APRIL 16, 2014

CS4218 SOFTWARE TESTING AND DEBUGGING SPRING 2014

QUALITY ASSURANCE REPORT

TEAM ROCKET
DAVID HERYANTO
DARREN-GAVIN HO WEILIANG
FRANCIS PANG DE XIAN
JANICE CHOW WEN XIAN

Topics: Comparison of testing strategies, measures of confidence, Orthogonal Defect Classification (ODC).

Analysis across project artifacts and milestones

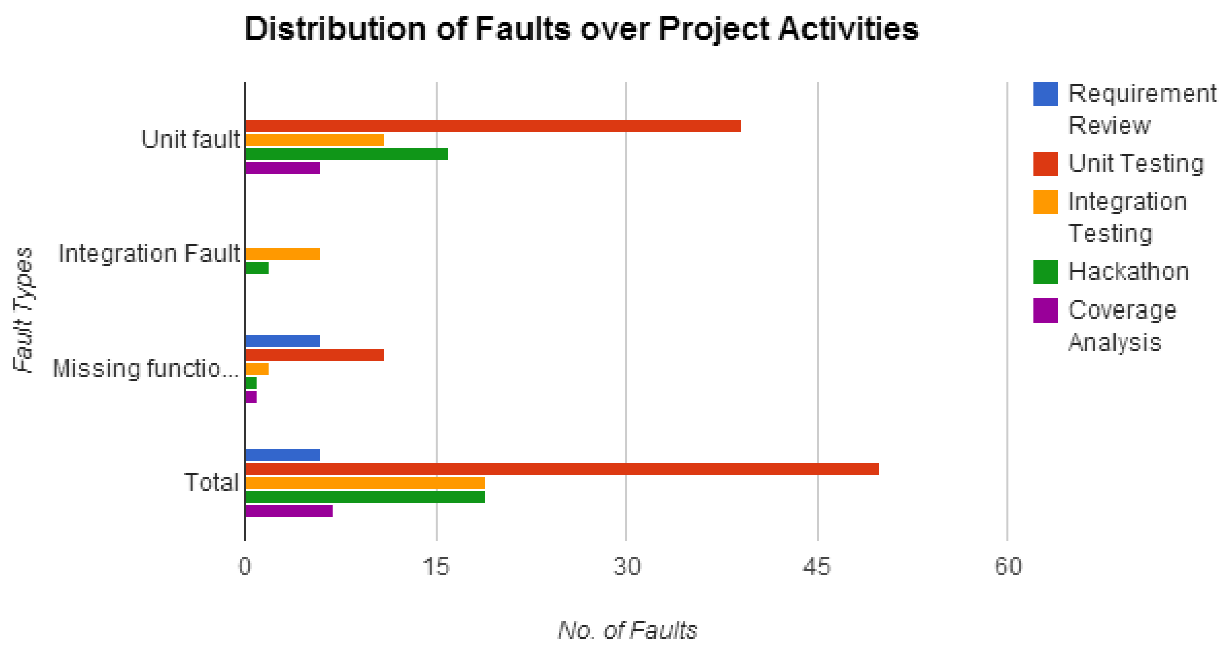
1. Quantity of written Source and Test Code.

Name	Total Lines of Test Code	Total Lines of Source Code
Darren	3583	1197
David	2484	1362
Francis	1500	980
Janice	3764	1518
Total	11331	5057

2. Analysis of the distribution of fault types versus project activities:

2.1. Distribution of faults over project activities.

Fault Type / Project Activities	Requirement Review	Unit Testing	Integration Testing	Hackathon	Coverage Analysis
Unit Fault	0	39	11	16	6
Integration Fault	0	0	6	2	0
Missing Functionality	6	11	2	1	1
Total	**6**	**50**	**19**	**19**	**7**



Discuss what activities discovered the most faults.

Most of the bugs that we found are during *unit testing* which was conducted right after we have implemented each tool. The bugs were mostly *unit faults and missing functionalities*. This is mostly because when we develop each functionality, we first focus more on getting it to work correctly and only scrutinize on the border cases (null and invalid values) afterwards. On the other hand, there were no integration faults found during unit testing since we are only testing the tools individually.

From requirements review, we can only discover missing functionalities through the discussions on the tools' requirement among ourselves and from clarifying the requirements with the TAs.

From heuristics, with more bugs found the more difficult it will be to find new bugs. Therefore, the total number of bugs found during integration testing and hackathon are significantly lesser than during unit testing. Moreover, they are mostly unit faults rather than integration faults.

For coverage analysis, the potential bugs found are from the branches that were not covered previously.

