NUS
National University of Singapore

School of Computing

CS4218 Software Testing

Testing Efforts

Introduction

For Milestone 3, we will mainly document the team's test plans, overall testing effort done for this project with some details, and an overview of the faults found. Also, our experience with Test Driven Development (TDD), tools for generating tests, the hackathon, and the rebuttal.

Test Code (LOC) vs Source Code (LOC)

For the number of lines of source code written, an estimate would be given as there are many commits over the course of this project which includes editing, deleting, or adding of lines and would be infeasible for the team to keep track of. Hence, the difference in the number of lines of source code in the latest commit compared to the number of lines of source code in the base project would be provided. For the number of lines of test code written, we excluded the count from test files provided by the teaching team and from the test files generated from Evosuite. The numbers are taken from the "Statistic" plugin in Intellij, under source code lines which would exclude the blank and comment lines, showing a better estimate of the number of lines contributing to the functionalities of our shell implementation and test cases.

Estimated Source Code Written	**Test Code Written**
1941	24750

The team realized that the number of lines of test code written is significantly more than the source code written. It is estimated to be about 12 times more. Although we did expect the number of lines for the test code to be more than the source code written, the team did not expect that much of a difference to be seen. The reason why there is such a huge difference is that the team did put significant effort into planning and generating test cases by using the techniques taught in the lectures and much more time is spent on testing the application than implementing the application.

Test Plan, Methods, and Activities

The team decided to split the various shell application features equally such that every member would be implementing and conducting tests for a set of features they are responsible for. This allows the shell features to be tested effectively as each member would be the most familiar with the requirements of the features they are implementing. The team would meet up every Thursday to discuss the progress of every member's work and the next set of tasks that is required to complete which allows the team to efficiently carry out the testing activities planned.

Milestone 1 (31 January - 28 February)

Our team first made sure to clarify as much as possible on the queries we had for the shell application features, ensuring that the specifications were well understood. Then, we moved on to implementing the features while keeping in mind that extensive testing would be required to be done. Both of these have helped in generating test cases, especially for edge cases, and helped us to write code that can be easily tested. Next, we utilized category partition testing and pairwise testing to efficiently generate effective test cases. At last, we wrote unit tests for methods using the test cases while utilizing Mockito to isolate the methods under test.

Milestone 2 (28 February - 21 March)

Since our team had already written test cases for the unimplemented shell features, we decided to try out TDD as suggested. We followed the TDD process of running the test cases, making some changes to the code to pass the test cases, and repeating this process until all test cases passed. In general, TDD took up more time to develop the shell features compared to our development process in milestone 1, but it did help us during our development of the application since the specifications were more well-defined, especially with edge cases. We then continued on to writing integration tests that mainly utilize pairwise testing to reduce the number of test cases due to the number of shell features present.

Hackathon (21 March - 26 March)

The team mainly conducted system testing on other teams' code as it can be done easily without taking up as much time. The system test could also be easily done as effective test cases were already generated in milestones 1 and 2.

Rebuttal (1 April)

We did our rebuttal late due to a small hiccup by the teaching team. However, we were still able to experience the rebuttal which did help us in including more test cases that we missed out on.

In general, the team felt that the Hackathon with Rebuttal is the most useful activity as we test the other team's code without bias that did help some of us find bugs in our own code, and getting a bug report for Rebuttal did help us to include test cases that we have missed out on. This activity helped us to add more test cases to our current test suite and increased our confidence in our implementation.

Coverage Metrics

In order to get assurance that the test cases are done sufficiently, the team makes use of Jacoco in order to check the branch coverage of our test cases. As branch coverage subsumes statement coverage, the team aims to achieve 80% branch coverage to ensure good test quality.

With this in mind, the team was able to create test cases with the following test coverage before the hackathon.

Application	**Method Coverage (%)**	**Line Coverage (%)**	**Branch Coverage (%)**
CatApplication	100	96	97
CdApplication	100	96	95
CpApplication	100	94	91
CutApplication	100	97	89
EchoApplication	100	86	100
ExitApplication	50	50	100
GrepApplication	100	98	98
LsApplication	100	92	96
MvApplication	100	90	95
PasteApplication	100	99	96
RmApplication	100	92	88
SortApplication	100	96	95
TeeApplication	100	97	90
UniqApplication	100	95	96
WcApplication	100	96	96

Othey Key Classes	**Method Coverage (%)**	**Line Coverage (%)**	**Branch Coverage (%)**
app/args	85	93	92
cmd	94	88	92
parser	84	94	86

util	88	94	88

Excluding Echo and Exit Application which are relatively simple features, CatApplication and GrepApplication *(Highlighted in Green)* have the highest branch coverage. In contrast, CutApplication and RmApplication *(Highlighted in Red)* have the lowest Branch Coverage among the features.

