MileStone3-QA Report

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

Code Type	Lines of Code (LOC)	Number of Files	LOC per file
Source Code	**5680**	**55**	**103~**
Test Code	**5063**	**33**	**153~**

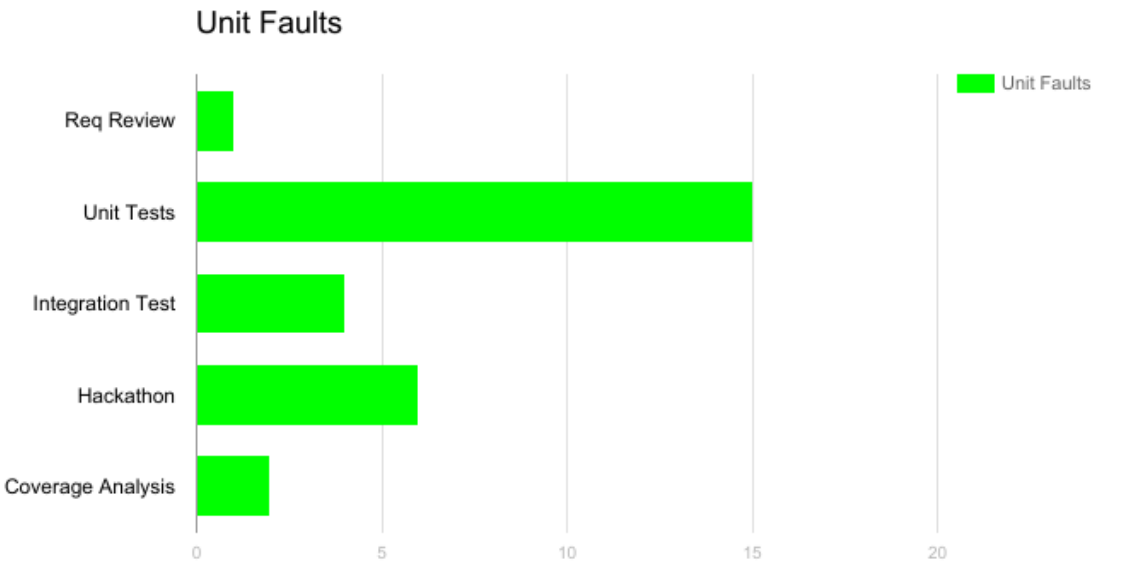
The total number of source code (LOC) is around 600 lines more than the test code (LOC). The difference in the line of code is not significant. However there are 22 more source files as compared to test files. The reason for a greater number of source files is due to the fact that there are additional interfaces hence, contributing to a higher number of source files.

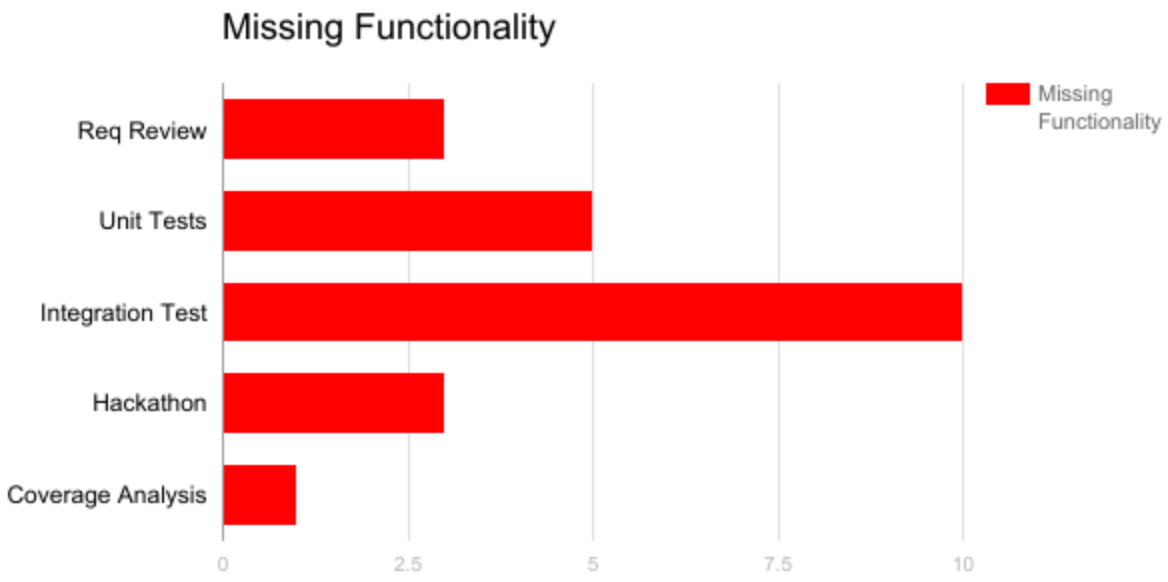
As testing is important it is not surprising to see that there is such a small difference between the source code (LOC) and the test code (LOC). It is also possible for the test code (LOC) to exceed the source code (LOC) if more test cases were written. The average LOC per file for test code is 50 lines more than the source code. This shows that on average there are more LOC written for test case as compare to source code.

