Software Quality Assurance (QA) Report

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

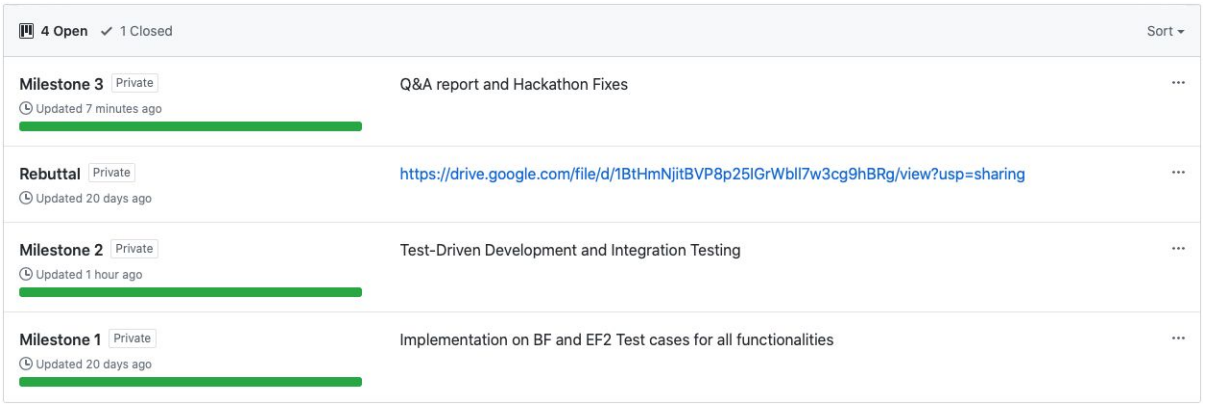
Source: 1519 LOC
Test : 5456 LOC

Code	Source code		Testing code	
	Original	Final	Unit	Integration & System
Lines Of Code	4,158	5,677	4,420	716 + 320
	1,519 addtions		5,456 in total	
Percentatge	21.78%		78.22%	

2. Give an overview of the testing plans (i.e. timeline), methods and activities you have conducted during the project. What was the most useful method or activity that you employed?

The most useful testing method we use is Test-Driven Development, which reveals lots of bugs and lacks implementation. The most useful testing activity we employed is Hackathon and Rebuttal, which provides us corner causes that we do not hold.

Also the upper figure shows the commit frequency every day from start until now. It roughly shows our development contribution frequency day by day.



For different milestones, we conducted different types of project boards. For Milestone 1 and Milestone 2, we used Automated Kanban, which has built-in triggers to automatically move issues and pull requests across To do, In progress and Done columns. For Milestone 3, we used Bug Triage, which triages and prioritize bugs with columns for To do, BUG pending, NEED assumption and Closed.

3. Analyze the distribution of fault types versus project activities:

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

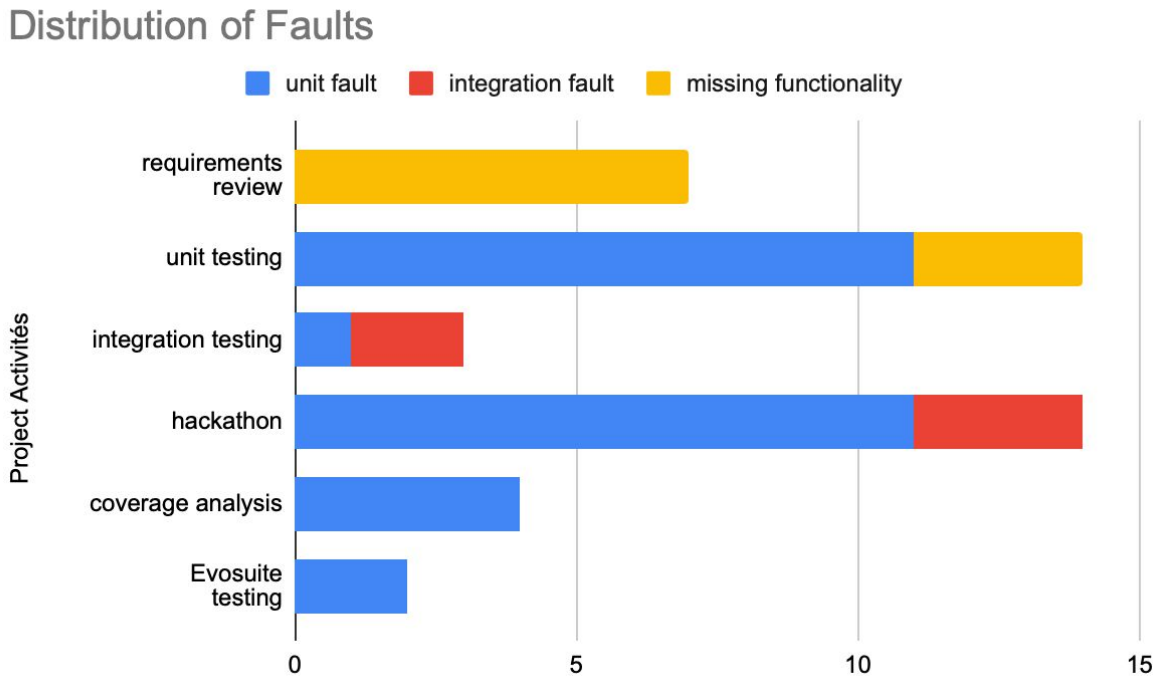
Types of faults: unit fault (algorithmic fault), integration fault (interface mismatch), missing functionality. Add any other types of faults you might have encountered.