Based on the Bug Report, the results are surprising as the 2 commands with the highest coverage have more bugs than the 2 commands with the lowest coverage. This could be due to the fact that Branch Coverage only proves that the code base is well covered, but it does not take into account the exact requirements for the command. The test cases created are based on the tester's understanding of the requirements which could be a shortcoming. If the tester's knowledge which acts as the oracle is wrong, despite the high branch coverage, it will not be reflected in the number of bugs that could occur in the program.

Application	**Line Coverage (%)**	**Branch Coverage (%)**	**Number of Bugs**
GrepApplication	98	98	2
CatApplication	96	97	2
CutApplication	97	89	0
RmApplication	92	88	1

Automated Test Case Generation

Evosuite was used to boost our test quality and branch coverage by automatically generating a set of test cases. This is particularly helpful for classes with low branch coverage as the team was able to set the parameters on Evosuite, to generate test cases that optimize branch coverage.

Overall, making use of Evosuite improves our overall coverage as summarized in the table below.

	Method Coverage	**Line Coverage**	**Branch Coverage**
Without Evosuite	94% (242/255)	92% (1940/2093)	90% (1121/1242)
With Evosuite	94% (281/296)	94% (2054/2176)	92% (1153/1244)

Evosuite undoubtedly helped the team include comprehensive test cases and improve test coverage for certain classes. However, the team kept our previous test cases in the project as we felt that there were some limitations in using Evosuite and did not generate more tests from it as most of them were not as helpful.

Based on the team experience, Evosuite limitation includes:

- The test cases do not understand the requirements of the application
    - For example, the test case generated do not know that running the MvApplication should move a file from one of the given source to the destination
    - Hence, the test cases generated are mostly based on the return values of the method only
- Increasing the time to generate the test cases does not necessarily translate to better coverage
    - Generating the tests with the recommended command for CutApplication only yields about 50-60% coverage. Even when running the command with `-Dsearch_budget 600` flag, it still generated the same amount of test cases with the same coverage.
    - Hence, even when increasing the search time for generating tests, it did not increase our test coverage.
- Evosuite might create test cases that are not dynamical enough
    - For example, it created a test case for LsApplication in which it checks directly for the string of the current directory of the user that runs evosuite. Hence when passing this over to the other teammates, this test case will definitely fail.
    - Test cases that pass on an OS might not be able to pass on a separate OS.

- Evosuite’s test cases cannot consider the “deeper” edge cases
    - For example, in Paste, the test cases generated do not consider the content of the files that are being pasted. So, key behaviors of paste that involve pasting files with content of different lengths will not be tested. Hence, we have to manually create those test cases.
- Increases the overall testing runtime significantly
    - For example, if our test cases are run without Evosuite's test cases, it takes 2.6 seconds to complete. With Evosuite’s test cases, it doubles the runtime to 5.31 seconds. This is presumably due to the overhead in setting up Evosuite’s test scaffolding.

The team was not able to find any bugs through generated test cases by Evosuite as well. Hence, the team still continues to perform manual testing to ensure that our test cases are effective in detecting buggy implementation.

Debugging

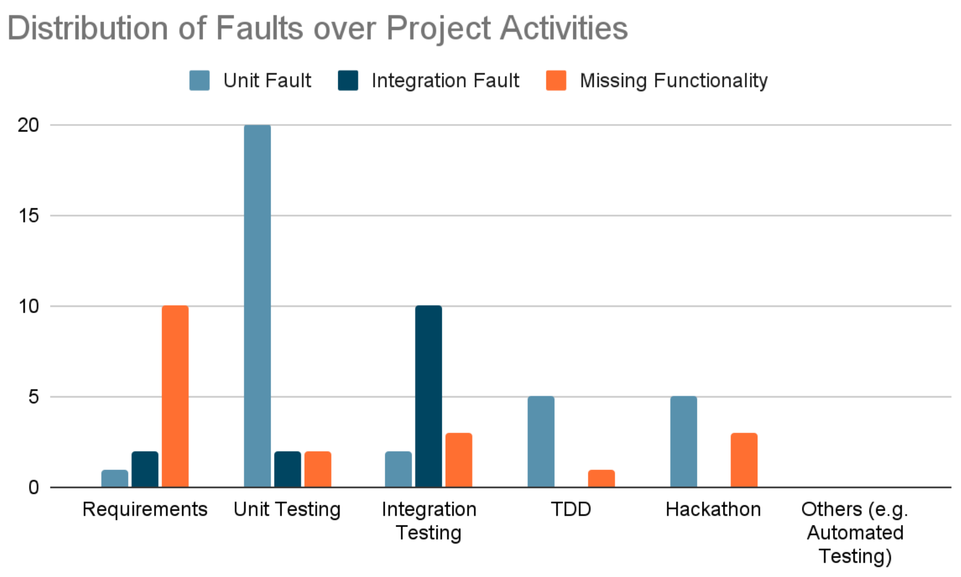
In terms of Debugging, the team makes use of Intellij’s IntelliSense and also the default debugger to assist the team during development. Intellij’s IntelliSense was able to alarm our team when there are compile errors while Intellij’s debugger allows the team to trace through the code to isolate errors.

