CS4218 Quality Assurance Report

Group 15

Cao Shengze

Kim Hyung Jon

Summary for the Hackathon bug fixes:

All bugs found in the Hackathon have been fixed. All new test cases have been passed. The test file is Hackathon.java and we also added these test cases into relative unit test module and integration test module.

1. How much source and test code have you written?

Source code: About 4700 lines of code, with around 10 interfaces and 30 classes. 150kb

Test code: About 6700 lines of manually generated code, and 40000 lines of automated generated code by Randoop. 400 kb of manually generated test files and 2MB of automated generated test files by Randoop.

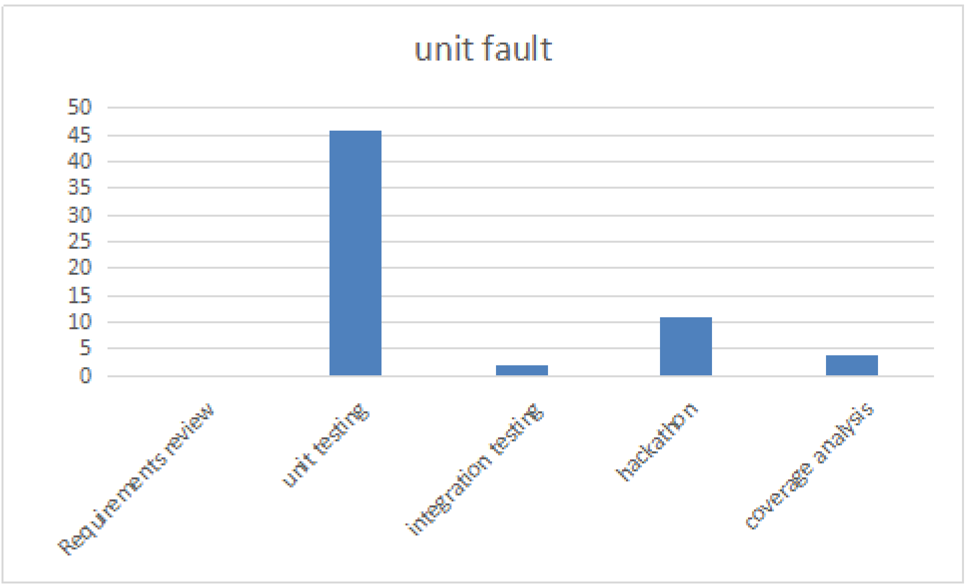
Around 500 manually generated test cases and 1000 automated generated regression test cases by Randoop.

2. Analyze distribution of fault types versus project activities.

For this section, we just count the number of bugs that we still can remember when reviewing the test cases and the github log. It may not be exactly same as the real faults: it might be slightly fewer than the real faults, but the difference is not big.

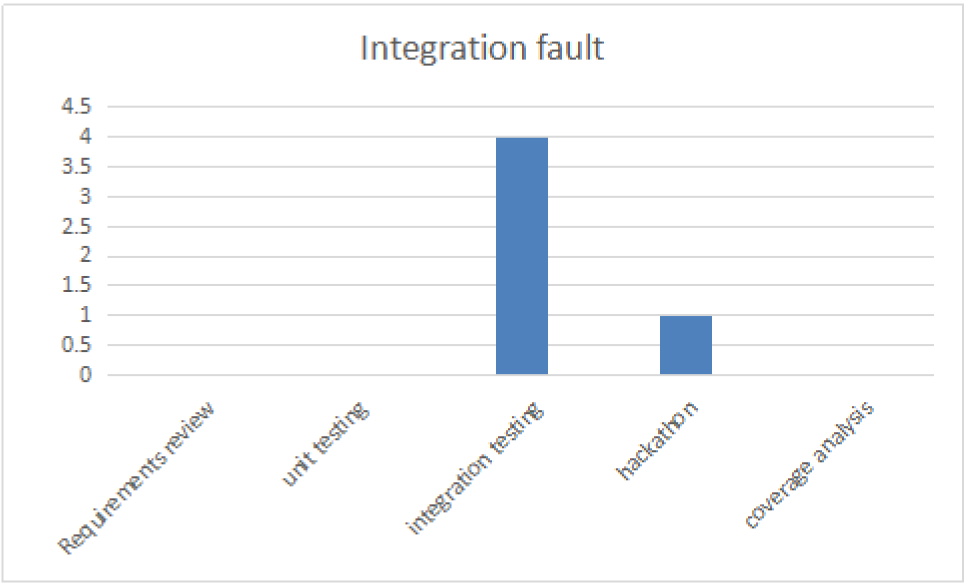
For the requirements review part, some are from the hackathon, therefore some faults count twice: once in the hackathon, once in the requirements review.