Activity: requirements review, unit testing, integration testing, hackathon, coverage analysis. Add any other activities you have conducted.

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.

fault types	requirements review	unit testing	integration testing	hackathon	coverage analysis	Evosuite testing
unit fault	0	11	1	11	4	2
integration fault	0	0	2	3	0	0
missing functionality	7	3	0	0	0	0



Unit testing and hackathon discovered the most faults, requirements review also found quite many missing functionality.

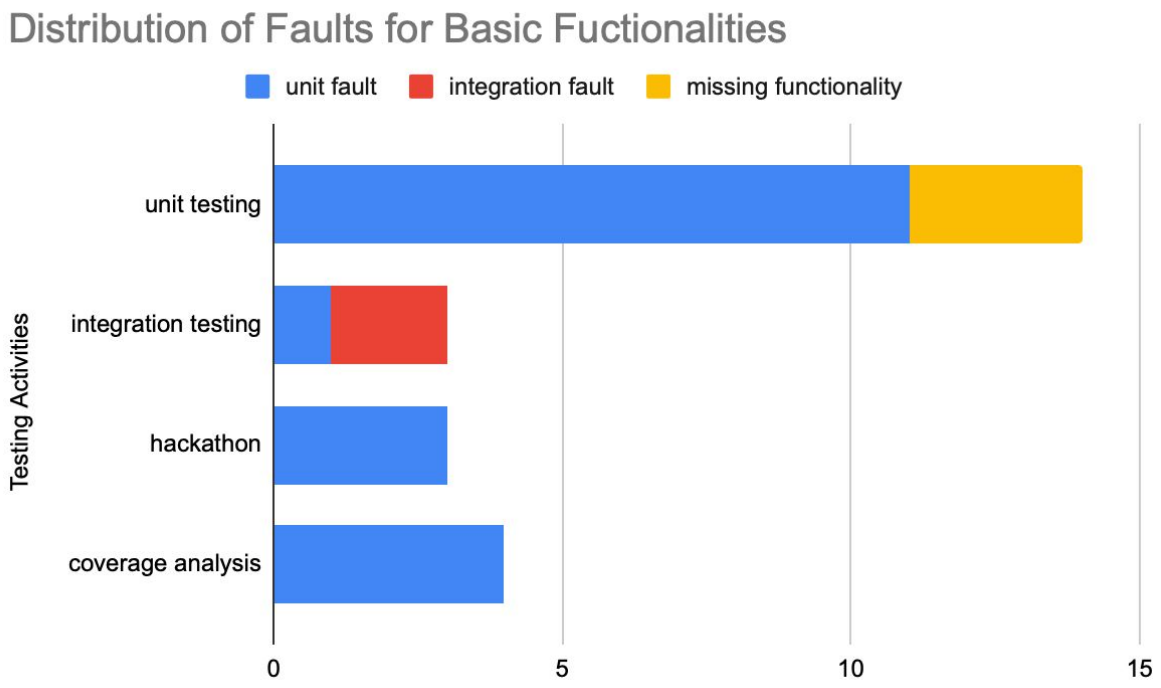
This distribution of fault types matches our expectation except in hackathon activity, in which we didn't expect so many bugs to be found. We realized that this mismatch was due to our misinterpretation of the requirements of some applications. For example, for the DiffApplication, we didn't consider binary files; for the MvApplication, we mixed reversing the meaning of the flag "-n".

3.2. 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. Add any other activities you have conducted.

