CS4218 SOFTWARE TESTING AND DEBUGGING

Spring 2014

QA Report

Team Undefined

Camillus Gerard Cai
Eu Beng Hee
Huynh Van Quang
Wang Boyang

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

Analysis across project artifacts and milestones

Project artifacts:

1 Unit test cases.

2 TDD added test cases.

3 Integration test cases + bug reports + 100% method coverage.

4 Hackathon test cases + bug reports + incremental coverage.

5 Unit test coverage vs. integration test coverage. Incremental coverage.

Please answer the following questions. Please provide numbers, plots and explanations 2-3 lines of text for each question.

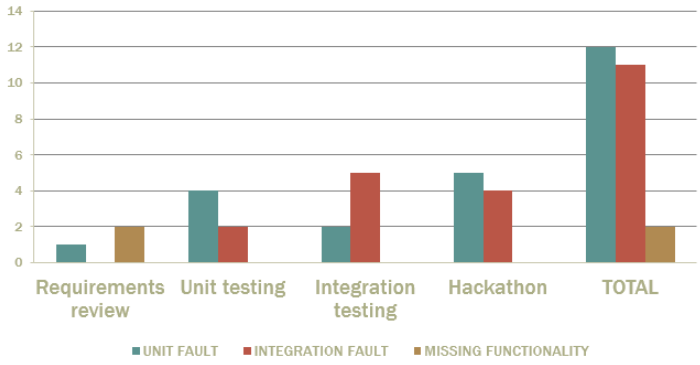
Report

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

Text code: 5641 vs. Source code: 3686 LOC

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.

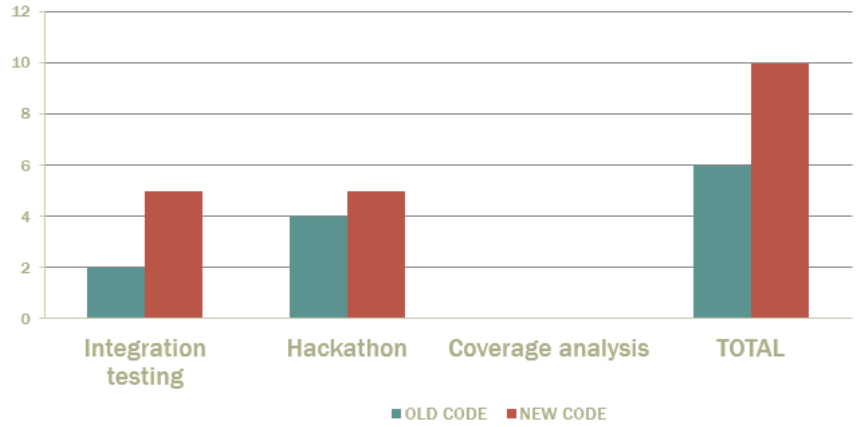


Type of faults	Requirements review	Unit testing	Integration testing	Hackathon	TOTAL
Unit fault	1	4	2	5	12
Integration fault	0	2	5	4	11
Missing functionality	2	0	0	0	2

The most number of bugs are found during Hackathon. This matched our expectation, as the other team has the motivation to find as many bugs as possible. Presumably, this is why software companies typically have separate development and testing team.

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

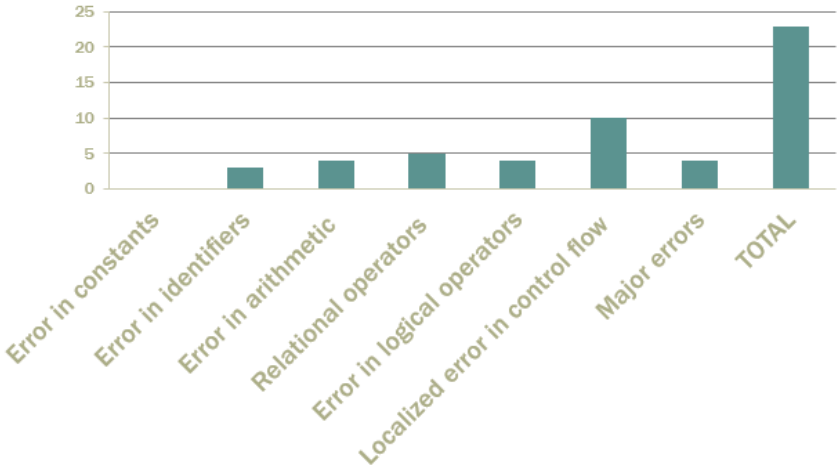
Activity: integration testing, hackathon, coverage analysis. Discuss whether the distribution of fault types matches your expectations.