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



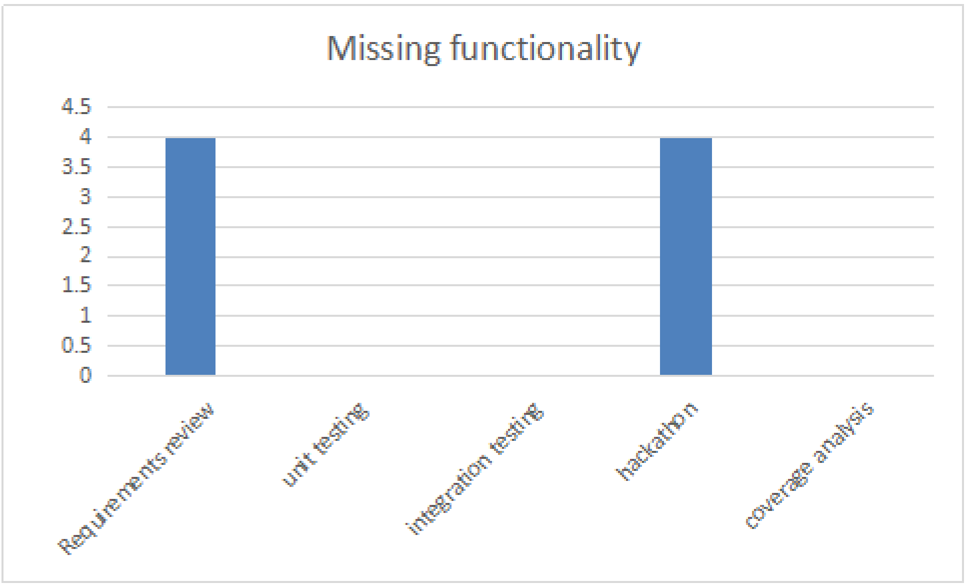
For the unit faults, surely the unit testing has found most number of faults. As all the other three parts are after the unit testing part, by the time they were conducted most bugs have already been fixed in the unit testing stage.

Therefore, the distribution does match our expectations.