Our team did not change any coding or testing practices due to the use of Intellij’s debugger. However, some of our team members had used Intellij’s debugger during milestone 1 to trace through the code and find bugs or errors through different parts of the codebase. Tracing the code also allows the team to have a better understanding of the code before directly resolving the bugs during development.

One possible automation that could be useful would be having a continuous integration (CI) pipeline. One issue the team has faced is the fact that our team members consist of both MAC and Windows Operating systems users. There are certain situations where test cases were able to pass when running on a Windows machine but it failed to run on a MAC machine. The team does not have access to a UNIX machine and hence there could be a possibility that our test cases could fail on a UNIX machine. If a CI pipeline is set up correctly, it will allow the team to run the test cases automatically on all 3 major operating systems' virtual machines, lowering the chance of having unexpected errors due to differences in the operating system. However, due to time constraints, the team did not set up a CI pipeline.

Bugs Found

Estimated Distribution of Fault Types over Project Activities



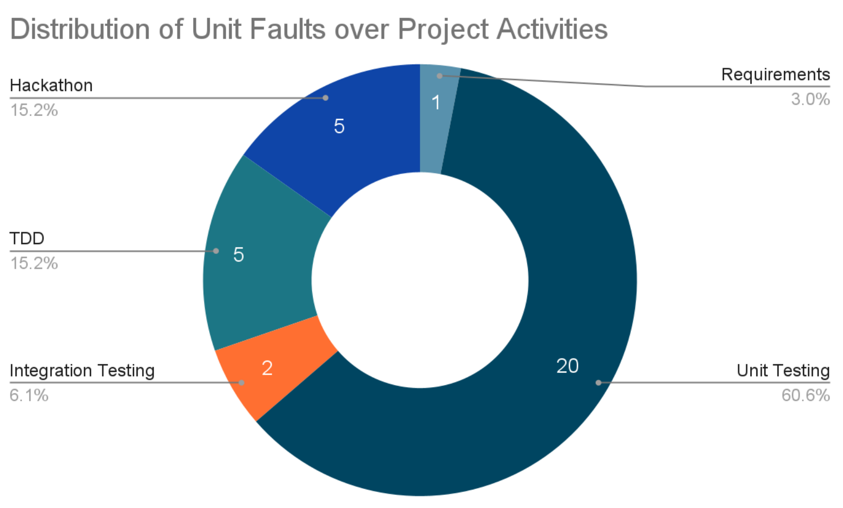
** Numbers used in the charts are estimations which strongly referenced the list of bugs in the appendix.*

We have tracked the list of bugs identified which are non-exhaustive, and recorded to best effort. In this bug analysis, we have used an estimation that strongly references the list as tracing bugs such as missing functionalities can be difficult as some faults are identified personally and fixed before Pull Request reviews.

We have categorized the faults into three categories (unit, integration, and missing functionalities). It's undoubtedly that the respective activities (unit testing -> unit faults) have identified most bugs within its category. Most unit faults are identified through unit testing which was completed in iteration 1, which led to lesser bugs in iteration 2 via Integration Testing and TDD.

