QA Report

CS4218 - Team 3

Pedro Teixeira	**A0208300Y**
Shradheya Thakre	**A0161476B**
Tan Heng Yeow	**A0156106N**
Tejas Bhuwania	**A0176884J**

1. Lines of Code: Source vs Test

How much source and test code have you written?

Test code, including some automated tests adapted to fit our test suite:	7827 LOC
Actual source code:	3388 LOC

2. Testing Plans, Methods and Activities

Give an overview of the testing plans (i.e. timeline), methods and activities you have conducted during the project.

Timeline

Milestone 1

1. Identify planted faults using mostly a combination of compilation errors and warnings, manual testing and the PMD static analysis. These faults were noted down and unit tests later created where appropriate.
2. Early development of unit tests, experimenting with different formats.
3. Decision to use test assets kept in a specific folder for each test class and standardization of directory naming and structure.
4. Create unit tests for positive and negative cases of each application class.
5. Create unit tests for shell operators and command classes.
6. Create unit tests for utility classes.
7. Create boundary value tests where they were missing.
8. Create basic integration tests.

Milestone 2

1. Implement remaining functionality using test-driven development with a combination of our own tests and the ones provided.
2. Run remaining TDD tests:
    a. Classify failing tests into differing implementation decisions and bugs
    b. If a test case is deemed to expose an actual bug, adapt it to our implementation if necessary, or add an equivalent one to our own test suite if easier.
3. Investigate results of automated testing tools:
    a. We experimented with EvoSuite and SquareTest, with EvoSuite proving more useful in giving us good test cases which we had missed and also help in finding bugs and regression errors later on.
    b. Adapt selection of EvoSuite generated tests into our own test suite

4. Create integration tests focusing on commands being joined with different shell operators (functional integration).
5. Create integration tests focusing on using different applications working together (structural integration).
6. Generate coverage report and attempt to increase line coverage where it is lower with both unit and integration tests.
7. Create integration tests of complex interactions joining as many operators and applications as possible.
8. Create a system test that runs a sequence of commands directly from the main function in the application.
9. Generate final coverage report to present in the milestone report.

Hackathon

1. Make necessary adaptations for our test suite to work with other teams implementations.
2. Run unit tests against other teams' implementations:
    a. Whenever a test fails, check whether this is due to differing assumptions, differing implementation details, or due to an actual bug.
    b. When the issue is related to differing implementations adapt the test or try to run the command manually if easier.
    c. Document bugs found along the way.
3. Run integration tests against other teams' implementations in a similar manner.
4. Perform some more manual testing, and create new test cases for bugs found this way whenever appropriate.
5. Compile all relevant test cases for bugs found.

Rebuttal

1. Sort through bug reports running the test cases and analysing if they are appropriate for our assumptions and implementation choices.
2. Document all accepted bugs in GitHub's issue tracker.
3. Dive into understanding the main fault of the bug and think of potential fixes.

Milestone 3

1. Fix most accepted bugs and create tests for instances where those were not provided with the bug report.
2. Final code quality, formatting, and static analysis checks.
3. Generate and analyse the final coverage report.
4. Collect and organize data for this report.

Useful Activity

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

We believe unit testing was the most productive activity, as it helped us find the bulk of faults, as well as refine the design of the shell application. Designing unit testing was also useful in understanding the design of test cases for the application, catalog values for inputs and what types of boundary values are possible for cases specific to a shell-based project.

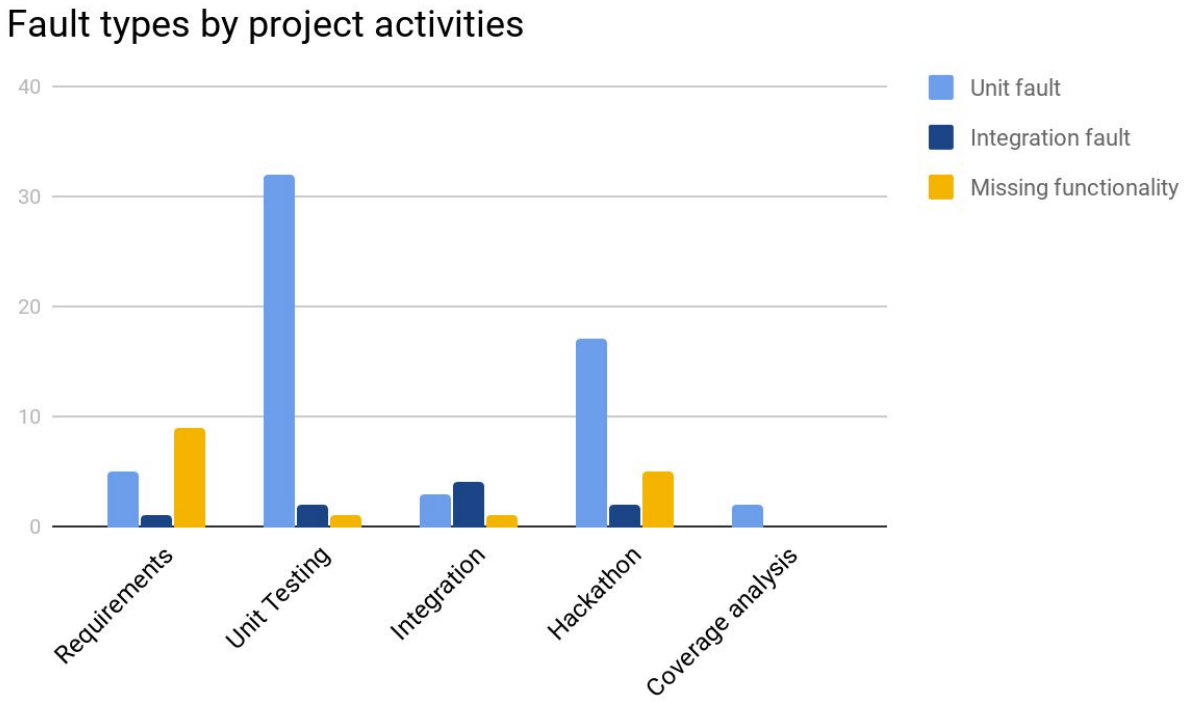
Another useful activity was assigning each person with certain commands and shell operators for the entire project timeline so as to maximise in-depth expertise in those modules and allow extensive testing. This, along with peer reviews from the team members, helped in facilitating better test suites and feature implementation.