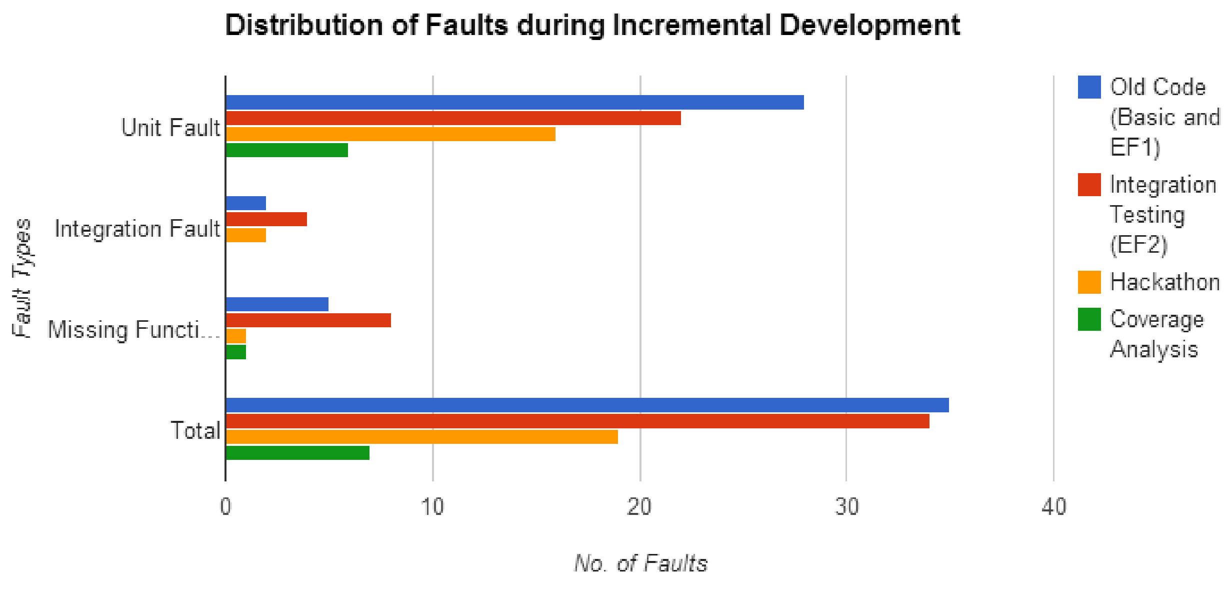
Discuss whether the distribution of fault types matches your expectations.

Overall, the distribution of fault type matches our expectations such as most faults are unit testing faults which are found during unit testing and most integration faults are found during integration testing.

However, the number of integration faults is lesser than the number of unit faults found during integration testing. This could be because the unit testing was not done thoroughly enough prior to integration testing. Furthermore, integration faults usually results from interface errors and shared variables between modules like in the case of our project, the system property "user.dir" is shared among the Shell and the CdTool. However, since the interfaces were already created at the start of the project, it is very unlikely to encounter any interface error.

2.2. Distribution of faults found in basic functionality (old code) during activities on adding extended functionality (new code):

Fault Type / Project Activities	Old Code (Basic & EF1)	Integration Testing (EF2)	Hackathon	Coverage Analysis
Unit Fault	28	22	16	6
Integration Fault	2	4	2	0
Missing Functionality	5	8	1	1
Total	**35**	**34**	**19**	**7**