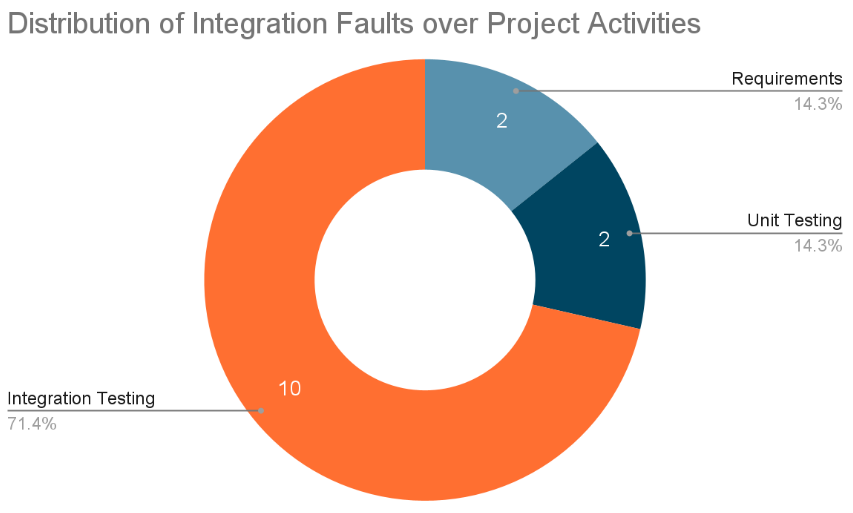
Overall, we felt that the distribution of faults had met our expectations since we have invested a substantial amount of our time in developing unit and integration testing, which resulted in no major bugs found and a smaller amount of faults found during the hackathon.

Unit Faults



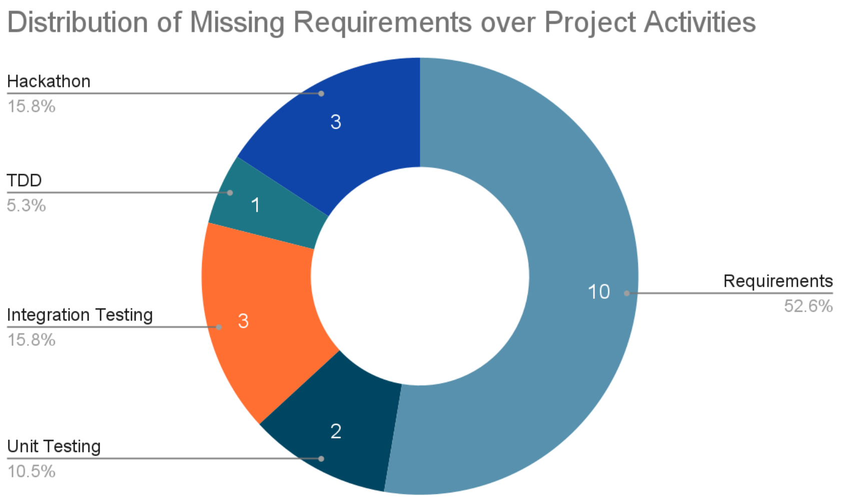
We came up with test cases involving different inputs during unit testing to ensure our application was working as expected. This is where the majority of the unit faults are discovered (e.g., null input does not throw an exception). In addition, the TDD (public tests given by the teaching team), hackathon, and others (mainly automated testing via Evosuite) were a great addition and have identified a number of unit faults that we were not able to identify during the unit and integration testing process.

Integration Faults



Similar to unit testing, in integration testing, we came up with test cases that involve the combination of different commands to ensure our application is working as expected. Test cases developed mainly involve chaining multiple commands through operators such as command substitution, piping, and semicolon operators. Hence, this process helps with the identification of integration faults. The most common fault identified is a mismatch in the exception interface (e.g., throwing the wrong exception).

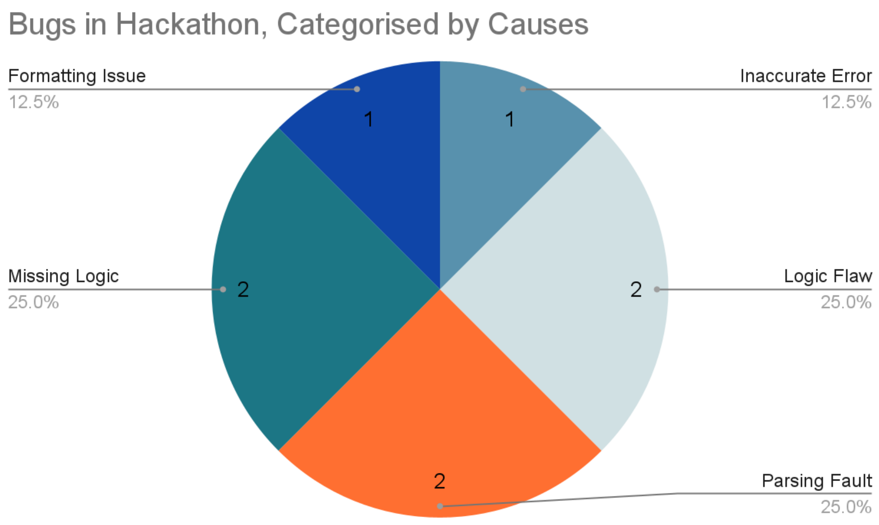
Missing Requirements



Since in iteration 1 we had to work on the implementation, most of the missing functionalities were found through requirement analysis since we had to reference the project description and Linux's shell behavior to figure out what was missing and what needed to be implemented based on the skeleton codes. Integration Testing has also identified a number of missing requirements during the chaining of multiple commands as well as Hackathon, as it eliminates the developer's bias in choosing what to test.