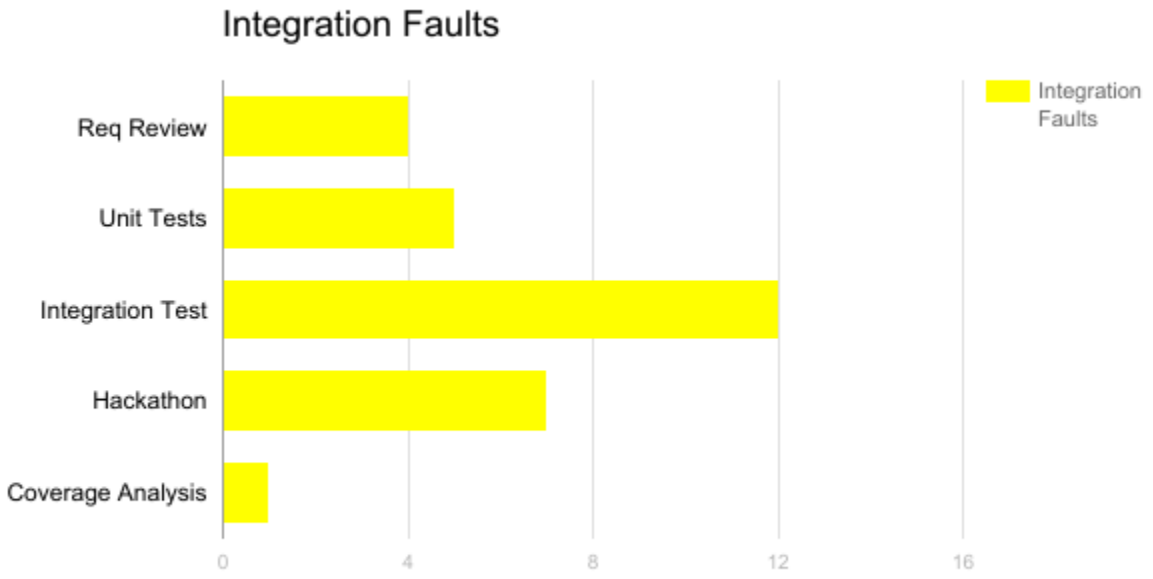
2. Analyze distribution of fault types versus project activities:

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

- **Types of faults: unit fault (algorithmic fault), integration fault (interface mismatch), missing functionality.**
- **Activity: requirements review, unit testing, integration testing, hackathon, coverage analysis.**
- **Each diagram will have a number of faults for a given fault type vs. different activities.**
- **Discuss what activities discovered the most faults. Discuss whether the distribution of fault types matches your expectations.**


Unit Faults
Req Review
Unit Tests
Integration Test
Hackathon
Coverage Analysis
0
5
10
15
20
Unit Faults

