5. Test Driven Development

It has been over ten years since Test Driven Development (TDD) made its debut in the industry. It came in as part of the Extreme Programming (XP) wave, but has since been adopted by Scrum, and virtually all of the other Agile methods. Even non-Agile teams practice TDD.

When, in 1998, I first heard of "Test First Programming" I was skeptical. Who wouldn't be? Write your unit tests *first*? Who would do a goofy thing like that?

But I'd been a professional programmer for thirty years by then, and I'd seen things come and go in the industry. I knew better than to dismiss anything out of hand, especially when someone like Kent Beck says it.

So in 1999 I travelled to Medford, Oregon, to meet with Kent and learn the discipline from him. The whole experience was a shocker!

Kent and I sat down in his office and started to code some simple little problem in Java. I wanted to just write the silly thing. But Kent resisted and took me, step by step, through the process. First he wrote a small part of a unit test, barely enough to qualify as code. Then he wrote just enough code to make that test compile. Then he wrote a little more test, then more code.

The cycle time was completely outside my experience. I was used to writing code for the better part of an hour before trying to compile or run it. But Kent was literally executing his code every thirty seconds or so. I was flabbergasted!

What's more, I recognized the cycle time! It was the kind of cycle time I'd used years before as a kid[1] programming games in interpreted languages like Basic or Logo. In those languages there is no build time, so you just add a line of code and then execute. You go around the cycle very quickly. And because of that, you can be *very* productive in those languages.

But in *real* programming that kind of cycle time was absurd. In *real* programming you had to spend lots of time writing code, and then lots more time getting it to compile. And then even more time debugging it. *I was a C++ programmer, dammit!* And in C++ we had build and link times that took minutes, sometimes hours. Thirty-second cycle times were unimaginable.

Yet there was Kent, cooking away at this Java program in thirty-second cycles and without any hint that he'd be slowing down any time soon. So it dawned on me, while I sat there in Kent's office, that using this simple discipline I could code in real languages with the cycle time of Logo! I was hooked!

The Jury Is In

Since those days I've learned that TDD is much more than a simple trick to shorten my cycle time. The discipline has a whole repertoire of benefits that I'll describe in the following paragraphs.

But first I need to say this:

- The jury is in!
- The controversy is over.
- `GOTO` is harmful.
- And TDD works.

Yes, there have been lots of controversial blogs and articles written about TDD over the years and there still are. In the early days they were serious attempts at critique and understanding. Nowadays, however, they

are just rants. The bottom line is that TDD works, and everybody needs to get over it.

I know this sounds strident and unilateral, but given the record I don't think surgeons should have to defend hand-washing, and I don't think programmers should have to defend TDD.

How can you consider yourself to be a professional if you do not *know* that all your code works? How can you know all your code works if you don't test it every time you make a change? How can you test it every time you make a change if you don't have automated unit tests with very high coverage? How can you get automated unit tests with very high coverage without practicing TDD?

That last sentence requires some elaboration. Just what is TDD?

The Three Laws of TDD

1. You are not allowed to write any production code until you have first written a failing unit test.
2. You are not allowed to write more of a unit test than is sufficient to fail—and not compiling is failing.
3. You are not allowed to write more production code that is sufficient to pass the currently failing unit test.

These three laws lock you into a cycle that is, perhaps, thirty seconds long. You begin by writing a small portion of a unit test. But within a few seconds you must mention the name of some class or function you have not written yet, thereby causing the unit test to fail to compile. So you must write production code that makes the test compile. But you can't write any more than that, so you start writing more unit test code.

Round and round the cycle you go. Adding a bit to the test code. Adding a bit to the production code. The two code streams grow simultaneously into complementary components. The tests fit the production code like an antibody fits an antigen.

The Litany of Benefits

Certainty

If you adopt TDD as a professional discipline, then you will write dozens of tests every day, hundreds of tests every week, and thousands of tests every year. And you will keep all those tests on hand and run them any time you make any changes to the code.

I am the primary author and maintainer of FITNESSE,[2] a Java-based acceptance testing tool. As of this writing FITNESSE is 64,000 lines of code, of which 28,000 are contained in just over 2,200 individual unit tests. These tests cover at least 90% of the production code[3] and take about 90 seconds to run.

Whenever I make a change to any part of FITNESSE, I simply run the unit tests. If they pass, I am nearly certain that the change I made didn't break anything. How certain is "nearly certain"? Certain enough to ship!

The QA process for FITNESSE is the command: `ant release`. That command builds FITNESSE from scratch and then runs all the unit and acceptance tests. If those tests all pass, I ship it.

Defect Injection Rate

Now, FITNESSE is not a mission-critical application. If there's a bug, nobody dies, and nobody loses millions of dollars. So I can afford to ship based on nothing but passing tests. On the other hand, FITNESSE has thousands of users, and despite the addition of 20,000 new lines of code last year, my bug list only has 17 bugs on it (many of which are cosmetic in nature). So I know my defect injection rate is very low.