3. Analysis of Fault Types and Project Activities

Analyze the distribution of fault types versus project activities

3.1. Faults by Project Activity

Plot diagrams with the distribution of faults over project activities.
Discuss what activities discovered the most faults. Discuss whether the distribution of fault types matches your expectations.

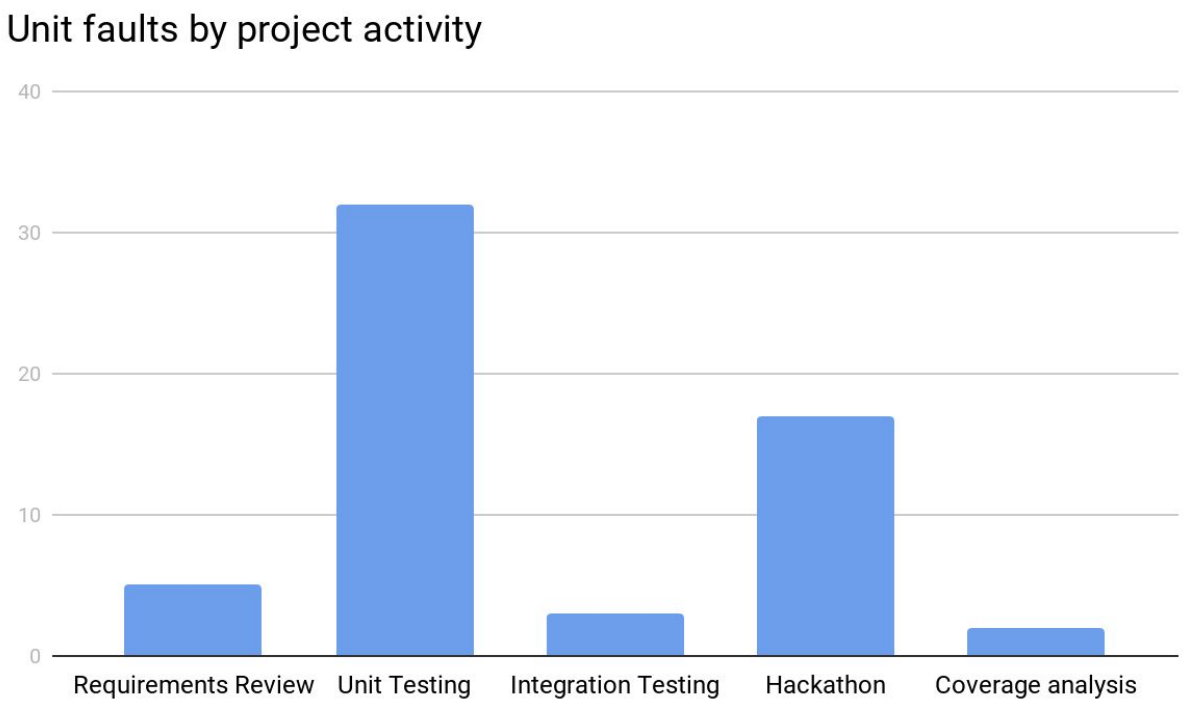


Overall, considering the nature of the project and our experiences throughout development, testing and other stages, the prevalence of unit faults matches our expectations, as the logic of each command was the main challenge during implementation, and the main source of problems when testing. Requirements review and the coverage analysis were also within our expectations.

On the other hand, the low number of integration faults, even during the effort leading up to milestone 2 was rather unexpected (see section 14). Especially given that the number of bugs found during the hackathon was significantly higher than during our integration testing. It must be noted, however, that when it comes to integration bugs specifically, there were fewer found in the hackathon than during integration testing.

