Jay Fields on Working with Unit Tests

Stefan Tilkov

Over the past decade, unit testing has become a core aspect of development. Instead of being neglected, testing is now aligned with the idiom “the more the better.” But have development teams turned testing into dogma, and in so doing lost sight of keeping a balance between costs and benefits? Are developers now creating more technical debt by writing unmaintainable tests? How much coverage is enough? Can you have too many tests? In this episode, host Stefan Tilkov explores this and other issues surrounding unit testing with software engineer Jay Fields, author of Working Effectively with Unit Tests.

You can hear the entire interview online at www.se-radio.net. Portions of the interview not included here for space include what to test; testing boundaries between systems, stubs, and mocks; validation; and test patterns. —Robert Blumen

Stefan Tilkov (ST): The original article on this topic was “Test Infected: Programmers Love Writing Tests.” Do programmers love writing tests? If they don’t, what’s a good way to get them to love tests?

Jay Fields (JF): I think the first time you start writing tests, you fall in love with code where you can be playful and then tell yourself, “There’s also value to this code; it’s giving me confidence in my application.”

For those who don’t love it, I think they fall into two categories. The first is people who believe they don’t have enough time to write tests. I think that’s a terrible justification. If you don’t have time to write tests, it’s possibly because you’re always debugging. Maybe you wouldn’t have to debug so often if you were writing tests.

The second category consists of people who have worked with tests, but the tests haven’t provided them as much value as they expected. For people in that category, I would encourage them to do more of what they were already doing, but see if there’s another way to write tests that would give them more value in the future.

ST: What things do people do wrong when they write unit tests?

JF: A lot of people conflate test-driven development [TDD] and unit testing as if they’re the same thing, but they’re not. A test that you use to TDD a new feature for your application is not guaranteed to be a valuable test for you in the long term. Sometimes the best thing to do is to delete the test. Maybe you wrote the test because at the time you weren’t really sure how you wanted things to work, and now you have a working feature, so you want to keep it. But maybe one user wants to use this feature every three
months, and if things go wrong, nobody cares. You get a phone call and maybe you update a database table yourself or something trivial like that. Do you really want to strap yourself with additional code that you have to maintain over time? It’s more likely that that feature will change—to either become more valuable or go away—than it is that a bug will creep into the feature itself and that your test will save you.

ROI is something I like to use when I talk about tests. You have to look at each test and ask yourself, “What’s the return on investment for this test?” If a test is only, at best, going to save me from getting a phone call once every six months, then I don’t want to maintain it in my code base. If the test helped you TDD, fantastic—but does it still need to be here? If the answer is no, then you need to delete it.

Let’s say you already wrote a test that helped you TDD it—it doesn’t mean it’s the best test to maintain that code going forward. Look at the code again. Ask somebody else, “If this test were failing right now, what clues would you want in order to figure out the source of the problem as fast as possible?” Then you can change the test from a test that helped you deliver to one that helps you maintain value over time.

ST: Assuming you want to create something that’s valuable in the long run, what things do you have to think about? What makes a test good and maintainable under that circumstance?

JF: One thing that bothers maintainers is how much test code you have to wade through before you can figure out what’s actually wrong. Whenever I see your test failing, what kind of clues does the test give me? Does it tell me where I can look in the domain to figure out what’s going on?

It’s more than likely when a test is failing that it surprises you because it seems unrelated to what you’re doing. You’re annoyed because you want to move on. It’s breaking and you don’t know why.

If you open it and you can pretty quickly figure out what’s going on by just looking at the broken method, you can probably just move on. But if you open it and find some variable in there, say a field-level variable in a Java class, and it’s declared at the top and initialized somewhere (but you don’t really know where), then you have to find where it’s used. Maybe you find a setup method, or maybe you find the helper method. You start navigating around but you’re moving farther and farther away, chasing something that seems unrelated to what you wanted to do.

Programmers practice DRY [don’t repeat yourself], but I think repeating yourself in ways that will enable someone who’s never seen the test before to more quickly understand what’s going on can be a good thing. It’s great to apply DRY to the test suite level. For example, running all your database updates in a transaction and then rolling everything back is obviously a good idea rather than trying to do that in every single test individually.

Within a single test, obviously, you don’t want to repeat too many things. But when it comes to a grouping of tests for maintainers, maybe the best thing is not knowing that there’s a grouping of tests. When a group of tests all fail together, then you know to look at the infrastruc-
ture code. The grouping is valuable because it helps you make sense of the failures. But when two out of five tests in a group fail, you first need to understand all the logic built into the grouping. Every second you spend trying to understand the infrastructure that applies to the other three that are still passing is a waste.