Analysis the causes of bugs found in Hackathon

In the hackathon bug report, we have accepted 8 bugs (3 missing requirements and 8 unit faults). We have further broken it down into the root causes.



The leading cause of the bugs found in Hackathon is tied between logic flaw (found in Command Substitution), parsing fault (found in ArgsParser), and missing logic (found in ls).

Accumulation of Faults

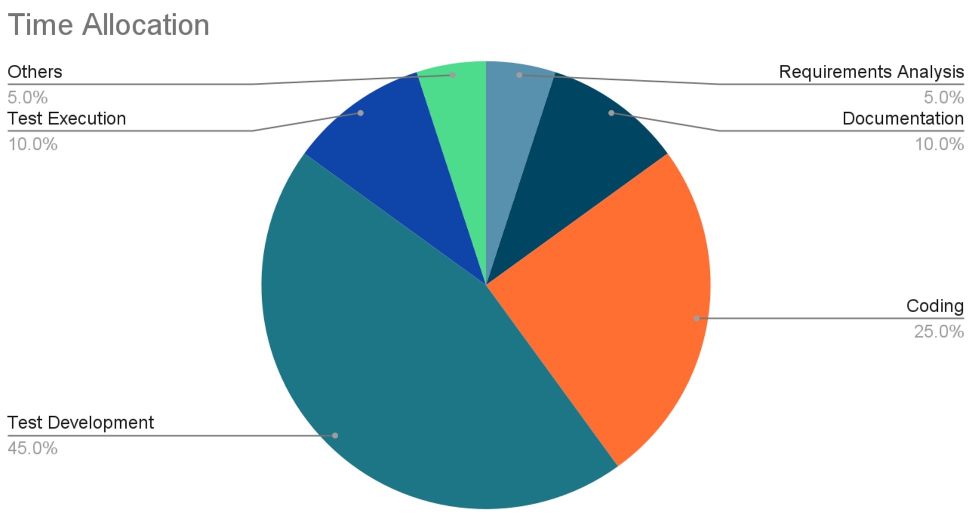
We believe it is true that faults tend to accumulate in a few modules. One of the examples is even though each application is using a different subclass of the parser, due to there being a parsing fault in the ArgsParser itself which propagates to two different bugs.

Predominance Fault Classes

In general, we have realized that unit faults predominate among the fault classes. This may be due to the fact that the shell applications often contain non-trivial logic and are mainly contained within a class.

We have observed a lower amount of integration faults. We believe it is due to having lesser unit faults. As if there are little to no unit faults, the different applications and commands should work together as expected since the interaction between applications follows a structure dictated by the shell operators. The causation of integration faults usually lies within the operator itself, such as the command substitution's Argument Resolver not parsing nested mixture of single, double, and back quotes properly, which leads to different behavior.

Estimated Time Spent on Different Activities



When we started the project, we spent some time on requirement analysis (~5%) by understanding and discussing the project requirements through the project description and understanding Linux's shell behavior through individual self-exploration since some of us are new to some of Linux's shell commands. In addition, we spent a substantial amount of time (~25%) on coding the implementation in both iteration 1 (Basic + EF1) and iteration 2 (EF2).

We have allocated most of the time (~45%) to test development, as software testing was the project's focal point. This includes developing a testing plan (MC/DC, category partitioning, pairwise combinatorial testing) and writing the actual test cases. Afterward, we will perform test execution (~10%). This involves reviewing PR that includes cross-testing among members. The test cases written by a member will be run and executed by another 2 members (preferably one on Windows and one on Mac to ensure the tests are passing on different OS).

Throughout the iterations, we have spent approximately 10% on documentation and report and the last 5% of our time on other activities, such as integrating mocking tools, automated testing, and generating mutation test reports.

