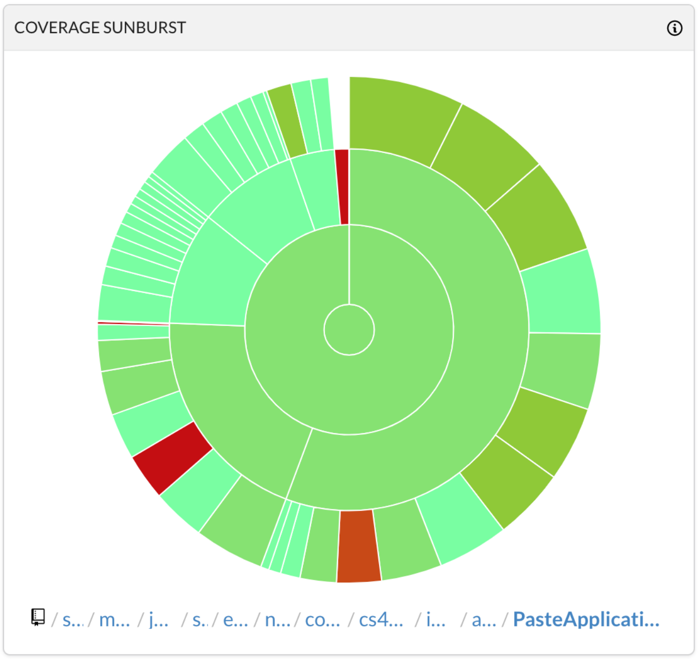
8.2 Gradle Build Scan

- Using Gradle, we have automated the PMD static code analyzer to run during tests or builds. As such, whenever there are any PMD warnings, tests/builds will fail and we will be alerted.
- Additionally, we have included Gradle Build Scan that will automatically generate a test report to provide insights to our testing (see https://gradle.com/s/b5vkl4yzrdyyc). This has proven to be extremely useful as some of our team members have sometimes relied on the test report to locate bugs.
- In particular, we have successfully identified the following bug using this tool:

Report	Bug	Cause	Buggy code fragment	Fix
Link	Test cases on listing files/directories with ls passed when run locally. However, the tests **sometimes** failed when run on other/remote platforms (E.g. Travis).	The build scan report revealed that the order of the listed files is not always deterministic. When run locally, the listed files are of a particular order and thus it passes the test cases. Yet, when run on Travis, a different order is output and thus the test cases failed.	See	Output files are sorted in lexicographical order to improve testability. See

8.3 Codecov + Jacoco

- Finally, our team has also included Codecov for a deeper analysis of our code coverage. The code coverage provided by this platform is much more comprehensive (based on Jacoco) than the one provided in Intellij. See https://app.codecov.io/gh/nus-cs4218/cs4218-project-ay2021-s2-2021-team12 (may need admin access to cs4218 repo).
- Codecov's coverage graph helped to facilitate structural testing as it highlights the coverage density of our program. This makes it easier for us to identify specific portions of the codebase which needs more coverage. Without this visualization, it is difficult and almost impossible for us to identify the classes which are lacking in coverage.



Codecov's Coverage Graph

9. Hackathon Experience

The test cases generated by the other team can be mainly grouped into 2 categories: (1) difference in assumptions and (2) edge cases.

For the test cases related to difference in assumptions, they are the result of how the different teams resolve ambiguity in specifications. For our team, whenever there is ambiguity in the project specifications, our team would fallback to the default Unix specifications for reference and implement accordingly. For the tester teams, they stated their own assumptions and hence they tested our implementations according to their assumptions. As such, this situation highlights the importance of clarity and agreement in the project specifications.

For the edge test cases, we were really impressed that our tester teams were able to come up with those edge cases. In particular, there is one edge case in our program where if we tried to *wc* a file with only 1 line but without trailing newline character, our app would output 1 for the number of lines, but the actual Unix behaviour is 0 for the number of lines (which is not intuitive). Indeed, such test cases generated by the other teams have helped to improve/correct our code logic.

10. Debugging Experience

It would be useful to have an automated slicing tool. With this, locating faulty statements when a bug is found would be faster, as the corresponding slice can be inspected to find the reason for the unexpected values.

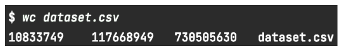
It would also be useful to have a debugger that can examine local files operated on by the program, as many of the commands and bugs involve files as inputs/outputs. If this was possible, fixing these kinds of bugs would be easier and faster as there would be no need to manually examine the files in an external program, similar to how using a debugger would be easier and faster than printing variable values to the console.

From the project, we have experienced that having a good software architecture plan at the start helps to improve code quality as well as testability. This could greatly reduce the number of potential bugs as well as make debugging much easier. Additionally, we found that continuous integration tools, such as Travis CI, are very useful in ensuring that any code we have pushed to GitHub does not introduce new/regression bugs. That is it helps us to automatically check that existing behaviour still works as expected in the process of implementing new features or fixing bugs.