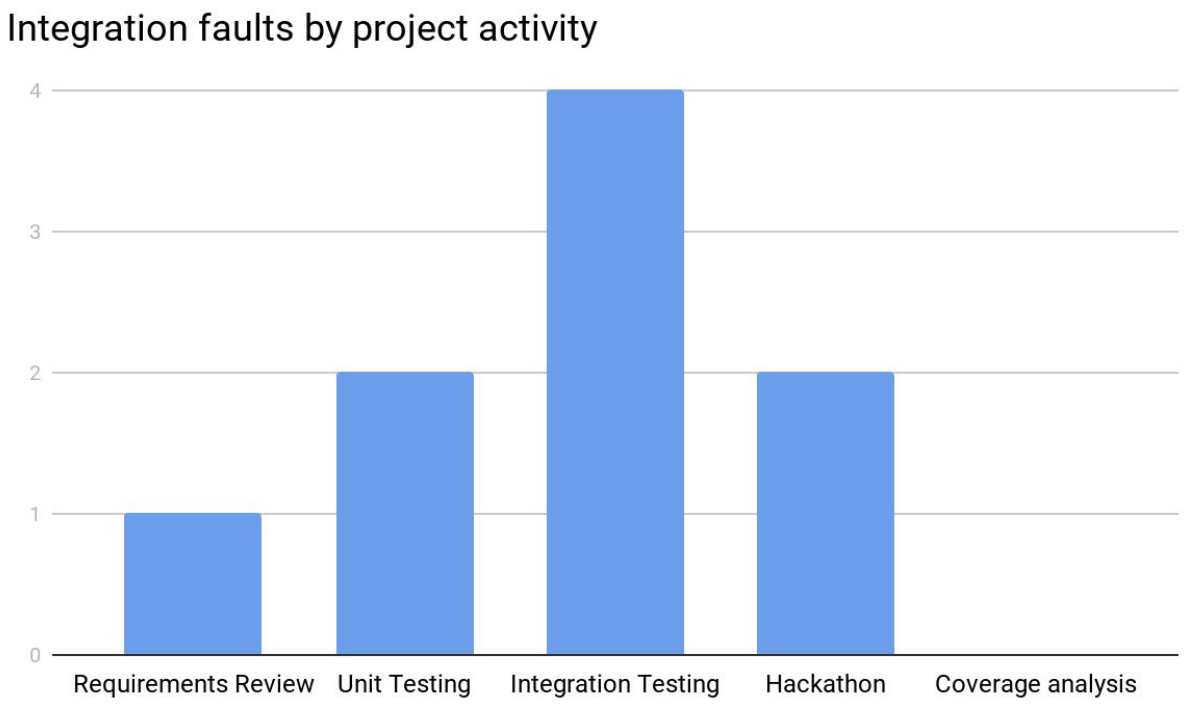
The following pages analyze this distribution for each type of fault in greater detail.

Unit Faults



As expected and seen on the graph, most of the unit faults were found in the unit testing phase which was our first phase of properly testing each class. Since we comprehensively tested everything from the early stages, we did not find many more unit faults later. Reviewing requirements and the coverage analysis also helped, as we were looking at each component in an isolated manner during the activity. Finally, the hackathon exposed a huge number of unit faults, as having other teams looking into our system removes the biases we had developed during our implementation and testing.

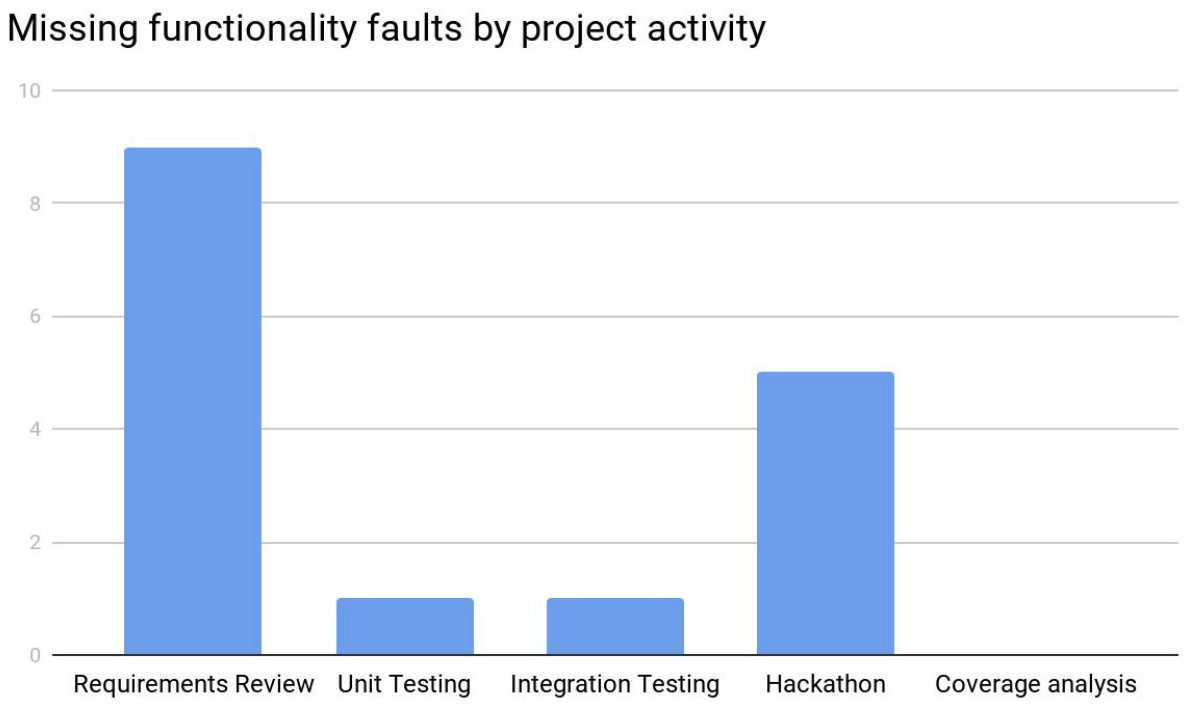
Integration Faults



We did not find many integration faults as compared to other types mainly because most of the implementation had predefined interfaces interacting well with each other. Most integration faults were a result of applying shell operators on command or combining result or execution of related commands.