Development

Test-Driven Development (TDD)

Based on our experience with TDD from Milestone 2, we recognize the significant advantages that TDD brings to our development process. By writing tests as we implement our features, we are able to build up a regression test suite as the development progresses, allowing us to almost never get a repeating bug since every bug we found should be discovered by a test that was introduced the last time that bug occurred. This provides us with the confidence to make large-scale refactoring like what was done in Paste (Revamp an algorithm to align with Linux's implementation) without worrying about breaking everything else as they are covered by our regression tests. This approach also ensures the amount of coding we have done on the implementation end is minimized to just what was required to pass those tests.

However, our experience is not always positive. TDD is much more time-consuming than RDD to get the functionality of a feature going due to its upfront overhead of writing tests, this results in many of us taking longer to implement the applications in Milestone 2 than those in Milestone 1. TDD can be troublesome when the project specifications are vague. For example, during the implementation of Paste, the expected behavior of executing *paste* was not specified, this allows the person writing the tests and the person implementing the functionality to have different interpretations. In our case, we decided to rewrite our tests to align with Linux's implementation of that behavior instead.

Although this is not what we directly experienced, we observe the possibility of TDD becoming troublesome when there are places in the project that are structurally volatile. By that, we mean the code or interfaces are likely to change a lot structurally. Therefore, we believe TDD should be implemented after a well-established specification has been made.

Requirement-Driven Development (RDD)

Based on our experience with RDD from Milestone 1, the requirements established for each feature allow us to determine when we can consider a feature completed instead of endlessly improving it and compromising the development process. Ideally, this approach will be the fastest to execute as we no longer have the overhead of writing out tests but we only need to address the requirements of each feature.

However, as the tests in RDD are written after implementation, regression bugs during the implementation of a new feature can be prevalent. To resolve bugs, we have to spend additional time debugging instead of developing. Furthermore, if the project specification is not precise enough, edge cases may not be identified during the implementation and they will eventually lead to further bugs.

Comparison of both approaches

After reviewing our experience with both approaches, we identify that the core deciding factor between them is a trade-off between the length of development time and the robustness of a codebase. Without TDD, the time spent debugging issues in RDD can build up to or even more than the overhead in TDD to write tests. On the other hand, RDD is an approach that almost all developers can easily follow while TDD requires the developers to know how to write effective tests or have a team of testers with the sole aim of writing tests. This additional skill requirement can be a deciding factor as well.

To conclude our observations in those approaches, TDD should be the default approach to have for development if the team has the skillset and the manpower to maintain the codebase's robustness while fulfilling the deadline. If not, falling back to RDD can be a viable option as well.

Code Design

In our implementation, many of our applications made use of a corresponding *ArgsParser* in their *run()* method to parse and process the arguments of an application. In order to unit test those applications, we have to mock the *ArgsParser* to remove the dependency. As such, the applications have been modified to allow an injection of a mocked instance of *ArgsParser* instead of using the instance that was created within the application.

Hackathon Experience

The hackathon experience gave the team a chance to better improve our shell implementation. With test cases brought over by other teams, it injected a fresh new testing perspective, which helped us discover blind spots in our testing earlier on. We were then able to better improve our shell implementation by including these test cases in our test suite, and fixing the discovered bugs till it passes.

Apart from the test cases given by other teams, some of us were also able to discover additional bugs in our own shell while testing the other team's implementation. As the team mainly did unit and integration testing on the components that we are working on, we tend to be fixated on a particular idea on how testing should be performed for it. However, when we performed manual testing on another team's implementation, we tested components that we weren't familiar with, and tried them on our own implementation as well. This allowed us to discover bugs that weren't found earlier, thereby improving our shell.

Learnings

Criteria to evaluate the quality of the project

There are several criteria to evaluate the quality of the project, which are as follows:

1. **Extensibility**
   Extensibility determines whether a project can be easily extended when a new feature is required. For example, in the context of our shell implementation, if there's a new requirement to add in a new shell application, the project's code should be able to easily accommodate this new feature with minimal or no change to the existing code.

2. **Performance**
   Performance is another aspect that should be considered when evaluating the quality of the project. Performance can be broken down into two sub aspects, which are: time and space. For time, we should expect the shell application to evaluate and execute its appropriate actions within a reasonable time. For example, executing a simple ls command on a trivial directory (with only 5 files) should not take more than 2 seconds. For space, it can be evaluated based on the memory space being taken up by the application. For example, for our shell implementation, we would expect it to use less than 200MB of memory. Since performance impacts how users use our application, it should be considered as one of the criterias to evaluate the quality of the project.

3. **Documentation**
   A project should also be evaluated based on how well documented it is. This allows developers who are working on the project in the future to better understand the system, speeding up the development process. By documenting a project with its relevant design decision, it allows the developer to internalize and understand the importance of each component, thereby having a concrete plan to produce a successful application.

