Test-Driven Dogma July 8, 2017

Test-driven development (TDD) has been around for a long time; however, enthusiasm for it has waned considerably over the last 5 years or so. This is a good thing. In particular, I've been encouraged by the number of former TDD advocates who realized the error of their ways and are willing to provide an honest evaluation of its costs, benefits, and limitations.

First, let's define exactly what we're talking about with respect to the concept of TDD. I'm talking specifically about the idea of test-first development, as opposed to writing tests after a feature/module has been written. I'll have a little bit to say about the test-after approach, but for now, I want to focus on "true" TDD, which is test-first.

My first problem with TDD is that it is totally unnatural with respect to how most developers actually work and solve software problems. Of course, I suppose that begs the question: developers could change and TDD offers a major paradigm shift in how developers would work. After more than 20 years in software and working with a diverse group of developers (probably close to two hundred), I have yet to meet any developer who thought TDD was an intuitive or natural approach to software development in an actual project. For me personally, I found TDD not only unnatural, but claustrophobic. No matter how well the requirements for a feature are defined, it almost always looks different after you get into it and realize that there are some subtleties that the business analyst didn't or couldn't see. Developing any kind of test prior to this "vetting" of the requirements is a waste of time.

Some of the goals of TDD are admirable, although certainly not exclusive to it. A couple of the goals include (1) code will naturally be more testable because it is explicitly developed around tests, and (2) refactoring is an integral part of the process. As much as I dislike TDD, I'll concede that it does encourage writing testable code and refactoring, both important. However, TDD is not required to achieve these goals. In fact, these goals could be just as easily (and more efficiently) achieved by writing tests after the fact. Indeed, I would argue that you're more likely to achieve greater and more relevant test coverage with a test-after approach. It is certainly much easier to get there once a feature has been developed, rather than "guessing" beforehand.

With respect to refactoring, I usually find it makes sense to sit with code for a few days at least and see how it operates, works with other code, and integrates into the whole. Usually, the relevant refactoring strategy reveals itself; however, it isn't always apparent right away. That's why I generally don't like to do much refactoring right away. I get a lot more bang for my buck if I wait.

My disdain for TDD should not be taken for a dismissal of unit testing and writing testable code in general. I generally support this, although when someone asks for my advice about it, I usually say that unit tests should be prioritized to mission-critical/high-risk parts of the app. Several years ago, I worked on a project with 5000+ unit tests (of which I wrote a couple hundred). The coverage was incredible, the only problems were (1) it took a few hours to run all the tests (and this made it challenging to integrate them into the build process), (2) developing and maintaining the tests required a very large percentage of each developers' time (in this case, 66 – 75%). In short, developing a large suite of unit tests becomes its own monster – a hungry monster that breaks when you don't expect it to, has difficult bugs to track down, has a steep learning curve, etc.. So, you have to evaluate unit testing where it makes sense, especially with regard to any budgetary constraints.

Finally, you may be wondering why I refer to TDD as “dogma”. In my view, that’s exactly what it is. We all know about various “religious” battles that exist in software development. TDD is one of those battles, defended to a large extent by hipsters and others looking for something new and trendy. Or maybe TDD’s defenders are just trying to prove that they are smarter or more sophisticated than everyone else. In any case, despite some bloody battles, time wasted, budgets busted, and lessons learned, it appears that the war is over. TDD is finished.