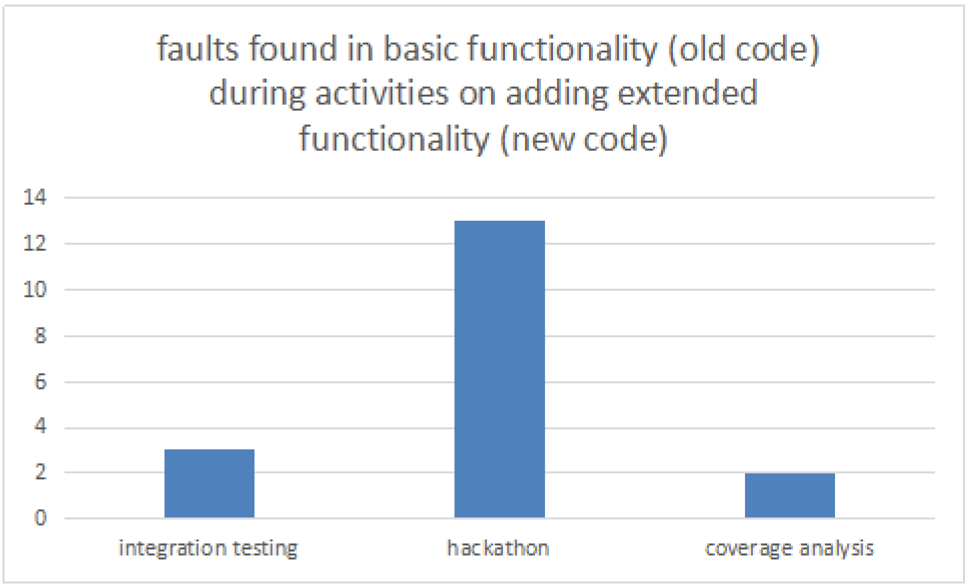
For Integration fault, there are quite few faults found, as we already test the interfaces during the unit test. The only integration faults are because of the shell functionalities. Some bugs appear when using multiple shell functionalities (quoting, globbing, pipelining and so on) together. Actually, these bugs can also be viewed as unit bugs. But as these bugs also appear only when these features appear together, therefore, we considered them as integration faults.

The distribution also matches our expectations.



For the missing functionality, the Hackathon and requirements review contributed most found faults. This is a natural outcome since unit and integration testing were written based on our understanding of the requirements and coverage analysis is unrelated to whether certain functions are implemented. Only by reviewing requirements by our or another group were we able to find bugs of the type missing functionality.

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

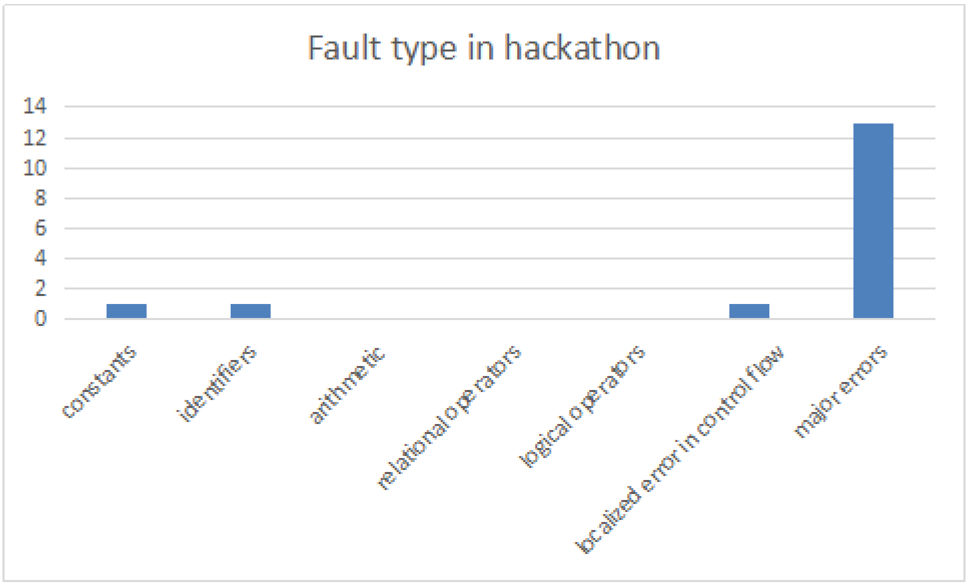


From this diagram, we can see the integration testing and coverage analysis contributes quite few while hackathon contributes the most.

This result more or less matches our expectations. Hackathon works the best since it is evaluation by another group that have different vision about the requirements, which makes them able to discover bugs in areas we did not think about. Coverage analysis would be the least helpful because although they show what components are not well-tested and what parts of a component need more rigorous testing, they do not necessarily lead to finding bugs, especially if the untested area had no bug.

We expected our integration test to uncover more bugs, but higher than coverage analysis and lower than Hackathon is still our expected result.

