Weifan Jiang (NetID: wfjiang)
Yuqi Huang (NetID: yh73)

Fail Fast

Motivation:

The goal of our project is to prioritize tests based on fail probability within a test suite.

Numerous empirical studies already showed that prioritization can improve a test suite's rate of fault detection. For instance, Kim et al. [1] presented a "history-based" prioritization technique, in which information gained from previous testing cycles is used to better inform the selection of a subset of an existing test suite for use on a modified version of a system. Elbaum et al. [2, 3] reported on experiments exploring the effects of program structure, test suite composition, and changes on prioritization. However, almost all prioritization techniques focus on C language, while other test frameworks do not gain too much attention. For instance, the widely used unit testing framework for Java, JUnit, does not implement test prioritization, which means that test cases within one test suite will always run under the same order, without optimizing based on correlations between test cases and past results. As we all know, a common situation when debugging is to run the test suite multiple times within short time, with making small change to the code between each run of test suite. We would like to be notified as quickly as possible whether any test cases already fail instead of waiting for the execution of all other test cases that are both irrelevant to the bug but placed earlier in the test suite.

This is not the very ideal situation. We want to run the test cases that are more likely to fail first to make user be notified that something is not working as quickly as possible. In order to achieve this goal, we would like to implement an extension (or some equivalent which provides the same side feature) that reorders the test to firstly run tests that are more likely to fail in JUnit.

Approach:

We believe that solving this question is valuable and the approach we mentioned above is scientific. Not only test case prioritization saves time for users (especially when debugging, as debugging happens frequently before homework is due), but also, reordering test cases is possible to implement: the hard part is to find the optimal way to calculate/model the failing probability.

When thinking about one test case's failing probability, we will focus on three aspects:

- The failing rate of this test case in past runs of the test suite. If a test failed a lot in the past, it is worth checking early that whether this test case fails again in this run.
- Consecutive failings. Maybe some test case's results in the last five runs of test suite are all fail, then this test suite would be more likely to fail again in this run of test suite. In the other hand, if developer has made effective change in the source code which get rid of the bug before running test suite this time, it is still valuable to check whether this test case passes and notify the developer as early as possible.
- Correlation between test cases. We can discover correlations between test cases based. Say that test A and test B always have identical results. When we run test A first, we can update test B's failing probability to the conditional probability of B fails based on result of A.

Thus, we can measure the numerical value of these three aspects, and combine them with some weight (need experimentation) to calculate the final prioritization of each test case. Note that this prioritization

value is up to change during runtime of test suite (because if test A passes, the conditional probability of test case B based on A will change.)

Since this prioritization value is changing, we can use a priority queue to hold each test case's priority during runtime, and after a test case finishes, update relevant test cases' priority score by updating their conditional probability component.

The difference between our approach and previous approaches is that we not only record the "history" of these tests, but also measure the correlation between test cases. The information from previous runs can be used to weight test cases according to these correlations.

The limitation of this approach includes that: keeping track of each test case's failing probability is expensive once the amount of test case gets too much; the dependency graph for test cases is hard to build; it is expensive to recalculate the priority score of relevant test cases.

Challenges & Risks:

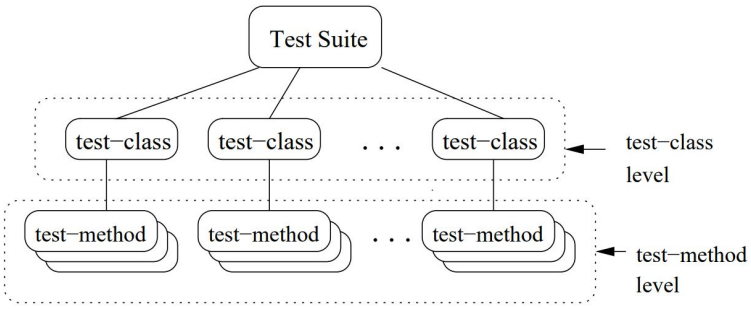

Figure 1: JUnit test suite structure

The most serious challenge we currently see in developing the product is measuring the influence of order in method-level, since JUnit tests are performed in class-level. One test class may contain numerous test methods, as we can see in Figure 1, and methods within the same class might also be correlated in some ways. It is quite difficult to order those methods based on the result of class-level experiments. One way that might be able to minimize this risk is to give each test method a unique ID, and then modify the test class in order to accept a list of IDs and run tests based on that order.

References

[1] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In Proceedings of the International Conference on Software Engineering, pages 119–129, May 2002

[2] S. Elbaum, D. Gable, and G. Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In Proceedings of the International Software Metrics Symposium, pages 169–179, April 2001.

[3] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. Journal of Software Testing, Verification and Reliability, 12(2), 2003.