Regarding the use of debugging tools, our team only utilized the IntelliJ debugger to locate the bugs as it already provides quite a number of useful features.

11. Project Quality Evaluation

Criteria	Modularity	Extensibility	Performance
Description	How complex is each class? How coupled/decoupled are the different classes?	How easy is it to add a new functionality/app to the existing project?	Is the shell app able to handle large file inputs and output the result in a reasonable amount of time?

Evaluation	We designed our classes and methods such that they are not overly complex. In particular, we tried to adhere to the Single Responsibility Principle (SRP) as this promotes modularity of the components and facilitates easier testing.
	
	E.g. we have the *Parser classes that handles input parsing for the different apps as well as the *Result classes that encapsulate and format the apps' outputs. This allows us to better test our logic as well as isolate potential bugs.
	
	Additionally, we also check and avoid cyclic dependencies among classes as this hinders testing as well.
		Our project utilizes a component-based architecture and due to its modular design, we can easily add a new functionality/app without modifying much of the existing source code.
		
		E.g. if we were to add a *touch* app, we only need to modify ApplicationRunner.java to add a new case. The code for the new app can be easily integrated (plug-and-play) into the existing shell app.
			When tested against a file with 1k lines, the app is able to output the result within 1 second.
			
			Stress-test: The app is able to handle extremely large file size (≈10 million lines/≈730MB). E.g. the app does not crash and is able to output the result within 20 seconds (relatively reasonable considering the large file size).
			
			
			`$ wc dataset.csv`
			`10833749    117668949    730505630    dataset.csv`

12. Counter-Intuitive Takeaway

One counter-intuitive takeaway we have is that code coverage for a particular class does not necessarily correlate with the number of bugs found in that class. While it is true that having good code coverage is desirable and it could help to sieve out bugs related to constants, arithmetic and control flow errors, it does not necessarily mean that the program will have no bugs when we have 100% branch/line coverage. Indeed we have learnt in lecture that structural testing cannot reveal functionality bugs, but seeing this claim being true physically in our code has further convinced us that we should not only rely on coverage analysis to conclude on correctness of the program. Instead, we should also manually examine the implementation code as well as the test cases to see if indeed reflect accurately the specification of the program.

13. Reflection

Through this module and the project, we have learnt that software testing is critical to any (especially large-scale) software project as it gives confidence in the correctness and efficiency of the software. Most of the time, when we were engaging in our own software projects, we did not really give much thought to testing as it is usually more enjoyable to churn out features as compared to writing test cases. Not to mention, for certain frameworks/languages, setting up of the test environment is also quite a hassle and time-consuming as well. However, what we did not realize was that once we have endured the initial pain of producing quality test cases, testing actually brings a lot of benefit.

In the case of the project, we were given a partially implemented app, and we were supposed to complete implementing the full specifications. The main issue with such brownfield projects is that it takes a while for us, the new developers, to fully grasp the inner workings of the codebase and during that period, we are all afraid/not confident enough to make rapid/abrupt changes to the source code. In particular, we were afraid that our modifications would result in breaking changes to the existing functionalities and this would hinder our progress as when we thought we had fixed one part of the app, another part of the app became broken.

For a start, we did manual testing for the initial phase of milestone 1. That is whenever we make changes to the source code, we would manually test via the shell some sample inputs to see if the existing features are still working and the bug is fixed. However, this quickly became unscalable and not comprehensive enough as the number of general cases as well as edge cases increases when more functionalities are implemented. This is where we decided to switch to automated testing and start to write quality test cases to verify the implemented functionalities with the specifications. While our progress in functionality implementation slowed down as we were investing the time in writing test cases, the benefits of testing eventually outweighs its initial cost. In particular, once we have our comprehensive test suites, we were actually able to implement and, specifically, refactor and improve our existing design/codebase at a much faster pace and with ease. For instance, every time when we make some changes to the code, with a simple press of a button, we can immediately know if there are any regression bugs that have occurred. This gives a huge confidence boost to us, developers, while coding as well as gives confidence in our app.

Additionally, we have also learnt that it is important to balance between the costs of implementation and testing. From the project, we have found out that there is a strong correlation between quality of test suites and its usefulness. That is if the test suites are very comprehensive, then naturally it would cover more edge cases and bugs are more easily revealed. Likewise, if the quality of test suites is poor, then they may not be useful in revealing bugs. However, producing quality, comprehensive and high coverage test suites is not trivial, and it requires a lot of effort and resources. Not to mention, in the industry, software requirements are constantly changing to match the client's needs and hence even more effort is required to maintain and update those test suites. As such, ultimately, we need to evaluate carefully both the costs of implementation and testing, and find a good balance point between the 2 so that not only can we meet the software specifications but also achieve a decent level of testing and confidence in our software. Also, one important skill we have learnt that could save some costs in implementation is to fully understand and, if need be, clarify the software specifications. This ensures that what we implement is exactly what is expected and helps us to avoid implementing features which are out of scope or not intended.