The team feels that none of the criterias are counter-intuitive as they do not contradict each other and are required to achieve a high quality project.

Important Reflection

Through CS4218 and the Shell Project, we learned that the testing space is vastly huge, and there are many different combinations that could lead to a combinatorial explosion. However, we could adopt various testing methods, such as pairwise testing, and modified condition/decision coverage (MC/DC) testing, to greatly minimize the amount of test cases needed, and yet achieve reasonable coverage for our program/application. Through creating test cases, we were able to give ourselves a certain level of assurance that the program wouldn't break if we were to make huge changes to our code base, i.e. preventing regression. In addition, the weekly labs were beneficial and introduced many useful tools and techniques which we have adopted in our project to make it even more robust.

New Topic Suggestion

While the Shell Application project was interesting as it exposed us to various shell commands we've not used before, the team felt that it lacked a "fun" element and would have preferred working with modern technologies instead. Therefore, we suggest using web development as a base topic for projects, and allow students to implement fun and creative web applications. An example would be a simple CRUD (create, read, update, delete) web application, or an online chatroom.

The following are the benefits web development as a base topic would bring:

1. **Exposure to modern technologies with testing frameworks**

   Having web development as a base topic would expose students to modern technologies that are being used in the industry. For example, a student could choose to adopt the MERN (MongoDB, ExpressJS, React, NodeJS) stack. This would allow him/her to be exposed to front-end and back-end web technologies, as well as handling databases.

   Furthermore, modern technologies such as NodeJS, do have testing frameworks available as well. One example would be Mocha. Mocha has specific functions which could very well cater to test-driven development (TDD). This would meet the aim of CS4218, where testing is centered around the module.

2. **GUI Testing**

   With our Shell application, we are not exposed to the possibility of GUI testing. However, if we choose to adopt the MERN stack as mentioned above, we would be exposed to using front-end technologies such as React. Since React renders a front-end web page, we could then use an automated GUI testing tool, such as Selenium, to perform automated testing on our web pages and user interfaces in our front end application, putting theory into practice.

Appendix

Bugs Found (In table form)

** The table below is non-exhaustive and is recorded to our best efforts.*

Bug Description	**Fault Type**	**Found During**
Ls with emptyfolders are not showing the correct result	Unit Fault	TDD
paste a.txt b.txt (where a.txt has more lines than b.tx) gives a wrong answer	Unit Fault	TDD
Running the cat command with 2 files with empty content returns a wrong answer.	Unit Fault	TDD
Grep’s individual methods (grepFromFiles / grepFromStdin) should throw an exception if null is passed in as the stream input.	Unit Fault	TDD
Grep’s exception message does not include the application name	Unit Fault	TDD
Pipe command that ends with ; returns an error	Integration Fault	Integration Testing
Tabs are not handled within quotes (eg. \t not disabled in ‘’ and “”).	Missing Functionality	Unit Testing
ArgumentResolver - To append tokens into an unmatched quotes segment, the subOutputSegment should not be empty.	Unit Fault	Requirement Analysis
ArgumentResolver - Added to close output streams. Without this, we cannot perform any cleanup or teardown of IO-redirected files.	Unit Fault	Unit Testing
CallCommand - Without the right conditions, the passing of input/output streams during piping will be incorrect for the final command. This can result in incorrect terminal output.	Unit Fault	Requirement Analysis
Call commands with input redirection were not handled. Had to delete the unnecessary “break;” under “CHAR_REDIR_INPUT” switch case for input redirection to work as intended.	Unit Fault	Unit Testing
Output of call commands was not correctly written into the redirected output file due to missing check for “>” argument. Added in the condition to check if the current argument is equal to “>” to handle output redirections.	Unit Fault	Unit Testing
IO Redirection - Did not handle null or empty argument lists correctly. Change from using “&&” to “||” to handle null or empty argument lists correctly.	Unit Fault	Requirement Analysis
Sort application did not sort according to the conditions stated in the project document, it was only sorting by ASCII value, not all symbols/special characters came	Unit Fault	Requirement Analysis