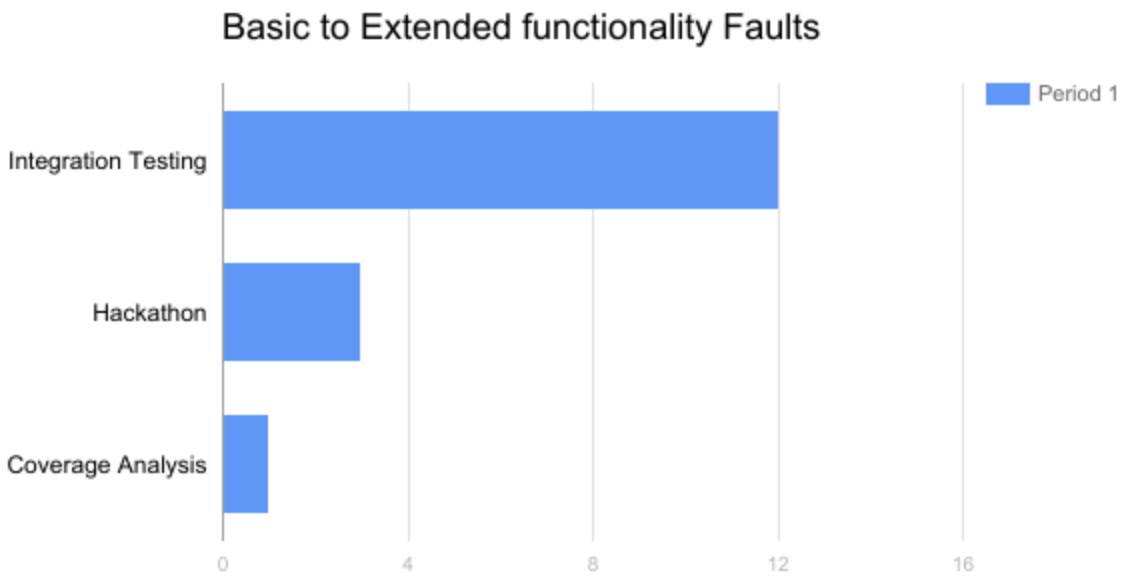

Missing Functionality
Req Review
Unit Tests
Integration Test
Hackathon
Coverage Analysis
0
2.5
5
7.5
10
Missing Functionality



From our analysis of the graphs, we observed that both unit and integration tests uncovered the most faults. These were expected as they test the correctness of the code and were crafted alongside the requirements of the project.

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

- **Activity: integration testing, hackathon, coverage analysis.**
- **Discuss whether the distribution of fault types matches your expectations.**



We observed that integration testing has always been the crucial component in finding faults in our program. Since extended functionalities include the pipe and other advanced functions, multiple smaller components are needed to work with one another seamlessly. The faults were easily identified when different components are put together, in integration testing.

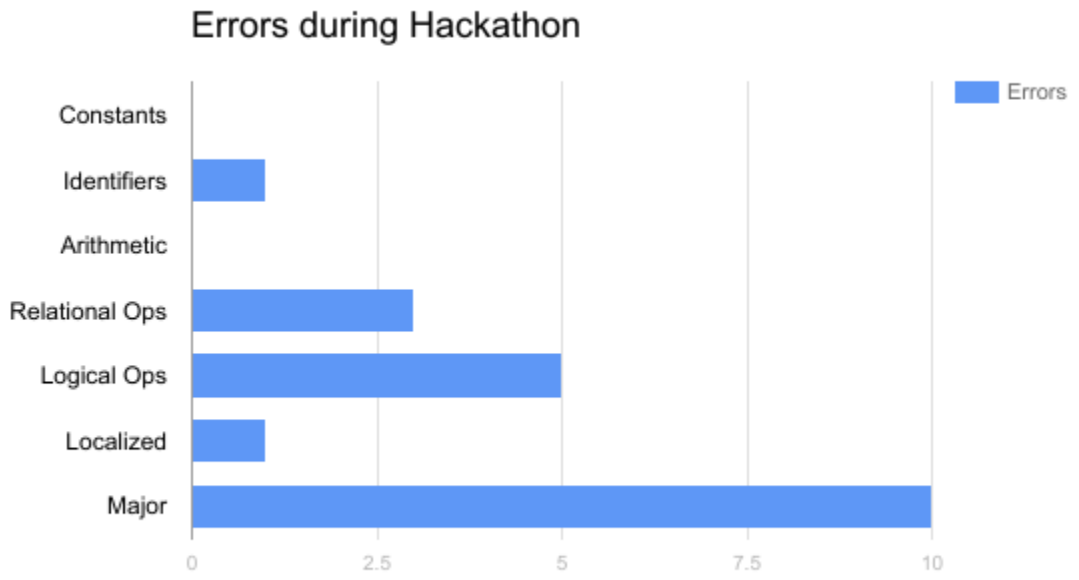
The distribution of fault types basically matches our expectation. As our overall coverage was not that high, the integration test had not covered all the conditions. As a result, the Hackathon will discover more faults in the process.

