32

Relentless Improvement

5. I will fearlessly and relentlessly improve my creations at every opportunity. I will never degrade them.

Robert Baden Powell, the father of the Boy Scouts of America, left a posthumous message exhorting the scouts to leave the world a better place than they found it. It was from this statement that the "Boy Scout Rule" was derived.

Check the code in cleaner than you checked it out.

How? By performing random acts of kindness upon the code every time you check it in. One of those random acts of kindness is to increase test coverage.

Test Coverage

Do you measure how much of your code is covered by your tests? Do you know what percentage of lines are covered? Do you know what percentage of branches are covered? If not, why not?

There are plenty of tools that can measure coverage for you. For most of us, those tools come as part of our IDE and are trivial to run. So there's usually no excuse for not knowing what your coverage numbers are.

What should you do with those numbers? First, let me tell you what not to do. Don't turn them into management metrics. Don't fail the build if your test coverage is too low. Test coverage is a complicated concept that should not be used so naively.

Such naive usage sets up perverse incentives to cheat. And it is very easy to cheat test coverage. Remember that coverage tools only measure the

amount of code that was executed, not the code that was actually tested. This means that you can drive the coverage number very high by pulling the assertions out of your failing tests. And, of course, that makes the metric useless.

The best policy is to use the coverage numbers as a developer tool to help you improve the code. You should work to meaningfully drive the coverage toward 100% by writing actual tests.

One hundred percent test coverage is always the goal, but it is also an asymptotic goal. Most systems never reach 100%, but that should not deter you from constantly trying to attain it.

That's what you use the coverage numbers for. You use them as a measurement to help you improve, not as a bludgeon with which to punish the team and fail the build.

Mutation Testing

One hundred percent test coverage implies that any semantic change to the code should cause a test to fail. TDD, TCR, and to a lesser extent, Small Bundles, are good disciplines to approximate that goal because, if you follow those disciplines ruthlessly, every line of code, every condition, and every branch is tested.

Such ruthlessness, however, is often impractical. Programmers are human, and disciplines are always subject to pragmatics. So the reality is that even the most assiduous test-driven developer will leave gaps in the test coverage.

Mutations testing is a way to find those gaps; and there are mutation testing tools that can help.

A mutation tester runs your test suite and measures the coverage. Then, it goes into a loop, mutating your code in some semantic way and then running the test suite with coverage again. The semantic changes are things like changing `>` to `<` or `==` to `!=` or `x=` <*something*> to `x=null`. Each such semantic change is called a *mutation*.

The tool expects each mutation to fail the tests. Mutations that do not fail the tests are called *surviving mutations*. Clearly, the goal is to ensure that there are no surviving mutations.

Running a mutation test can be a big investment of time. Even relatively small systems can require hours of runtime. So these kinds of tests are best run over weekends, or at month's end.

I have frequently been impressed by the kinds of subtle problems these tools have uncovered. So it is definitely worth the occasional effort.

Semantic Stability

The goal of test coverage and mutation testing is to create a test suite that ensures semantic stability. The semantics of a system are the required behaviors of that system. A test suite that ensures semantic stability is one that fails whenever a required behavior is broken. We use such test suites to eliminate the fear of refactoring and cleaning the code. Without a semantically stable test suite, the fear of change is often too great.

Following one of the testing disciplines gives us a good start on a semantically stable test suite; but none of those disciplines is sufficient for full semantic stability. Coverage, mutation testing, and acceptance testing should be used to improve the semantic stability toward completeness.

Cleaning

Perhaps the most effective of the random acts of kindness that will improve the code is simple cleaning—refactoring with the goal of improvement.

What kinds of improvements can be made? There is, of course, the obvious elimination of code smells. But I will often clean code even when it isn't smelly.

I make tiny little improvements in the names, in the structure, in the organization. These changes might not be noticed by anyone else. Some folks might even think they make the code less clean. But my goal is not simply the state of the code. By doing minor little cleanings, I learn about the code. I become more familiar and more comfortable with it. Perhaps my

cleaning did not actually improve the code in any objective sense; but it improved my understanding of and facility with that code. The cleaning improved me as a developer of that code.

The cleaning also did something else that should not be understated. By cleaning the code, even in minor ways, I am flexing that code. And one of the best ways to ensure that code stays flexible is to regularly flex it. Every little bit of cleaning I do is actually a test of the code's flexibility. If I find a small cleanup to be a bit difficult, I have detected an area of inflexibility that I can now correct.

Remember, software is supposed to be soft. How do you know that it is soft? By testing that softness on a regular basis. By doing little cleanups and little improvements and feeling how easy or difficult those changes are to make.

Creations

The word used in the promise above is *creations*. In this chapter I have focused mostly on code; but code is not the only thing that programmers create. We create designs, and documents, and schedules, and plans. All these artifacts are creations that should be continuously improved.

We are human beings. *Human beings make things better with time*. We constantly improve everything we work on.