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

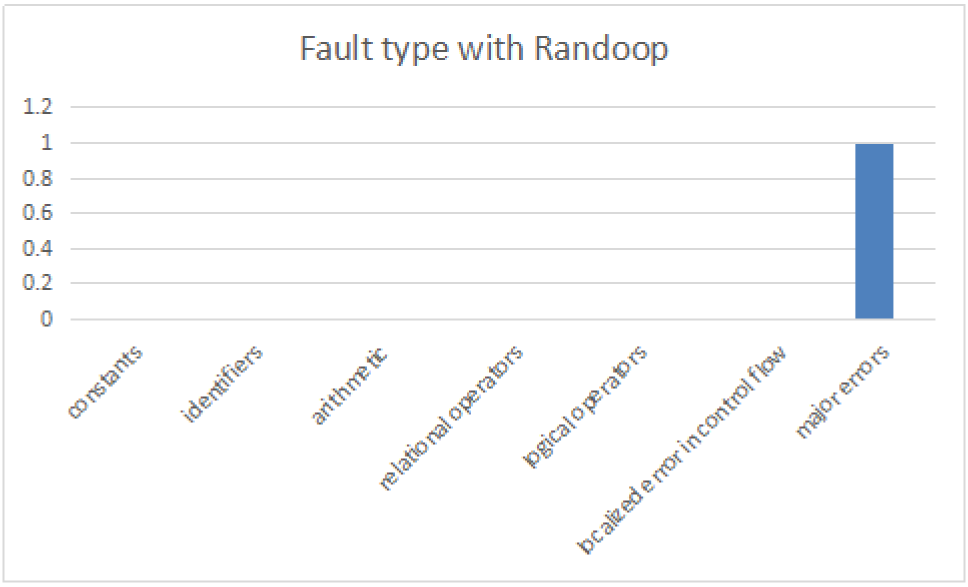


Is it true that faults tend to accumulate in a few modules?

Yes, most errors were found in the head, tail and the quoting functionality.

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

Yes, the major errors was the predominant types. The reason is that the faults found in the hackathon are mainly because other groups consider some scenarios that we do not ever think about. In other words, the faults are always in some corner cases. The first several fault types have been prevented before the hackathon because of our own test.



As mentioned in milestone 2, Randoop only found one error, which was an unhandled exception.

Is it true that faults tend to accumulate in a few modules?

Yes, this error happens in the SortApplication..

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

Yes, the major error one.

The Randoop actually cannot help with the first several types of faults as it would assume and treat the current behaviors as correct behavior. The Randoop would mainly work for the situation that the developers forget to handle some potential exceptions.

3. Provide estimates on the time that you spent on different activities.

Requirements analysis: 10%
Coding: 30%
Test development: 35%
Test execution: 25% (including fixing the bugs revealed by the tests)

Analyzing the requirements normally did not take a long time, since the project descriptions were clear and we were allowed a large degree of freedom on exactly how our program should behave for unspecified parts.

A small part of the requirement analysis was spent on understanding the input format. This was a short but very important step because it would determine our program's robustness. The requirement of each application and Shell is that it should 1) perform input validation, 2) if invalid, it should correctly detect the cause, 3) if valid, it should correctly categorize the input so that the input goes through the correct execution path and the correct output is produced. For this purpose, we first set some assumptions for the input and wrote a set of rules that the input should follow. Based on these, we created a pseudocode for input validation and all cases to be tested (valid, invalid, where corner cases are, combinations).

For coding, a large portion of the time was spent on writing the core functionalities. Input validation and categorization did not take a long time, although some applications had complex input handling logic. On the whole coding took a long time.

Writing tests, whether before or after coding, took a long time because in order to test our program's robustness we had to write tests for all possible kinds of input. We systematically wrote tests with that are valid, invalid at some points, invalid at many points, have a mixture of valid and invalid parts, and so on. This was a very tedious process and in some cases took much longer than coding the component under test.