We also found some integration faults in the unit testing phase which we believe is because despite trying our best to test components in isolation, there were times when integration of components was tested which led to these faults being exposed. Since the coverage analysis was mostly done for each component we did not find any integration faults but we did find some unit faults while exploring functional integrations during the requirement stage.

Missing Functionality Faults

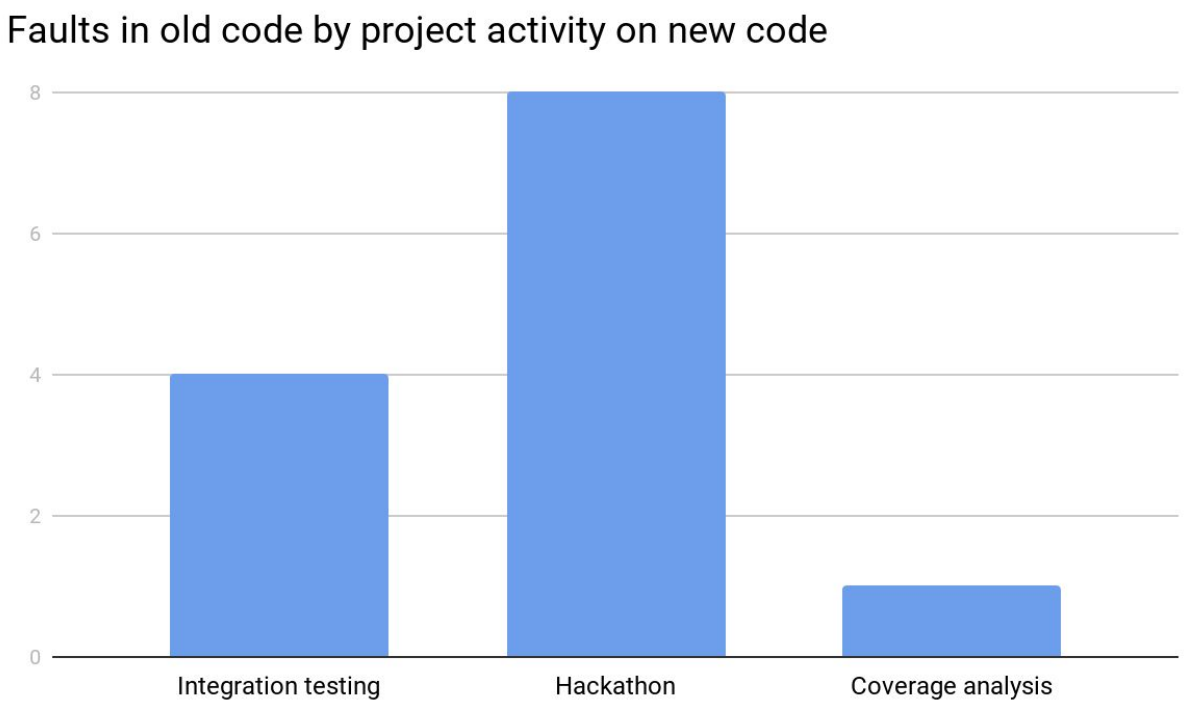


Most faults regarding missing functionality were found in the Requirements stage and Hackathon. Unit and Integration tests helped in finding very few from which we learned that good Requirement Review is helpful in finding faults early and exploring all possibilities and boundary cases for the system as it helps the developer understand the system well before testing it.

The hackathon helped in discovering faults that we overlooked or misinterpreted by looking at other team's functionality which we could have found in Requirements stage by spending more time on it. Coverage was mostly structural and hence no functionality bugs were found during that activity.

3.2. Faults in Old Code Found During Later Activities

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



Discuss whether the distribution of fault types matches your expectations

Faults in old code were expected due to regression errors when adding new functionalities that might affect the existing ones. We expected integration testing to find more of such regression errors and we feel this number may be low due to not focusing on every possible integration in the best way due to combinatorial explosion. Moreover, since the old code was already implemented, many possible requirements and implementation details that come intuitively while writing source code might have been missed while writing tests.