2.3. Analyze bugs found in your project according to their type.

- **Analyze and plot a distribution of causes for the faults discovered by Hackathon activity.**
- **Analyze and plot a distribution of causes for the faults discovered by Randoop.**
- **Causes: Error in constants; Error in identifiers; Error in arithmetic (+,-), or relational operators (<, >); Error in logical operators; Localized error in control flow (for instance,**

mixing up the logic of if-then-else nesting); Major errors (for instance, 'unhandled exceptions that cause application to stop').

- **Is it true that faults tend to accumulate in a few modules?**
- **Is it true that some classes of faults predominate? Which ones?**



We observed that there were many major errors, mainly not throwing exceptions. The fault lies only in a few modules as these modules were coded by a single individual. So it is more of a recurrent fault due to the way it is coded. Other faults were mainly logical operators where due to short circuit in evaluating them, the unit tests we made did not cover the rest of the logic.

We were not able to integrate Randoop, hence no diagram was plotted.

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

Requirements Analysis	Coding	Test Development	Test Execution
10%	**30%**	**50%**	**10%**

The requirement analysis of the project took around 10% of the time.

- 5% were spent on understanding the general project description.
- The other 5% were spent on assumption and seeking clarification of ambiguous description or undefined application behaviour.

The coding took around 30% of the time.

- 20% of the time was spent on coding the application.
- 10% of the time spent fixing bugs found during testing.

Test Development took around 50% of the time

- 10% of the time was spent on analysis the test case requirement and devising suitable test case for the application
- 10% was spent on unit testing
- 10% was spent on integration testing
- 10% was spent on system testing
- 10% was spent on correcting wrong test case or rewritten test case for bugs found in application

Test Execution took around 10% of the time

- 10% was spent on ensuring that the test done and the test results corresponds to the expected behaviour of the application

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

Test-driven Development	
Advantages	• Forces a good modular code design, for ease of testing. • Ensures the correctness of the program at all time, based on the unit tests written. Defects and bugs can be found earlier • The tests can serve as a kind of live documentation and make the code much easier to understand. • It can encourage us to think from an end user point-of-view.
Disadvantages	• Initially, it slows down development, as time is spent in writing unit tests • Sometimes we have to mock a lot of classes. It's beneficial in the long term, but might be time confusing and complex. • It is difficult to write good tests that cover the essentials and avoid the superfluous. • If the design is changing rapidly, test cases have to be changed too. Lots of time could be wasted writing tests for features that maybe quickly dropped.

Requirements-driven Development	
Advantages	• Ease of tracking progress of the project • Solid understanding of project scope and requirements

Disadvantages	• High reliance on project co-ordinator • Not as powerful in small projects where there is mainly just one developer. • Requirement may be constantly changing

Both TDD & RDD has its own advantages and disadvantages. For the project it seems that RDD was applied at the initial phase followed by TDD. We will say that TDD was a more prefer approach for this project as many assumption has to be made for the project, hence the RDD factor was not emphasize. Instead a TDD approach will be better as it will ensure the correctness of the application for this project

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

- **In particular, 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?**

a)

Smaller classes achieved the most branch coverage (eg EchoApplication, DateApplication) while larger classes had the lowest branch coverage. This is because larger classes are more complex and our unit tests did not achieve testing all possible branches.

b)

An important takeaway from this module that we learnt was that code coverage is simply one of the many metrics to estimate code quality. Just because we have a good branch coverage does not mean that the program we wrote is correct. For example, we could write a JUnit test without an assert statement at all! This could give us 100% branch/statement coverage but is of no use as there aren't any assertions. This is where mutation testing helps. Mutation testing would spot that too many mutants were not killed.

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

PMD test, it checked the code and showed us the violations contained within the project. It triggered us to change the design of our code.

Integration testing also helped us discover the design problems that have surfaced when passing the data between classes. Integration testing can discover the condition cases and more detailed exceptions that may be overlooked.

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

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

We were not able to integrated Randoop to our project, thus we were not able to find any new bugs with Randoop. However, we believed that the automatically generated test case using Randoop will help us find new bugs. Test case manually written by us may not cover all possible cases due to human error. Sometimes duplicated test cases could be written without our knowledge. Corner cases are usually not obvious to us hence we tend to miss it. On the other hand, Randoop had been proven to be useful in bug finding, hence the automatically generated test case should be able to help us to find new bugs.