You should avoid any type of looping, constructs, or reflection if possible. Basically, what you want are straightforward tests that you can easily navigate. Only navigate into domain code when necessary, if possible.

ST: You’re saying that there are different rules for the test code and for the rest of the system. Should test code be as maintainable as the production code, or more or less?

JF: It really makes no sense—even from a high level—to approach these two pieces of code in the same way. The test should be approached from value: what is the value of the test with respect to keeping the application up and running? In production, pretty much all code should be created equally. A bug in some trivial feature could just as easily take down the system if it throws an exception.

You want to apply different thinking to your tests. For tests, readability is more important than performance. But, at the same time, performance is still going to be important enough on a different scale. I work in the trading industry, where milliseconds are important to us. Individual milliseconds for tests are not a big deal, but when the test suite starts to run at around 10 minutes [which is a long time for us], you have to wonder if people are going to keep using that test suite. There are a lot of tradeoffs, but I think you start with readability because you want people to be able to maintain your tests and then start making tradeoffs where necessary.

ST: A debate recently took place between Kent Beck, Martin Fowler, and David Heinemeier Hansson about whether unit tests are a waste of time. Did you follow it, and do you have an opinion on it?

JF: I did follow it and agree that the kind of tests David talks about are a waste of time. I hope they did delete them and that they’re dead. But that doesn’t mean everybody is spending time that they shouldn’t on testing. I think there are plenty of unit tests out there that are very helpful.

You’re going to get what you put into it. Like I said before, it’s very easy to write terrible tests, and it’s very easy to then blame the tests. I advise every single person out there who is writing tests that are not making them more productive to just stop. If they’re not making you more productive, you shouldn’t do it. You can replace the time spent writing tests with any other [productive] activity. I applaud [Hansson] for coming out saying that this sacred cow is not necessarily what it’s made out to be. I think more people should delete their tests if they’re not finding them helpful.

But then they have a choice to make: Do you want to just go without tests? Do you want to go without that confidence? You didn’t start out with a test suite that was all bad. At some point, it was providing you value. Now, do you want to invest in trying to write better unit tests? I think testing is worth investing in, unless you’ve found some other way to get the same level of confidence [without it]. But I don’t know what that would be.

ST: I originally planned to ask you whether there’s such a thing as having too many tests, but I think you just answered it.

JF: Too many tests is the same as not enough tests. In both cases it’s suboptimal. Whether you waste time debugging because you don’t have enough tests or you waste time maintaining tests that don’t need to be there, at the end of the day both of those things amount to waste.

ST: What do you think about a metric such as code coverage? Is that a useful thing?

JF: I do think that code coverage is a useful thing. I remember when Relevance, now Cognitect, was putting 100 percent code coverage in their contracts. At the time, I thought it was a great idea. But they don’t do that anymore, and most people I know aren’t looking for 100 percent
code coverage because you have to do silly things to get it.

ST: Like testing getters and setters?

JF: Exactly. Also testing framework methods. I don’t have any desire to test Joda-Time ever again. I want to be able to assume that Joda-Time works out of the box. I want to use it without having to fight some silly test coverage battle.

ST: You mentioned in passing that your 10-second test suites hit the file system and hit the database. Some would say that this doesn’t qualify as a unit test if it does something like that. What’s your take on that?

JF: In the first version of my book, I said unit tests were not allowed to cross boundaries: no messaging, no file system, no database. And unit tests were only allowed concrete classes as the class under test. Martin Fowler told me, “This will just not work. You’re not going to convince the industry that this is how to unit-test. There’s too much momentum from people who believe that unit testing is allowed to hit the database.” He convinced me that it was a bad plan to try to redefine it. I’ve come to terms with the fact that unit testing is such a general term that you just have to roll with it.

You can have tests that don’t hit the database or file system. These are the tests that you know are going to run fast, and that’s why you avoid those things. You have a bunch of tests that run really quickly and mock things out. That’s great because they give you some confidence. Then you have other tests that you know are going to run a little bit slower. Those tests are going to hit those things because that’s what you need them to do. At some point, you need to hit the database. But at the end of the day, you want at least one test that hits the database to ensure that integration is correct.

ST: Why did you write a book on the topic?

JF: There are some really good beginner books and books that give you a lot of detail. But there’s a big gap in between that I think the community will benefit from: Here’s how I like to write my unit tests. Here’s my guidance in a big-picture way.

What I set out to do in the book was to take the experience I’ve picked up over the last dozen years and put it together in a way that shows more than the trivial blog-posting example. It takes a domain that people are familiar with. It takes some tests that start out looking awful and evolves them into a maintainable style.

STEFAN TILKOV is cofounder and principal consultant at Innoq. Contact him at stefan.tilkov@innoq.com.