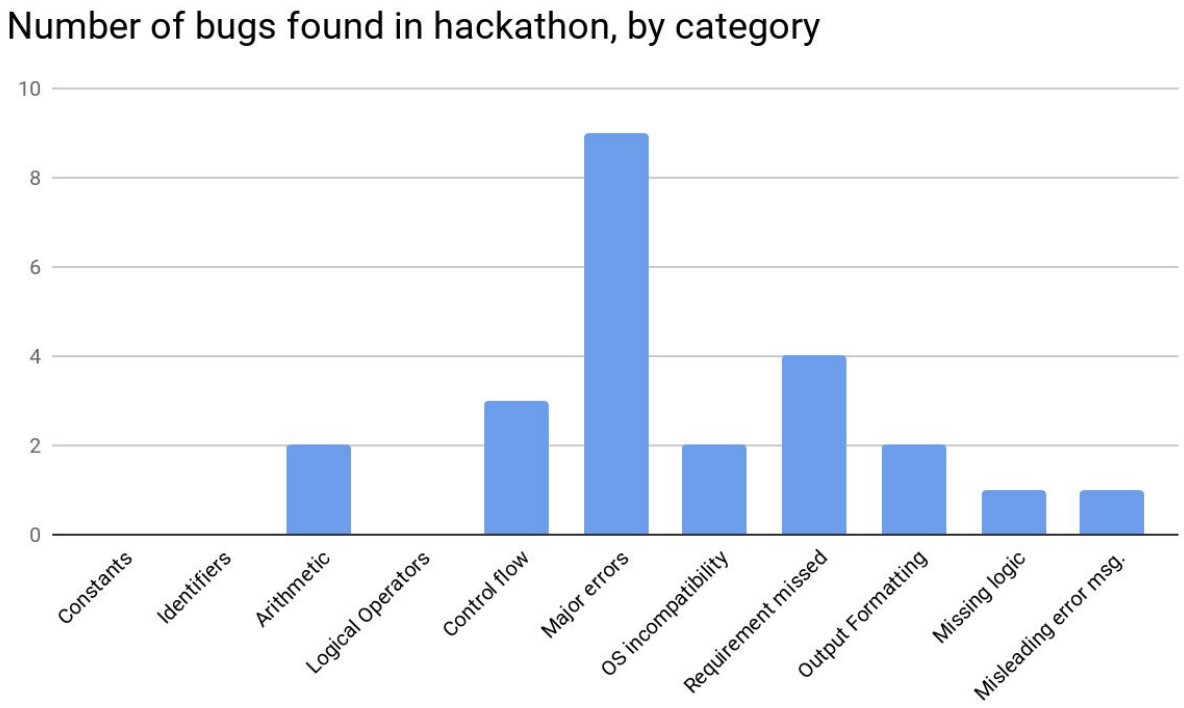
Hackathon found most such errors in old code as teams with a different perspective evaluated our requirements and explored areas we may have missed. Coverage analysis due to its structural properties didn't result in finding many faults in old code as our unit tests covered most possible branches.

3.3. Bugs by Type

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

Hackathon

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

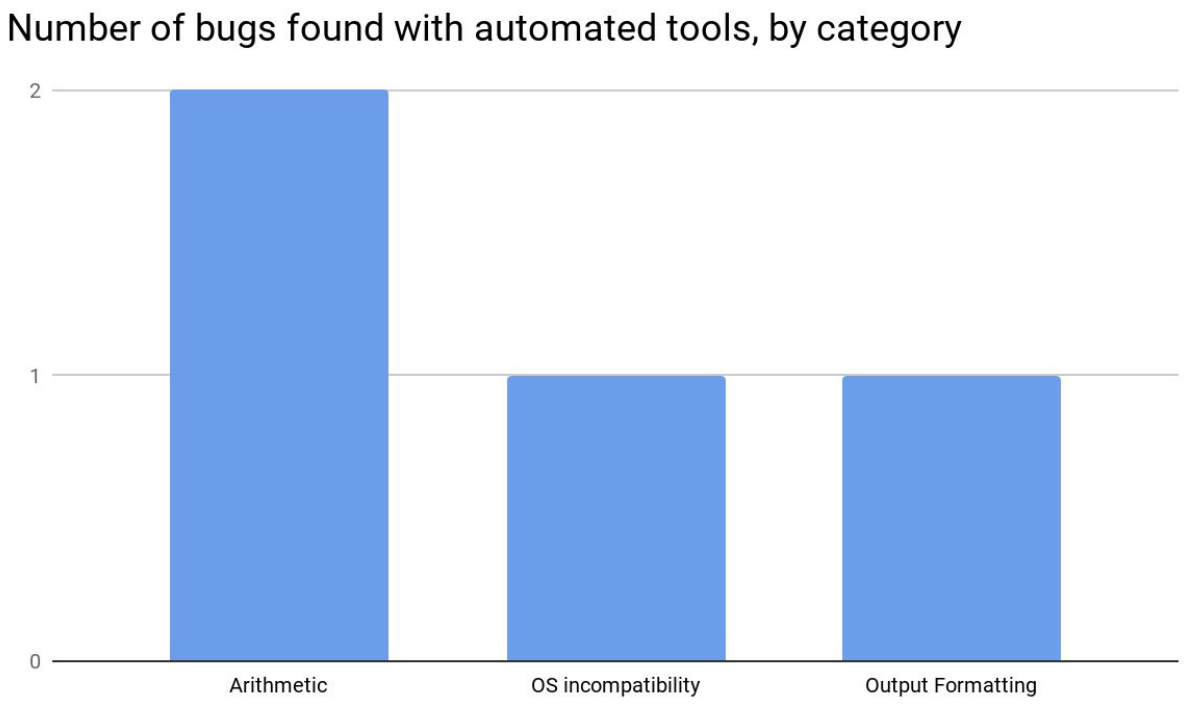


The leading cause of bugs found in the hackathon is major errors, most of them due to specific inputs with characteristics we had not considered during testing, and most of them caused by unit faults. One such example is the fact that copying non-empty directories would not copy their contents.

After that, missed (or overlooked) requirements were the second most common cause of bugs. While we did check our code against requirements, such details were nuanced and our biased analysis could not catch them. One example of this is the requirement that the filename passed to `find` with the `-name` flag has to be quoted. This example may seem irrelevant, but we recognise that in a real world scenario it could make a big difference.

Automated tools

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



While automated test generation tools were useful in generating more cases and increasing coverage, it didn't help us much in uncovering bugs and faults as this activity was done after Unit and Integration testing. However, since they had been testing random input with all possibilities we uncovered a few faults which would cause error if incorrect input is given. Some static analyser tools helped in pointing out possible NullPointerExceptions etc but these were already found during early stages of testing.