Type of faults	Integration testing	Hackathon	Coverage analysis	TOTAL
Old code	2	4	0	6
New code	5	5	0	10

It matches of expectation that most of the bugs appear in the new code. From time to time we also discover regressions in the old code.

2.3. Analyze and present a distribution of causes for the faults discovered: *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').



# of faults	Error in constants	Error in identifiers	Error in arithmetic	Relational operators	Error in logical operators	Localized error in control flow	Major errors	TOTAL
	0	3	4	5	4	10	4	23

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

- requirements analysis 10%
- coding 20%
- test development 50%
- test execution 20%

4. TDD vs. Specification-Based Testing.
What are advantages and disadvantages of both based on your project experience? Can they be combined in a beneficial manner?

TDD is significantly more time-consuming. In a small, throwaway school project, TDD is a waste of time. However, it scales well. When the project grows to become sufficiently big, the process of TDD provides a strong test suite along the way, and thus integrate the development and testing well.

Specification-based testing is less time-consuming. But it often misses out corner cases that are not inside the scope of the specs.

It might be possible to do TDD only for a critical feature, thus combine the two techniques.

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

Not necessarily.

Coverage metrics do not measure the quality of the test cases provided. Poor test cases may not detect subtle logic bugs.

Ridiculous KPIs tagged to code coverage metrics presented a perverse incentive to create redundant test cases, increasing the cost of software development, and reducing the signal-to-noise ratio of unit testing.

In particular, what 10% of classes achieved the most branch coverage? How do they compare to the 10% least covered classes?

The 10% of classes that achieved the most branch coverage are typically the ones that have *the least number of branches*. They are easy to cover.

Provide your opinion on whether the most covered classes are of the highest quality. If not, why?

No.

Covering more branches doesn't necessarily lead to exposure of faults for the reasons described above. Program logic that marshals data between the input/output formats and the format required for the algorithm contain the most branches and are usually the easiest to test. Once again, poorly designed KPIs create a perverse incentive to test code that rarely impacts software quality over the actual algorithm.

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

The initial designs we used were not intrinsically problematic.

However, the requirement to do unit testing selected for designs that were more easily testable with JUnit. We do not consider these designs superior or inferior to the original designs. Had a different testing paradigm been used, our program architecture would have had facilities to better integrate with it.

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?

Would you change any coding or testing practices based on the bugs/debugging you

encountered in the CS4218 project?

A delta-debugging tool would have been useful if coupled with a revision control system that supports lightweight branching like Git.

No, the project had no impact on our engineering practices. The problems we faced were largely due to the poorly designed assignment and assessment criteria. The software interfaces given could not support the tools they were supposed to abstract in an idiomatic way. The specifications were too ambiguous; it would have been better if they were called “design guidelines” instead.

8. 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?

It was helpful in specific instances where it caught possible logic errors. In the context of this project, it was significantly more distracting than useful.

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

By manually test the functionalities and by actually using the program.

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

The test suite that *we* developed to ensure that our software met the requirements we set gave the most confidence.

The tests that had to be included to bring method and branch coverage beyond a certain threshold were practically useless.

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

Specification-based testing relies heavily on requirements and specifications. Before clearing out doubts on the requirements, we could barely start the testing.

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

The fact that in this project, implementing and testing required functionality took up only 30-40% of the time, whereas administration and meeting KPI metrics took up the vast majority of our time.