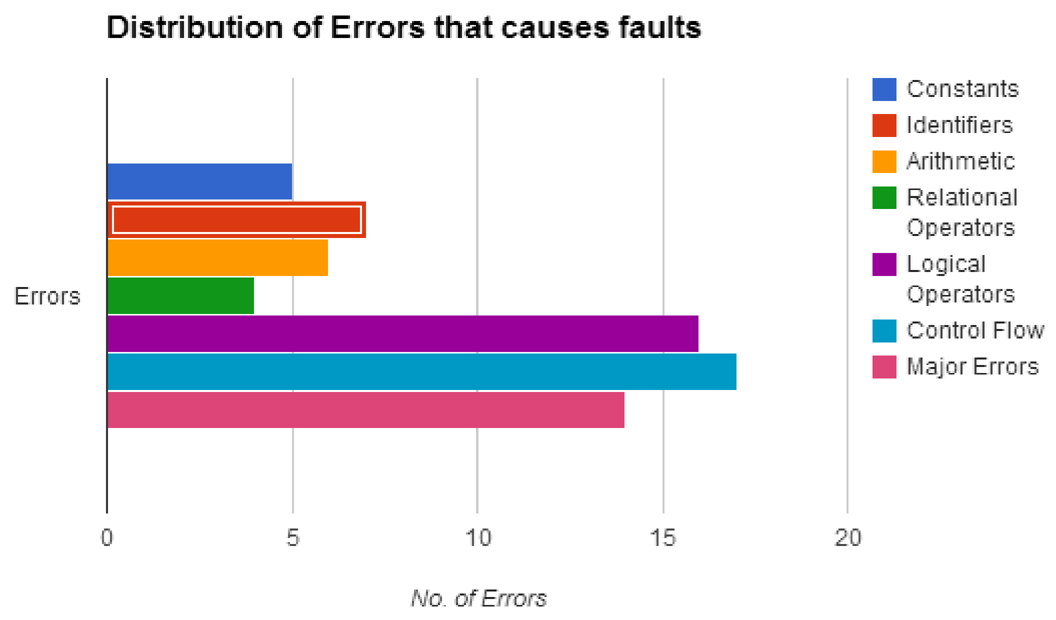


Discuss whether the distribution of fault types matches your expectations.

In general, the distribution of fault types matches our expectations. This is because as seen in the unit faults, the number of errors gradually decreases as we go through the various project phases. In addition, for integration faults and missing functionality, it would be obvious that integration testing would be able to surface the most faults as during integration testing, it is common to see tools that may or may not work when executed in integration; thus making obvious in our case.

2.3. Distribution of causes for the faults discovered:

Error Class	Errors
Constants	5
Identifiers	7
Arithmetic	6
Relational Operators	4
Logical Operators	16
Control Flow	17
Major Errors	14



From the table above, we can see that most of the faults were caused by errors pertaining to *localized errors in control flow*. This is evident and obvious as such errors occur due to algorithmic operations that are found within each implementation of the tools. For example, tools such as *cut* would have methods that requires the coverage of various kinds of scenarios pertaining to the execution of the command; whereas for tools such as *cat,* the number of errors surfacing could be lesser. In addition, this is also similar for the second most number of errors produced by *logical* operators as they are a subset of control flow statements such as *if* and *while.*

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

Activity	% of total project time
Requirement Analysis	7 %
Coding	47 %
Test Development	28 %
Test Execution	18 %

4. TDD vs. Specification-Based Testing.

What are advantages and disadvantages of both based on your project experience?

Advantages of TDD:

- Having the test cases *before* implementation helps the developer to be certain on the requirements by simply referring to the test cases.
- Time taken to implement the functionalities will be shorter
- Deriving test cases before implementation helps to ensure that all features have been implemented.
- Since we have the test cases prior to development, for each functionality we developed, we can run the relevant test cases to see if we have implemented it correctly before moving on to implement the next functionality

Disadvantages of TDD:

- Does not ensure high code *coverage*.
- More time spent on requirements analysis and coming up with the test cases.
- If requirements are *not well understood*, it would be difficult to derive test cases and there could be changes made to the test cases during development.
- *Boundary* values for testing are not obvious if requirements are not well understood. For example, constraints on a command's option arguments may not be well understood prior to implementing the tool.
- Negative test cases may be missed out when deriving test cases for TDD before implementation.

Advantages of Specification-Based Testing:

- When deriving test cases after implementation, the programmer will have a clearer understanding of his code and will be able to come up with more constructive test cases of higher code coverage.

Disadvantages of Specification-Based Testing:

- Unable to ensure that all errors are handled properly by each tool. This is because test cases that are developed will be based on the functionalities that have already been implemented and will not detect functionalities that were missing.

Can they be combined in a beneficial manner?
Yes it can. Start off with the TDD process by developing test cases based on requirements and features to implement. After implementation, we can add in more test cases to ensure a higher code coverage.

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

In particular, what 10% of classes achieved the most branch coverage?
The 10% most covered classes are CatTool and EchoTool where both achieved 100% branch coverage

How do they compare to the 10% least covered classes?
The 10% least covered classes are CommTool (83%) and CdTool (84%). The reason why these two classes are the least 10% covered classes is because they are generally much more complicated to implement as compared to the classes that achieved the most branch coverage. In addition, cd causes the shell *environment* to change (the working directory changes) and this may causes faults if cd is not properly implemented.