Discuss whether the distribution of fault types matches your expectations.

fault types	unit testing	integration testing	hackathon	coverage analysis
unit fault	11	1	3	4
integration fault	0	2	0	0
missing functionality	3	0	0	0



The distribution of fault types matches our expectation. Most bugs are found in the unit testing activity, indicating that our test suite was comprehensive. Also, the correctness of the whole program depends on the correctness of each part of it. So, the more bugs we found in unit testing, the less bugs we shall find later in integration testing and hackathon.

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

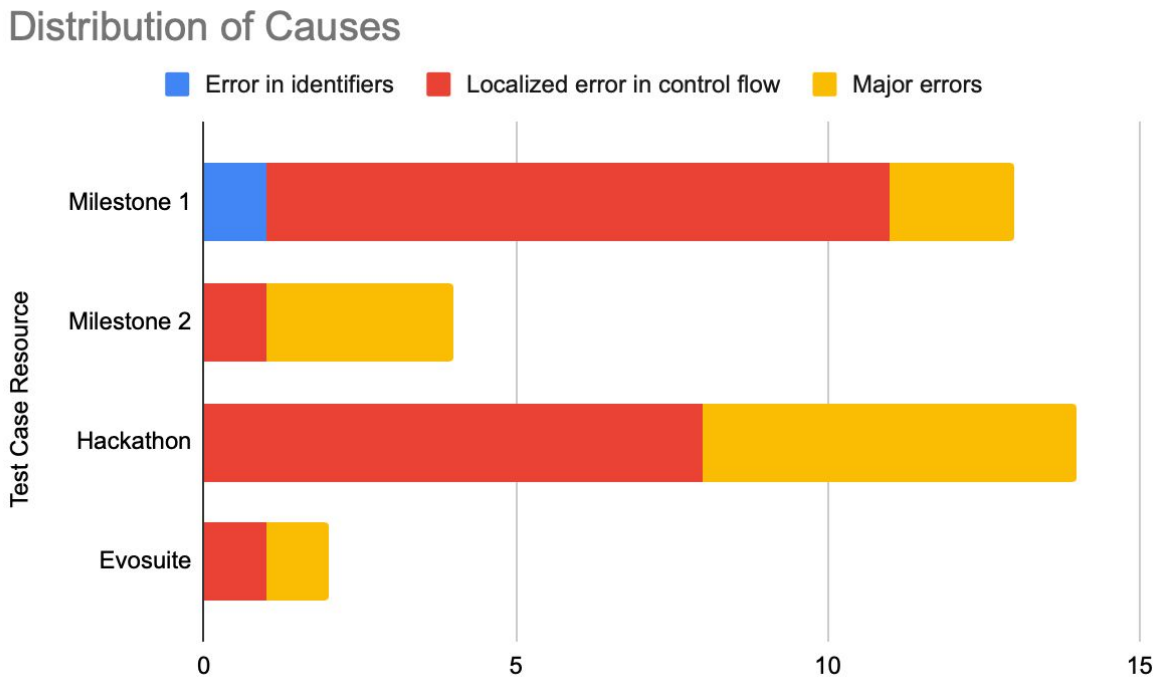
Analyze and plot a distribution of causes for the faults discovered by Hackathon activity.

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

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'). Add any other causes you might have identified.

Is it true that faults tend to accumulate in a few modules? Explain your answer.
Is it true that some classes of faults predominate? Which ones?

Cause types	Milestone 1	Milestone 2	Hackathon	Evosuite
Error in constants	0	0	0	0
Error in identifiers	1	0	0	0
Error in arithmetic	0	0	0	0
Relational operators	0	0	0	0
Error in logical operators	0	0	0	0
Localized error in control flow	10	1	8	1
Major errors	2	3	6	1



Yes. The faults tend to accumulate in a few modules for the bugs found in the hackathon period. For example, for the several bugs of MvAppliciation, we incorrectly handle the exception. However, it causes more than one bug. And only modifying very few lines could solve all the related bugs.

Localized error in control flow and major errors are the most two frequently appeared faults. It is easy to forget to handle some types of exceptions. And the logic of some applications are pretty complex, mixing up the logic in control flow is very common. These will cause faults.