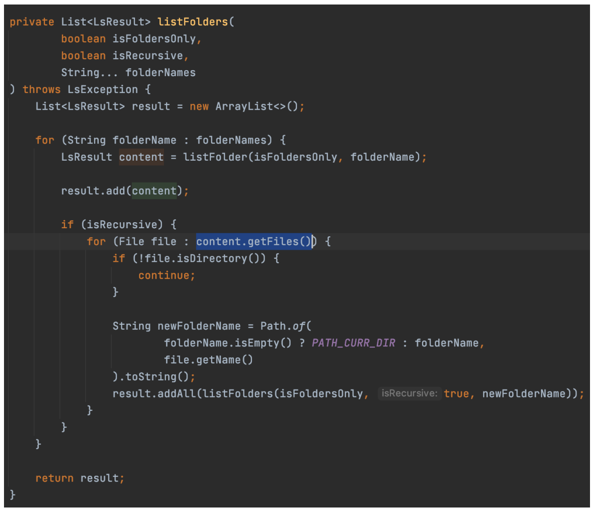
14. Suggestion

To keep up with the rising trend in apps with GUI, perhaps this project can be extended to include UI testing as well. To keep the workload reasonable, we propose that the project can cut down on the number of apps to implement and instead shift some focus to implementing and testing a GUI. In particular, we feel that the implementations and testing for some of the apps are rather similar and are quite repetitive. E.g. *cat*, *wc*, *grep* and *tee* have similar IO formats and only differ much in the computational logic, which is not the main focus of this module. As such, implementing one or two of such apps should suffice in our learning/evaluating of the relevant testing techniques. The remaining time can then be spent on the UI component.

One example of how this project can be modified is that we can make it a 2-tier architecture (frontend-backend). The frontend can be JavaFX or any other frontend frameworks while the backend remains as the shell app. The GUI/frontend can have simple features such as "Run" button, retrieval of previous command or view command history. More complicated yet interesting features such as drag and drop a local file onto the command box to auto-fill a file path can also be considered.

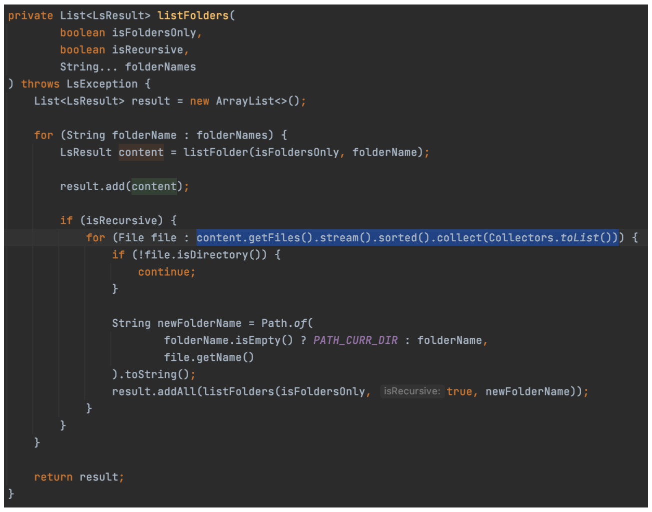
15. Appendix

Buggy Code Fragment and Fix



```java
private List<LsResult> listFolders(
        boolean isFoldersOnly,
        boolean isRecursive,
        String... folderNames
) throws LsException {
    List<LsResult> result = new ArrayList<>();

    for (String folderName : folderNames) {
        LsResult content = listFolder(isFoldersOnly, folderName);

        result.add(content);

        if (isRecursive) {
            for (File file : content.getFiles()) {
                if (!file.isDirectory()) {
                    continue;
                }

                String newFolderName = Path.of(
                        folderName.isEmpty() ? PATH_CURR_DIR : folderName,
                        file.getName()
                ).toString();
                result.addAll(listFolders(isFoldersOnly, isRecursive: true, newFolderName));
            }
        }
    }

    return result;
}
```

Buggy code



```java
private List<LsResult> listFolders(
        boolean isFoldersOnly,
        boolean isRecursive,
        String... folderNames
) throws LsException {
    List<LsResult> result = new ArrayList<>();

    for (String folderName : folderNames) {
        LsResult content = listFolder(isFoldersOnly, folderName);

        result.add(content);

        if (isRecursive) {
            for (File file : content.getFiles().stream().sorted().collect(Collectors.toList())) {
                if (!file.isDirectory()) {
                    continue;
                }

                String newFolderName = Path.of(
                        folderName.isEmpty() ? PATH_CURR_DIR : folderName,
                        file.getName()
                ).toString();
                result.addAll(listFolders(isFoldersOnly, isRecursive: true, newFolderName));
            }
        }
    }

    return result;
}
```

Fix