After writing the tests and running them, we had to deal with unexpected failures and assertion failures. First we identified whether the mistake was in the code or the test, and if the cause was in the code we tracked down the precise location of the bug and the kind of input that caused the failure, and tried to fix the bug. Although we sometimes had difficulty figuring out why our program was failing, the process did not take a very long time.

4. Test-driven Development vs Requirements-driven Development

For the Test-driven development, the advantage is that all the details would be emphasized at first and we would not miss them during the development. We do not need to worry that we may miss any corner case as all of them would already be in the test cases. We can just simply check by running the test cases.

However, the test-driven development can also have some disadvantages. The quality of the code would depend on the quality of the test cases. If the test cases have some problems, it may lead to a fault in the code as well. Once we pass all the test cases, we may not reflect on these test cases anymore and there might be some potential faults at the end.

For the requirements-driven development, the advantage and disadvantage would just be opposed to those for the Test-driven development. The advantage is that since we would follow the requirements to develop the code directly, it would be quite unlikely to have potential faults as long as the requirements are correct.

The disadvantage is quite obvious: we may not consider all the cases when we code and miss some cases. And when we design the test cases, we may be influenced by our code, and therefore generate some wrong test cases due to our biases caused by our wrong code.

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

We would say that there is a moderate correlation between branch coverage and code quality. Having a test case that covers most or all branches definitely helps improve our code, but it is not a strict requirement, nor does it guarantee good code quality.

Out of the 15 classes we have, not counting exception classes and all tests, the two classes with highest branch coverage are SedApplication and GrepApplication at 100% coverage. However, during Hackathon the other team found bugs from SedApplication, while no bugs were found from SortApplication that had coverage of 92.7%. Similarly, the least covered classes are Environment and CallCommand classes, at 57.1% and 72.3% instruction coverage respectively, yet no bugs were found from these classes.

This happens because tests are written based on our understanding of the software requirements and expectations of what inputs might be used, which may not be complete. Therefore, even if we wrote many good tests and achieved very high coverage, there could be some input types we did not account for because we never expected them, which can cause our program to act in unexpected ways. Hence, we conclude that although good tests with high coverage can help us detect very obvious bugs, it is not enough to guarantee that our code quality is good, and a good understanding of the requirements is also necessary.

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

Our development was quite smooth, so there were no events that we had to make major changes to the code we already wrote, even when our code was reviewed by another group. However there was one time that we had to question our code design.

After we started on Milestone 2, we followed the TDD approach and then proceeded to actually implement the code. At this stage we realized that because we started requirements-based development for Milestone 1 without much knowledge of how the functionalities for Milestone 2 should be implemented, the code structure of our Shell may not be very compatible with what we need to do for Milestone 2.

Fortunately, there were just a few tweaks required to make Shell compatible with the new features needed. However, we learned a lesson that working with an incomplete understanding of the requirements of a project can lead to disastrous cases where we have no choice than to completely overhaul the code. This could happen when joining a project in the middle of it, picking after what the previous employee was working on, etc. We learned the importance of understanding the project requirements fully, even if they are not required immediately.

Did integration testing help you to discover design problems?

Integration test did not help find many design problems. A majority of the tests we had prepared passed without trouble, and those problems that caused failure either did not require major changes to the design or were not exactly design problems but more superficial issues (e.g. PipeInputStream unable to handle large contents).

7. Did automatically generated test cases (using Randoop) help you to find new bugs?

Yes. However, only few bugs were found by Randoop. Same as what we have in the milestone 2, the Randoop has quite a limited ability to find out errors. It would have no idea what are the actual requirements towards the code we write. It just assumes what we are doing, the current behaviors of the code are correct. And the test cases generated by the Randoop would also be more suitable for regression test instead of error revealing test.

The errors we found by Randoop actually is quite trivial, usually exceptions that we forget to handle. However, this exception would be quite unlikely to happen because of the context.