before characters and numbers. Added code to ensure all symbols/special characters come first instead of sorting purely by their ASCII values.		
Ls command does not work with absolute file path	Unit Fault	Unit Testing
Tee command does not exit after a command that is similar to *tee file.txt* < *content.txt* is ran	Unit Fault	Unit Testing
Paste command has wrong output after a command that is similar to *paste - - -* < *c.txt* is ran	Unit Fault	Unit Testing
When copying a mix of valid and invalid files to a destination using Cp command, the valid files were not copied over	Unit Fault	Unit Testing
Wrong type of exception was thrown when ls command is executed using absolute paths	Integration Fault	Integration Testing
Paste command return a wrong error message when an invalid flag is given	Unit Fault	Unit Testing
Executing Paste command without any argument should behave like an echo where any user input is displayed back	Unit Fault	Unit Testing
Grep command from the skeleton code does not parse absolute path properly	Unit Fault	Integration Testing
Argument Resolver should not append space to the last sub command output	Unit Fault	Integration Testing
Regex Argument - File.separator can give "\" on Windows which is an invalid separator on unix platform, and "/" is usable on either OS	Unit Fault	Unit Testing
Cd should not accept more than one arguments	Missing Requirements	Requirement Analysis
Cd should not accept zero arguments	Missing Requirements	Requirement Analysis
No error thrown for *ls “”*, should be exception	Missing Functionality	Hackathon
echo "a`"c"`"B"``" - Echo fails to continue parsing in the presence of an incorrect command substitution, however in the default Unix implementation the error message was simply returned and echo continued parsing the rest of the arguments	Unit Fault	Hackathon
No error message for rm -d	Missing Functionality	Hackathon
Extra new line printed for cat empty.txt when empty.txt is an empty text file	Unit Fault	Hackathon
Inaccurate error message for mv: Cannot copy a directory into itself, but fileB.txt is not a directory	Unit Fault	Hackathon
No output given when ls hidden file/folder. Expected listing of hidden folder	Missing Functionality	Hackathon
echo "`exit` `cat README.md`" - does not run cat	Unit Fault	Hackathon

command and only exits. Should run cat command AND exit		
wc fileC.txt > fileC.txt - fileC is not blank but treated as a blank file, resulting in 0 0 0 fileC.txt In the fileC.txt to be written	Unit Fault	Hackathon

Acceptable PMD Violations

PMD Violation	**Explanation**
Closure Resources	● Overall, there are various PMD issues regarding CloseResources that were suppressed. ● This is largely due to using an external util method to close the resources which PMD does not recognize.
Long Variables	● Found in the ErrorConstant class. Long and descriptive constants help the programmer understand the error easily. Hence, the team decided to keep the constant naming, prioritizing understandability.
Excess Method Length	● Found in classes such as ArgumentResolver, CommandBuilder, GrepApplication, RmApplication, ApplicationRunner, and IORedirection. ● **ApplicationRunner**: As we introduce more applications, it is inevitable that the length of ApplicationRunner's switch statement increases and leads to the runApp method getting excessively long. To ensure that it continues to work with the given interfaces, we will not make any significant modification. ● **Others:** Results due to the interface methods given or the method are rather complex, and many parameters are required.
Class Naming Conventions	● Found in CommandBuilder and Environment, the class name given seems intuitive and descriptive which the team feels is reasonable
God Class	● Despite the team's best efforts, the team feels that the complexity in the classes is high, and having a larger class is reasonable. ● **CutUtilsTest:** The test file requires a relatively large amount of constants due to repeated usage of certain values for testing and a large number of methods to test the methods of a class. ● **GrepApplication:** The original skeleton (with TODO) had already triggered this warning. However, an attempt to refactor the methods persisted with the warning. Since we are not allowed to remove any predefined method from the skeleton codes, usage of auxiliary files such as GrepArguments introduced even more PMD warnings due to redundant assignments and repeated logic that violates Software Design Principles. ● **PasteApplication**: The necessary process to merge the files and stdins into one output is fairly complex, resulting in methods being fairly long. In order to keep the class readable, we have extracted certain operations of the methods into smaller methods which increases the length of the class and the number of classes. Even though this triggers the god class error from PMD. We believe that it is reasonable to prioritize readability of the methods.
Avoid String Buffer Field	● Found in RegexArgument, using string builder gives the developer more flexibility when dealing with strings.
UseVarargs	● Using string array instead of varags allows for better flexibility to the developer and the team feels that it is easier to be used.
PMD violations from Evosuite	● As mentioned in the forum, it is not required to resolve PMD violations from auto-generated tests.

generated tests	● Forum post: https://luminus.nus.edu.sg/modules/efe37db4-83fb-4090-a0d4-c86d3cdba61c/forum/categories/b371264e-d699-409e-8a20-bfbc923bd2ca/d4405086-cd6f-4fa1-ba73-ce5cf9a1e4ac
PMD violations from TDD test cases by teaching team	● As mentioned by the teaching team, it is not required to resolve PMD violations from the TDD test cases provided by the teaching team.