Accumulation of faults in few modules

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

For the most part, it is indeed true that faults accumulated in a few modules, in this case in the applications that contained the most complex logic, which reflects on the fact that unit faults were the most common throughout the project. As an example, in the hackathon specifically, out of 25 accepted bugs, 12 were in just 3 classes (`ls`, `cut`, `diff`). During development, applications like `paste`, which contains some complex control flow, were the classes where most faults accumulated.

Predominant fault classes

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

Overall we have seen unit faults predominate throughout the project. This may be due to the fact that the shell applications often contain non-trivial logic and must handle a large number of scenarios, while at the same time they are mostly self-contained.

Even though there is significant interaction between different commands this does not manifestate as much in integration faults because such interactions follow a very clear structure, dictated by the shell operators, and as far as the operators themselves are correct, there is not so much room for error between commands. Where things were more likely to fail is in interactions of the operators themselves with the applications and we saw this when a number of the hackathon bug reports indicated problems in the way `ls` handled globbing.

4. Time Allocation

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

- Requirements analysis and documentation: 10%
- Coding: 30%
- Test development: 45%
- Test execution (including debugging): 10%
- Others (e.g. coverage analysis): 5%

5. Test-driven Development

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

The main clear advantage of TDD was that once the tests are written, it makes it very clear for the developer what goals the software needs to satisfy. It also helped minimise the risk, often associated with requirements-driven development, of letting the application's complexity grow too much without catching certain bugs, which will become much more costly to identify and fix as the codebase gets larger and more convoluted.

On the other hand, TDD can remove some perspective as the actual requirements are abstracted away and the focus is all placed on the individual small pieces. In that sense, requirements-driven development can give a better perspective of the holistic goal of the application, avoiding potential validation problems later. Another area where requirements-driven development seemed to bring some advantages is in keeping design choices open when it comes time to implement. TDD can be restrictive and impose too many constraints on defined behaviour, if care is not taken to balance keeping the tests somewhat flexible to design choices and still useful for fault localization. For this reason, we noticed that careful interface definitions are of the utmost importance.

In the specific case of this project, TDD was not such a smooth experience because everyone had very different implementations. As a result, we also spent a lot of time fixing such differences rather than focusing more on bugs, the assumptions and validating them, which made the activity rather inefficient. The way the project was organized meant that when we started experimenting with TDD we received the tests already written (which in fact ends up being test-first development), but in a more realistic situation we would have to take some time to come up with the tests before implementing each new feature or module. Nonetheless, TDD helped us reconsider design choices such as when to throw exceptions and where to catch them, as well as the content of error messages.

6. Coverage Metrics and Code Quality

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

Coverage Metrics

Class Coverage:	100%
Method Coverage:	98%
Line Coverage:	98%

The following table shows the coverage metrics for the classes which had bugs identified in the hackathon:

Classes	**Bugs found**	**Line Coverage**	**Branch Coverage**
LsApplication	5	97.8	93.5
CutApplication	4	98.9	91.7
DiffApplication	3	93.2	75.6
GrepApplication	2	98.4	90.7
CpApplication	2	91.7	94.1
FindApplication	1	100	89.7
IORedirectionHandler	1	97.6	92.3
PasteApplication	1	92.7	97.5
RmApplication	1	100	84.3
SedApplication	1	100	100
SequenceCommand	1	100	87.5
CommandBuilder	1	100	100
WcApplication	1	97.8	86
MvApplication	1	100	93.8
Total / Avg.	**25**	**97.7**	**91.2**
Pearson's Correlation Coefficient (w/ bugs)		**-0.21**	**-0.16**

Reflections on Code Quality

Overall, coverage metrics do seem to very moderately impact code quality, although we did not specifically expect that the most covered classes would have fewer bugs as all had over 90% statement coverage. Commands like `sed`, `find`, `mv`, `echo`, `cd` and `rm` all had line coverage of 100% and were indeed mostly bug free (as far as we know from the hackathon).

At the same time, (the absolute value of) Pearson's correlation coefficient is rather low, indicating that there is only an extremely slight (linear) negative correlation between the number of bugs found in a class and its line and branch coverage by our test suite.

Classes like `echo`, `cd`, `sed` which achieved the highest branch coverage actually have the least number of branches. This does not necessarily mean their quality is better than that of classes like `find`, and `wc` which have lower branch coverage and significantly more branches but actually have few faults as well.

High quality of code is not one-dimensional, and so it also depends on aspects such as maintainability, readability and efficiency. Additionally, coverage does not in any way reflect whether or not the application satisfies all requirements, and some subtle bugs might have still been missed since it is not tested for all possible input values and full path coverage. There will almost always be inputs that expose faults in the system that testing has not considered, even if (statement/branch) coverage is 100% for everything.

7. Design and Integration

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

Most non-trivial changes we made to the application's code happened in the lead up to milestone 1, meaning they were triggered mostly by unit testing, regardless of whether those changes happened due to unit faults or otherwise.

While doing Test Driven Development for Milestone 2 we had to follow the style used by the tester and did not make design choices based on requirements which led to some inconsistencies with how the command would be implemented otherwise. We learnt how implementation structure can vary for TDD and Requirements based development and can change design of code.