Compared with the automated generate test cases, the manually generated ones would be more likely to find out bugs. That is as a coder, we can always come up with some complicated test cases purposely. And even the automated generated test cases are easier to generate, compared with manually generated ones, it is not so powerful.

However, the automated generated test cases are still meaningful as a complement to the manually generated ones.

8. Did test cases generated by the other team for your project help you to improve its quality?

The test cases of the other team were not very useful for our project, mainly because of different expected behaviors. Small differences like output format could be easily modified, but sometimes there were large differences, especially in the input structure. For example, our group assumed that all arguments starting with the character '-' should be considered as options (which is not true for actual Linux Shell, but we did not know this) and the rest as filenames, while the other team checked for each argument whether it is a filename and treated the remaining as candidates for options. Similarly, both our group and the other group implemented some application interfaces to take in the contents as parameter, instead of the arguments, and there were differences in which interfaces the two groups did this. As a result our unit tests for these applications were incompatible. These issues were due to the ambiguous project descriptions, leading to different ways our programs can be implemented. We are not sure whether this ambiguity is intended (e.g. for simulating the actual work environments), but it seemed to have limited the extent to which other teams' tests can be used to improve our project.

Still, we did find a number of bugs thanks to assertion failures caused by the other group's tests, and the bugs discovered by the other team also helped us improve our program's code quality.

In summary, the other team's test case help us improve our code by:

- Raising some corner cases that we cannot handle properly.
- Raising some features posted in the forum that we did not notice previously.

9. What kind of automation would be most useful over and above the Eclipse debugger you used?

The automation features which we find most powerful about the eclipse debugger are the breakpoints and the watcher. The breakpoints would help me to locate to the place which we think the faults more likely to happen. And with the watcher which can observe the values of each variables, we would be then able to observe the variable which has something wrong and find out where the fault happens.

However, we would not actually change the coding practices based on the bugs/debugging we encountered on CS4218 project. We have already been using eclipse for a long time and the debugging skills are not quite new to us.

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 did not prove particularly useful for our project's code quality, mainly because the rules set for our project were already the habits we were following. However, we could see why at least some of the rules stated in PMD would be important.

For example, we had a habit of not using curly brackets when a condition branch consisted of a single statement (e.g. if (found) return 1;) This is forbidden by the PMD rules we were required to follow. We observed that although this is not problematic by itself, if we were to add some more lines later (e.g. "this.found = true" before the return statement) and forget to wrap the code block in curly brackets then there would be a bug. By following the PMD rules, we were probably able to avoid such bugs. Also, following a single set of rules for writing codes would help understand codes written by other teammates or other group's code, just like coding conventions.

There were some rules we could not understand, such as needing to check equality to string in the form if (“”.equals(output)) even in the code and not just the tests. Still, these did not distract us from writing codes well. On the whole we believed that PMD generally helped us write better codes.

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

Without test cases we would resort to random testing, by passing commands directly to Shell. This was often done during the early stages of developing applications, where we wrote single functionalities one by one and checked if their basic behaviors were working as intended.

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

Achieving 100% test pass, with the Hackathon results gave us the most confidence.

As explained above, our tests were mainly focused on proving our program’s robustness by making sure all possible inputs go through correct execution paths. We extracted the assertion statements to outside the try-catch block so that we would avoid false positives and false negatives. (i.e. if the program was supposed to run correctly but it threw exception, if the assertion was inside the try block the test passes because the assertion is never executed. Likewise when the program was supposed to throw exception but it does not.) If we put in valid input, we compared the expected output with the actual output. If we put in invalid input, then we checked whether the error message was pointing out the correct cause. This way, we had high confidence with our program’s robustness.

But there were input types we did not think of and hence could not handle, and the Hackathon pointed out some of them. This made our understanding of the input format more complete, and we were able to make our program even more robust. Therefore, our tests and good results of the Hackathon gave us the most confidence with our project.