Provide your opinion on whether the most covered classes are of the highest quality. If not, why?
Using the bug report from hackathon, the number of valid bugs for CatTool, EchoTool and CdTool are 0 while CommTool have 2 valid bugs. From this, we could imply that the most covered classes are of the highest code quality in the sense that there are fewer bugs. This is assuming that the more the number of errors for a tool the lower the code quality for a tool. However, this may not be the case because each error may have different *importance* and negative effects. For instance, an error on the method to read file is quite important because it is used by a lot of other tools and an error on that method will propagate upwards. Thus, 1 error in this aspect may be worse than 3 errors in other aspects (such as error in outputting error message).

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

Unit testing triggered us to change the design of some of our code. This is especially so if we were *not sure of the requirements* while implementing the tools and subsequently realized that we have missed out some cases or have misinterpreted a large part of the requirement.

For example, during the TDD phase, we designed our code *merely to pass* the test cases and hence may *falsely belive* we have fulfilled all the requirements. Subsequently, as we reviewed the requirements by cross-checking with the actual unix shell, we found that the test cases in TDD were missing out some functionalities. Hence, we had to redesign the code.

Did integration testing help you to discover design problems?

From our project experience, integration testing did not help in discovering design problems. Since the *command design pattern* was used in the shell's design architecture, it allows the tools to be implemented independent of each other. Furthermore, the interfaces for each tool was already clearly defined. Hence, if each individual tool is implemented properly, design problems should not arise from integration testing. Moreover, our team ensured that we have a common consensus on the arguments passed from Shell to each tool and the results returned from each tool's execution before diving into the implementation. This helped in preventing design problems from integration testing.

7. 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?

The most useful tool for test automation which we used for our project is JUnit as it allows us to run all or selected test cases in the test suite and focus on debugging the failing test cases by using the Eclipse debugger.

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

A coding practice that we would change is to test each method individually before proceeding on to implementing the next method. This is especially so when we developed private method. We would test and ensure the private method is running correctly before implementing the public method which calls it. Hence, we are more inclined to the TDD approach. Since the test cases will be constructed before implementation, it is more convenient for the programmer to test his method that he has just implemented.

For testing practices, we would write more negative test cases for unit testing.

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

PMD is useful but only to give an overview of our code quality and to pinpoint obvious unconventional coding practices in the code such as unnecessarylyLongVariableNamesInTheSourceCode. However, PMD may not know our coding *convention* such as our test method being named in the format

"[Test Method]_[Testing Scenario]_[Expected Result]" and might give a false warning. Hence, PMD can be further enhanced by being customisable in only detecting a violation of specific rules so that it is easier for the programmer using PMD to make the necessary amendments to his code.

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

If we use it continuously as we write the code, it distracts us from coding well, because sometimes we just want to make the program work first without considering too much about the coding style. However, the excessive warning and notifications by PMD may prevent us from doing so effectively.

However as a final step that is performed, after we finish writing the code and test cases, PMD's warnings help us to figure out where and why we should refactor portions in our code.

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

Without test cases, we can check the quality of our code by documenting the inputs we used to run the program as a whole and manually validate the output of the program. It is more efficient and effective than random testing since we can reuse the inputs recorded in the test documentation for regression testing. The test documentation gives us a level of assurance that some cases have been tested for.

We can also exchange code with other people or teams so as to do black box testing by providing inputs to the program. They can help us figure out functionalities we missed out and check for assumptions in our code.

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

Test cases amount to 100% statement coverage will give us the most confidence. However, we note that it may be impossible to cover 100% of the statements because some code may be unreachable by testing such as the main method of the Shell and catching of exceptions thrown by the Java library functions. Therefore, as long as all testable statements have been covered by the test cases, we can then achieve the most confidence in our code quality.

11. Describe the one most important thing on testing that you have learned/discovered.

We learned that high code coverage is important in achieving high code quality assurance.

12. What answers/results in the questions above are counterintuitive to you?

Usually, it is said that integration testing helps to spot for bugs at the interface between two modules that are integrated together. However, from our project, we realized that there were hardly any interface errors spotted from integration testing. This could be because the structure of our program (i.e. the classes and interfaces) have already been well thought through and created for us before the commencement of the project. Therefore, the bugs that are spotted during integration testing are either bugs that can be detected through unit testing or a misinterpretation of arguments between the Shell and tools. For example, the CutTool did not allow -d option to have arguments without quotes but the Shell removed the quotes from the argument before passing it to CutTool.

Specification-based testing is intuitive to programmers since we usually write the test cases after we have implemented all the functionalities. On the other hand, TDD is counterintuitive as we had to write the test cases prior to implementation. Furthermore, by doing so, there are many positive experiences we had when developing our project using TDD. However, as no development method is completely perfect, we also spotted some flaws with TDD as compared to the specification-based testing.