Overall, we did not have to change the architectural design of the code because we found the initial source code design to be testable in terms of unit and integration tests and well implemented. Some design changes like displaying of errors thrown through exceptions should be handled by Shell's output stream instead of individual commands output stream were made after initial black box functional testing.

Did integration testing help you to discover design problems?

In fact, we believe that we caught most design problems before we reached the stage of focusing on integration. While integration tests did indeed help us identify certain issues we noticed that when compared to unit tests, they did not help us uncover nearly as many bugs.

Integration testing did not find any design or interface mismatch issues in our source code and did not lead to any major design changes. Most bugs found only led to minor bug fix changes in shell operators and argument resolvers.

8. Automated Test Generation

Bugs Found

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

For Automated Test Generation, we mainly used EvoSuite and SquareTest. EvoSuite generated many functional tests and helped us find existing bugs in addition to regression bugs when fixing the hackathon issues for milestone 3. By creating tests for all utility classes and manually analysing those, a couple of bugs in `MvArgsParser` and `ArgumentResolver` could be found.

The regression bug was related to having a semicolon operator inside command substitution(e.g. `echo` `` `echo hi ; echo sup` `` `bye`) and the order in which it was resolved. On making changes to fix other bugs, this functionality was changed and we could easily figure this out due to our generated automated test failing.

Hence, in total we could find 2 bugs solely related to unit testing (more details in MS2 report) and 1 regression bug as mentioned above. However, it must be noted that manual effort was still required to debug the fault and analyse the generated test cases.

Manual vs Automated Test Cases

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

Manual and Automated Test Case generation each had their own pros and cons. As mentioned above, we experimented with EvoSuite and SquareTest. SquareTest was not of much help to us because it mainly provided structure for tests and sandbox structure but the functionality still required manual effort. EvoSuite helped in generating functional tests with random inputs and covering many different types of possible inputs. The following are some analysis we made based on trying both manual and automated test generation:

- While generating automated test cases through EvoSuite took very little time, it required substantial effort in analysing the test cases and understanding what it is trying to test. Moreover, there are also duplicate tests which can be redundant, adding time to Continuous Integration.
- Analysing failing tests due to regression was easier for manual test cases as we knew what code they specifically tested and were more readable than automated test cases.
- Automated tests helped in quickly testing classes/units with random values of different types, making sure all input types are covered which might be tough to do manually and require more effort and are prone to human judgement.

Overall, we went with writing unit tests manually first and using automatically generated test cases to figure out what we have missed and add it into our test suite making it more robust.

9. Hackathon

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

In the cases where we agreed there was a bug, there were good useful points about the behavior of our application. It helped us identify a few corner cases we had not considered especially through creative inputs, and a few non-trivial design issues, such as the way `cp` and `mv` targets are interpreted.

However, the test code in itself was not extremely helpful. This is because, from one of the teams we only accepted 6 of 8 bugs reported and the other had a very specific test structure that was purely IO (or black box) based. In this case, the internal mechanisms of the tests were abstracted away. It was documented well so we could see what was being run and what came out but it was hard to integrate that structure in our usual workflow.

10. Debugging

Useful automation

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

Overall, the IntelliJ debugger was very effective in helping us to find bugs and debug them. While using it over time we were also introduced to advanced features which made our work faster and was very useful. Java Streams introduced in Java 8 were used a lot since they help make the code readable. However, the IntelliJ debugger did not provide a nice way to test these stream operations. We used a plugin when needed for this case called Java Stream Debugger.

For the initially planted bugs, since most of them were trivial, they could have been easily found by static code analysers like FindBugs and we would not even need to run the code. In production level code, it is also useful to have a live crash reporter which logs all backend operations and stores stack trace for errors. (e.g. Raygun Crash Reporter)

Improving coding and debugging practices

Would you change any coding or testing practices based on the bugs/debugging you encountered in the CS4218 project?

In terms of testing practices, although we did validation of tests early on, due to changing requirements and assumptions we did not continuously check our code against requirements. This resulted in a couple of bugs being spotted during the hackathon and so we should have focused on adding test cases which specifically tested the requirements and assumptions we had and also done manual validation of the system against the requirements.

While coding implementations, we started working on different commands at the same time, and since at an early stage we did not have a complete understanding of the software this led to some inconsistencies for some common functionalities like taking input via STDIN and hence caused different commands having different styles, which is the case for instance with `paste` and `cut`. To avoid this, we should perhaps have focused on understanding the project early on and making design decisions accordingly, based on the issues we could encounter.

Did you use any tools to help in debugging?

As mentioned above, we used tools like FindBugs, and Java Stream Debugger in addition to the IntelliJ debugger which helped in providing us more information while debugging.

11. Static Analysis

Did you find the static analysis tool (PMD) useful to support the quality of your project?
Did it help you to avoid errors or did it distract you from coding well?

PMD was useful for very specific kinds of errors such as broken null checks. A lot of the warnings are exaggerated so we did not use the tool continuously, but rather mostly at the end of milestones to ensure good quality for the “releases”. This also meant that it was not distracting as we were not too worried about it. Additionally, effort had to be put into suppressing irrelevant errors since they did not impact the project. With that said, there were also good changes we made to the code due to PMD such as preserving the stack traces of more generic exceptions when replacing them with our specific types of application exceptions.