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

The answer to question 6, part 2 was the most counter-intuitive result. Based on past experiences with integration testing in software projects, before we started our integration testing we expected that although individual components worked well the integration test would probably expose a large number of failures in the ability of the components to work together. In reality, we found very few failures, and none required major changes to our code. We even tried very unconventional tests, such as testing command substitution with a file containing a name of another file. What this does is that a file-reading application such as cat or head will read the file A and output its contents, which is the filename of file B. This filename would be passed as argument to another file-reading application such as sort or wc.

The expected behavior is that the second app will behave exactly like when we just passed the filename of B to the app. We anticipated that such a test would fail, but it worked perfectly.

We are not sure exactly what caused our integration testing to work much better than we expected, besides the fact that we did unit test before integration testing as noted in questions 1 and 6. Our code might just have happened to be good. A more reasonable explanation might be that this project is a special case because the components are almost entirely independent. The Shell acts as the command center using pipelines and command substitution to transfer data, and no application directly calls another application, so integration test worked no better than doing unit test on Shell. This may be an explanation, but nonetheless we did not expect integration testing to uncover so few bugs, therefore we consider this to be the most counter-intuitive answer.

14. Describe one important reflection on software testing or software quality that you have learnt through CS4218

For CS4218 project, we think the most important lesson was: remember that bug-free software is impossible, and always do what is necessary to keep discovering and fixing bugs.

I think the reason that why CS4218 chooses the shell as the project is because the shell command can be complicated. And it is really hard to find out all the bugs and make a perfect one. Actually that is quite true. Before the hackathon, we have tried our best to make our program robust in the first place and then improve it even more by finding all the possible bugs we could think of. At that stage, we thought there should be very few, possibly zero, bugs existing in our program. However, during the hackathon, there were still 16 bugs found. Some were related to the project requirement and some are the special corner test cases that we have never thought of. And even after fixing those bugs, we still cannot say our project is bug-free.

Therefore, we were reminded of Dijkstra’s quote that tests can only prove the presence of bugs, and not their absence. We learned that “bug-free software” is the ideal that can never be reached, yet we should always struggle to reach towards it by improving our understanding of the requirements, writing more tests, having the program evaluated by others and so on.

We also reflected on the software project we would work on in the future, in the industry. Software projects in the real industry would be much more complex than the most complex projects that we work on in NUS. And the projects run in real lifes can have much more interactions with the outer environments. Finishing the features is only the first step. One must try to write many tests to prevent the accident in the real usage. Once the program crashes in the real environment, it may cause large losses. In order to prevent such disasters, it is the programmer’s responsibility to ensure the robustness of the program. Testing also plays a very important role in the real industry. In short, we have learnt to treat the program more seriously, with responsibility.

15. Please try to suggest an improvement to the project structure

In general, we observed that one team's test cases would be very incompatible with another team's, due to the amount of freedom we have when building our requirements. Although tests should not be completely compatible because that is unlikely to be possible in an industrial situation, we felt that the incompatibility was so severe that testing with another group's test cases was very unhelpful, as discussed in question 8. If the project can be adjusted so that the test compatibility is better, it might be more helpful for our juniors.

The interface methods expecting the parameters as the args combined back to a string is another restriction that we are negative about.

Also for Sort, we felt that declaring a separate method for each and every single combination of char types (simple, capital, number, special) in content was a very counter-intuitive and inefficient restriction that also appears unrealistic. We just implemented a sort algorithm that works for any combination and made all interfaces call that single algorithm. We did not understand what industrial practices would lead to this restriction.

To summarize, although we do not understand the full rationale behind these choices for the project and hence our observations may well be wrong, in our journey of implementing and testing our project we felt that these were strange restrictions that caused much difficulties for lessons we did not understand in end. Thus, for future versions of CS4218 we believe that if these choices are removed or at least better explained it would be a positive improvement for the module.