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

The test case generated by the other team did help improve the quality of the project. The other team helped to discovered bugs in our project. These bugs were missed out during our testing. This shows that more testing has to be done and more test cases have to be written. While fixing the discovered bugs, we also discovered some other bugs in the process. The overall hackathon experience greatly improve the quality of our project as it encourages us to test our project more thoroughly

9. Debugging experience: What kind of automation would be most useful over and above the Eclipse debugger you used – specifically for the bugs/debugging you encountered in the CS4218 project?

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

Gitlab CI, as the project information is getting bigger and bigger, the unit test files created will also increase. It is hard for us to keep track of changes that affect the other modules or component whenever we implement or modify the code. Using Gitlab CI that does periodic test running and informing us whenever the tests fail will be convenient.

Yes, we will try to formalise the coding practice by using applications similar to issue tracker. It will greatly reduce our difficulty in getting the statistics and data for future reference.

10. Did you find 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 does help to support the quality of project by checking whether the project is following the coding convention and it enforces us to adhere to the coding convention. It is quite important for programmer to follow a set of coding convention as it will increase the readability and standardization of code. However, the PMD does not help to avoid errors as some errors were cause by the overall logic and design of the application itself. The PMD did distract us a little from coding as we were require to change some of our code to suit the coding convention.

11. How would you check the quality of your project without test cases?

We would perform Black-box testing as it helps to check the functionality of an application without peering into its internal structures or workings. This method of test can be applied virtually to every level of software testing: unit, integration, system and acceptance.

12. What gives you with the most confidence in the quality of your project?

When it comes to quality, we value the *correctness,* extensibility *and reliability* of the project.

Correctness: The utmost importance has to be given to the correctness to ensure the program works as what the requirement specifications states. Else, it would be considered as an unfinished product.

Extensibility: The architecture of the software should be well structured and documented so that when new features are required, the software could easily be updated and tested before releasing to the users.

Reliability: After we have tested for all the features and their functionalities it also very important that the application or product should be reliable and not break down due to stress or some inputs.

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

Black-box testing is definitely a good idea to check the quality of own project without test case. However, it may be a very hard if the project belongs to others. Not having knowledge of the internal structure of the project requires the testers to provide a huge number of inputs and valid outputs for comparison.

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

We have learn that no matter how sufficient we think we have done our testing, bugs may still be present in our project. A software application is never 100% bug free. It is important to come out with a test plan first before conducting any testing. A good test plan will list down the details of what types of test should be conducted. Having a test plan will give us time to think through what we want to do and provide an overview of what to do. The test plan may not be able to provide 100% coverage but at least it will ensure that test conducted is able to provide sufficient test coverage for most of the application.

15. 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?

Project description and instruction should be made clear. Throughout the whole project the word "assumption" was used constantly. "Assumption" should be stated clear at the start of the project as part of the project requirements, similar to requirement-driven development. The stating of "assumption" halfway through the project lead to quite a few confusion among teams, especially during hackathon when different team have different assumption of the functionality of application.

Guidelines for hackathon should be well defined. During the hackathon, my team followed the instruction defined before the hackathon that we should adhere to the project discussion stated in the IVLE forum thread and thus we defined our bug base on the forum discussion. However the other team argues that they stated the behaviour in their assumption and the forum thread was constantly updating thus it was unfair for them to keep track of the changes. In the end our bug was deem invalid as the other team argument was accepted.
In "Lab7-Hackathon.pdf" page 7, the definition of the bug "conforms discussions in the IVLE forum" was stated before "does not violate developer assumptions explicitly stated in the source code comments" so shouldn't the discussion in forum had precedence over assumption stated?

Improvement to the hackathon can be made by allowing TAs and both team to discuss together their respective "assumption" for their project before carrying out the hackathon. Another option is that assumption should be declared clearly in a pdf instead of the comment of the code. This is because not all team uses the interfaces for the project, hence it will be unfair for teams to go look through the whole code to locate the comments.

The current hackathon format has flaws as both team worked on the hackathon based on their respective assumption. Even if there was a lab session for us to clarify our assumption, the TAs perspective may be influence, as he/she only hears one-side of the argument and thus may be convinced that the bug was valid. However there may be a more convincing argument provided by the other team during the rebuttal thus making the bug invalid. So it is very unfair for the team who thinks they found a bug and clarify with the TA but in the end get penalised because the other team had a more convincing argument. I don't think we are trying to hone our debating skills here.