12. Quality Evaluation

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

- Functional Checking: Comparing the behaviour with the bash shell: can even conduct user evaluations with people used to bash and see how their expectations match or are broken by the app
- Logs tracing for errors in production running code. We can give it to use for a set of users and let the logs be tracked and find out how often errors are occurring and track various metrics for each app, shell operator etc. We can also track metrics like Defects/KLoc and faults/hour.
- Formal proof of correctness for each application. Since all commands have a specified requirement and can be modeled we can use tools like SPIN, TLA+ to test using static slicing and code analysis using symbolic execution to verify if it behaves according to requirements and assumptions.
- Since this is an Object Oriented Programming based software, we can use SOLID principles to test the design quality or Arbeu's MOOD Framework (criteria are Method Hiding Factor, Polymorphism factor, Attribute Inheritance Factor, Coupling Factor, Attribute Hiding Factor, Method Inheritance Factor) or CK Metric Suite (criteria being Weighted methods per class, Lack of Cohesion in methods, Response for a Class, Depth of Inheritance Tree, Number of Children, Coupling between objects).
- Mutation testing with PIT or other tools with criteria would be to check the percentage of mutants killed.
- Generate a driver that issues random commands automatically from the grammar, then fire it up inside a sandbox (can maybe just use chroot) and log errors and crashes. Record the inputs that caused crashes, count them and analyse how likely they would be in real use. This is similar to random testing, but outputs are not validated, the only goal is to expose errors and input that crash the system.

13. Confidence

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

- Having a 98%+ statement coverage gives us high confidence in the quality of the project as it shows that almost everything has been tested in some way.
- Early effort put into designing test cases and what boundary values and inputs to choose helped in building a good catalog and providing good standardisation for all tests. Moreover, by assigning different functionalities to team members and having expert knowledge on those functionalities helped in ensuring they are well implemented and tested.
- Having Peer Reviews, writing/updating tests along multiple milestones of different levels like Unit, Integration, System and hackathon bugs help in ensuring good software quality.
- As seen in Section 3, over time fewer bugs were found until the hackathon and the severity of found bugs was also low which helped show that the quality of project was increasing.
- Having a standard way of writing and formatting our code tests allowed us to successfully collaborate and even work on other's code as tests made it easier to understand.

14. Surprises

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

Integration tests were not as useful as we expected. This may have been caused by different aspects. It could be that our unit test suite was thorough and included tests for all components of the system, such as shell operators, utility classes, therefore bringing up some integration bugs as well. It can also be that the characteristics of the application made it such that testing things like shell operators even in isolation would bring up integration problems. We believe these are the two main reasons. Finally, it could be that we did not approach integration testing correctly, but due to the amount and variety in terms of both focus and complexity of our integration test cases we think this was not the case.

15. Reflection

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

- Going into the hackathon with 97% statement coverage we were really confident that not many bugs would come up and yet we ended up accepting 25. While it is true that some of them were small details and even some due to OS specific issues, there were a lot of major errors pointed out, and this left us thinking about whether the coverage metric may have inflated our confidence more than it should. As mentioned before by Edsger Dijkstra , “Testing can show the presence of errors, but not their absence.”
- This project is a good example of an application that can take inputs with a huge variety of characteristics. Testing all of them is impossible so we really have to test smart, and in that way we go back to where the lectures started: the importance of practicing the ‘art’ of finding the relevant inputs, thinking about what could break, and under which circumstances.
- This module through its projects has highlighted the importance of testing in software development. It has brought out the importance of testing at each and every stage of the software. It has pointed out how despite having the requirements given to a developer, their personal bias can come in and result in major errors in the software. At the same time, it also sheds light on how developing and maintaining large, real-world production systems is not an easy task, as every change which you don’t think could have caused a regression, actually does the exact opposite.
- The importance of testing at scale was clearly highlighted through the project and over time with better test suites we could feel confident in our project and could make large scale refactor and changes easily by being able to trust our tests. Automated testing helped in quickly ensuring everything worked as expected and which is a must have quality for any production system. Over time as the project got bigger with more changes the maintenance and development was definitely a lot smoother due to regression tests. We learned that having a CI which runs tests before any changes are made and making sure to test all possible components in an effective manner is highly important for a production level project.
- We learned that by standardising our requirements and test and coding format we could make it easier to test during the project timeline. Moreover, peer reviews helped in preventing any developer bias and uncovered many faults.

16. Feedback

We have designed the CS4218 project so that you are exposed to industrial practices such as personnel leaving a company, taking ownership of other's code, geographically distributed software development, and so on. Please try to suggest an improvement to the project structure which would improve your testing experience in the project?

Testing applications that deal with file manipulation can very easily make a mess in the file system as it is very hard to keep track of all files created, moved, copied and deleted and provide ways to restore the file system to its previous state regardless of whether the test cases pass. We understand that this is part of the challenge, but we think it could be improved if it was made clearer for everyone.

Ideally we would have been emulating the file system but we could not add other dependencies. Mocking was too time consuming so also not the appropriate solution. To improve this, we suggest, the project could use one of the resources below, and/or give us a lab with detailed guidance on the subject, which would maximise the learnings of dealing with such a challenge.

- Google's Java in-memory file system: https://github.com/google/jimfs
- Apache Commons VFS: https://commons.apache.org/proper/commons-vfs/

Having a good Continuous Integration setup along with testing frameworks like NGTest and online coverage checkers like CodeCov would really help in improving the tester productivity as these offer a wide range of effective functionalities that are also used a lot in the industry.