This is not an isolated effect. There have been several reports[4] and studies[5] that describe significant defect reduction. From IBM, to Microsoft, from Sabre to Symantec, company after company and team after team have experienced defect reductions of 2X, 5X, and even 10X. These are numbers that no professional should ignore.

Courage

Why don't you fix bad code when you see it? Your first reaction upon seeing a messy function is "This is a mess, it needs to be cleaned." Your second reaction is "I'm not touching it!" Why? Because you know that if you touch it you risk breaking it; and if you break it, it becomes yours.

But what if you could be *sure* that your cleaning did not break anything? What if you had the kind of certainty that I just mentioned? What if you

could click a button and *know* within 90 seconds that your changes had broken nothing, *and had only done good?*

This is one of the most powerful benefits of TDD. When you have a suite of tests that you trust, then you lose all fear of making changes. When you see bad code, you simply clean it on the spot. The code becomes clay that you can safely sculpt into simple and pleasing structures.

When programmers lose the fear of cleaning, they clean! And clean code is easier to understand, easier to change, and easier to extend. Defects become even less likely because the code gets simpler. And the code base steadily *improves* instead of the normal rotting that our industry has become used to.

What professional programmer would allow the rotting to continue?

Documentation

Have you ever used a third-party framework? Often the third party will send you a nicely formatted manual written by tech writers. The typical manual employs 27 eight-by-ten color glossy photographs with circles and arrows and a paragraph on the back of each one explaining how to configure, deploy, manipulate, and otherwise use that framework. At the back, in the appendix, there's often an ugly little section that contains all the code examples.

Where's the first place you go in that manual? If you are a programmer, you go to the code examples. You go to the code because you know the code will tell you the truth. The 27 eight-by-ten color glossy photographs with circles and arrows and a paragraph on the back might be pretty, but if you want to know how to use code you need to read code.

Each of the unit tests you write when you follow the three laws is an example, written in code, describing how the system should be used. If you follow the three laws, then there will be a unit test that describes how to create every object in the system, every way that those objects can be created. There will be a unit test that describes how to call every function in the system every way that those functions can meaningfully be called. For anything you need to know how to do, there will be a unit test that describes it in detail.

The unit tests are documents. They describe the lowest-level design of the system. They are unambiguous, accurate, written in a language that the audience understands, and are so formal that they execute. They are the best kind of low-level documentation that can exist. What professional would not provide such documentation?

Design

When you follow the three laws and write your tests first, you are faced with a dilemma. Often you know exactly what code you want to write, but the three laws tell you to write a unit test that fails because that code doesn't exist! This means you have to test the code that you are about to write.

The problem with testing code is that you have to isolate that code. It is often difficult to test a function if that function calls other functions. To write that test you've got to figure out some way to decouple the function from all the others. In other words, the need to test first forces you to think about *good design*.

If you don't write your tests first, there is no force preventing you from coupling the functions together into an untestable mass. If you write your tests later, you may be able to test the inputs and the outputs of the total mass, but it will probably be quite difficult to test the individual functions.

Therefore, following the three laws, and writing your tests first, creates a force that drives you to a better decoupled design. What professional would not employ tools that drove them toward better designs?

"But I can write my tests later," you say. No, you can't. Not really. Oh, you can write *some* tests later. You can even approach high coverage later if you are careful to measure it. But the tests you write after the fact are *defense*. The tests you write first are *offense*. After-the-fact tests are written by someone who is already vested in the code and already knows how the problem was solved. There's just no way those tests can be anywhere near as incisive as tests written first.

The Professional Option

The upshot of all this is that TDD is the professional option. It is a discipline that enhances certainty, courage, defect reduction,

documentation, and design. With all that going for it, it could be considered *unprofessional* not to use it.

What TDD Is Not

For all its good points, TDD is not a religion or a magic formula. Following the three laws does not guarantee any of these benefits. You can still write bad code even if you write your tests first. Indeed, you can write bad tests.

By the same token, there are times when following the three laws is simply impractical or inappropriate. These situations are rare, but they exist. No professional developer should ever follow a discipline when that discipline does more harm than good.

Bibliography

[Maximilien]: E. Michael Maximilien, Laurie Williams, "Assessing Test-Driven Development at IBM," http://collaboration.csc.ncsu.edu/laurie/Papers/MAXIMILIEN_WILLIAMS.PDF

[George2003]: B. George, and L. Williams, "An Initial Investigation of Test-Driven Development in Industry," http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf

[Janzen2005]: D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," *IEEE Computer*, Volume 38, Issue 9, pp. 43–50.

[Nagappan2008]: Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," Springer Science + Business Media, LLC 2008: http://research.microsoft.com/en-us/projects/esm/nagappan_tdd.pdf