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

- requirements analysis and documentation 15%
- coding 30%
- test development 40%
- test execution 10%
- others 5%

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

	advantage	disadvantage
TDD	In TDD, detailed specification is explicitly defined in the test cases, which makes the faults easy to be located by the failed test cases. Also, by covering all use cases and edge cases in the test suite, it gives us confidence on the implementation and further modification. It also helped us to really understand the requirements and our code by figuring out concretely the expected inputs and outputs.	It takes a large amount of time to write a test suite with good coverage, hence slowing down the development at the beginning. The test suite itself has to be maintained, some tests would have to be rewritten if the requirement is changed or the code is refactored.
RDD	The functionality of the application can be developed first, allowing the user to test the product as it develops, which is good in agile development.	The quality of the implementation is not guaranteed. Bugs not caught in the early phase of development may become hard to locate and fix, increasing the cost overall.

6. 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?

We get 100% line coverage for classes CdApplication, EchoApplication, ExitApplication, ArgsParser, DiffArgsParser, LsApplication, RmArgsParser, ApplicationRunner, CommandBuilder, IORedirectionHandler, StringUtils.

The classes with least line coverage are CpArgsParser (82%), SequenceCommand (78%), IOUtils (89%), RegexArgument (79%), CpApplication (80%), DiffApplication (89%).

The coverage difference is among 10% to 20% on average, which is not significant, indicating that our test suite is comprehensive.

There is some relation between the line coverage and code quality. The classes with low line coverage tend to have more bugs than the classes with high coverage.

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

The Integration Testing and System Testing triggered us to change our design, with comparing the results with a real Linux Shell, we get a better understanding about our deficiencies and our limitations.
Integration testing did help us to discover design problems, especially problems with IORedirection, CommandBuilder, ApplicationRunner and Globbing etc.

8. Automated test case generation: did automatically generated test cases (using Randoop or other tools) 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).

Yes. It helped. We use evosuite to automatically generate tests. It helps us find some boundary related bugs. With the help of them, we implement our functionalities more completely.
Compared with manually writing tests, generating and analyzing tests are more time efficient. However, the quality of the automatically generated test suite can not always be guaranteed, some of the test cases we need may not be covered in it and have to be written manually. And it also costs time to set up the tools. Therefore, we think the manual effort of the two methods is not much different overall.

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

Yes. The test from other teams helped us find more bugs and make us acknowledge that our assumptions are not sufficient.

10. 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 project?

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

Did you use any tools to help in debugging?

We only use the debugger provided by intelligent ideas. To localize the bugs, we may add more specific tests to narrow down the part which may be wrong.

For example, when we found the command “echo “hello” | grep hel; rm test.txt " crashed, we may add more simple test like “echo “hello” | echo hh ”, to localize this bug.

From this experience, we decided to write unit tests and implement at the same time as possible to ensure the sub-model we will use is correct..

11. Did you find static analysis tool (PMD) useful to support the quality of your project?

Yes, PMD is useful to support the quality of our project.

Did it help you to avoid errors or did it distract you from coding well?

It did help us avoid some potential problems, such as unclosed inputstream or outputstream. Also, it helps us avoid some duplicate code fragments.

12. 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.

Line Coverage: 86 % class, 88 % lines coveraged
The coverage gives us confidence for our tests. When writing tests, we also consider some boundary conditions that make our tests more comprehensive.

PMD warnings: no PMD warnings in our implementation
The implementation is without PMD warnings , which shows that our code has a good style and is consistent with coding standard.

Java doc: comprehensive and detailed Java doc in our implementation
We have written detailed Java docs in all the necessary place in our implementation, making our code easy to understand and maintain.

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

Code Coverage and number of tests.

The more test cases we generated and passed for a class, the more confidence we have in the quality of the class. For example, we have many test cases in FindApplicationTest, PasteApplicationTest, SedApplicationTest, CutApplicationTest, and all of them achieve very high code coverage. Finally, seldom faults were found from the classes that correspond to these test classes.

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

The answer of question 8 is counter-intuitive to us. At the beginning, we thought that automatic test case generation can effectively discover many new bugs that we fail to discover. However, after we evaluated the test cases generated by evosuite, we realized that only a few test cases generated by evosuite can be used by us. We have to agree that automatic test case generation is a good complement for manual effort but it cannot take the place of it.

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

We used to think testing is the least important part in project development. And holds a little bias on the testing engineer. However, through the CS4218 project, we learned how important testing is and how it improves our developing efficiency and ensures our code quality.

16. 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?

1. Docker environment. In case of testing some dangerous commands and the hackathon.
2. Specified testing files standard.
3. Please ask us to do this quality assurance report thorough the whole testing process, but not do it at the end. It is difficult for us to summarize what kind of faults that we have discovered if we did not record them before. Especially for question 3.