The Pragmatic Bookshelf

PragPub

The Second Iteration

IN THIS ISSUE

* ANNIVERSARY ISSUE
* Alistair Cockburn on Agile at 15
* Ron Jeffries on XP
* Michael Nygard on unicorns
* Allen Holub on #NoEstimates
* Jonathan Rasmusson on testing
* Jeff Langr on TDD benefits
* Venkat Subramaniam on impediments to TDD
* and Natasha Murashev on Swift
* Plus Johanna Rothman on unavoidable estimates, Marcus Blankenship on management no-nos, Antonio Cangiano on books, plus tech news and a puzzle

Issue #85
July 2016

Contents

FEATURES

DEPARTMENTS

Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail webmaster@swaine.com. The editor is Michael Swaine (michael@pragprog.com). Visit us at http://pragprog.com for the lowdown on our books, screencasts, training, forums, and more.

ISSN: 1948-3562

On Tap

The Seven-Year Hitch

by Michael Swaine

Step up to the bar and see what we have on tap.

I've been editing *PragPub* for seven years. I think we're effectively married. This month, we're celebrating seven years of *PragPub* with a big fat issue. In addition to regular columns by Johanna Rothman, Marcus Blankenship, and Antonio Cangiano, we're serving up a double helping of eight (8!) feature articles, and we've invited some old friends to help us celebrate with articles that look back on where we came from and forward to where we might go ...

Alistair Cockburn writes about the Agile Manifesto and how it came about. Another Manifesto author, Ron Jeffries, reflects on the Extreme Programming philosophy. Allen Holub weighs in on #NoEstimates and Johanna Rothman advises on dealing with estimates when no is not an answer.

Michael Nygard sorts the unicorns from the dinosaurs and concludes that the software of the future needs to be fleet of foot and highly disposable. Suppose software were trivial to create and only ever needed to be used once. Completely disposable. What would that mean to the way we write it? And what would it take to make that a reality?

And then there's testing. Test-Driven Development has transformed how we (at least for some values of "we") program. Three experts in testing and development weigh in on TDD this issue: Jonathan Rasmusson, Jeff Langr, and Venkat Subramaniam. They explore the testing pyramid, the virtues and benefits of TDD, and some of the barriers to implementing TDD.

Natasha Murashev has been mastering the Swift language and shares a practice she's worked out for bending the language to her needs. It's totally wrong, it's not how you're supposed to do it — but it works. Plus there's Antonio with all the latest tech books and your editor with his review of the past month in tech. And a puzzle. And a poem. Because it's our anniversary!

Who Did What

writing, editing, production: Michael Swaine mike@swaine.com • editing, markup, coding: Nancy Groth nancy@swaine.com • extreme curation: BoB Crew • customer support, subscriptions: webmaster@swaine.com • submissions: Michael Swaine mike@swaine.com • staff curmudgeon: John Shade (john.shade@swaine.com)

Photo Credits: Cover: "Laguna Beach artists" [U1] by Pierre Omidyar is licensed under Creative Commons [U2], page 13: "Dev Team Charter" [U3] by gdsteamis is licensed under Creative Commons [U4], page 17: "Wakeboarding Hamburg 2013" [U5] by www.GlinLowe.com is licensed under Creative Commons [U6], page 23: "Unicorn" [U7] by eliztesch is licensed under Creative Commons [U8], page 27: "Beer Vision" [U9] by Johnny Silvercloud is licensed under Creative Commons [U10], page 33: "Pyramid" [U11] by Mario Guo is licensed under Creative Commons [U12], page 37: "Karen, an example of a computer programmer" [U13] by Nelson Pavlosky is licensed under Creative Commons [U14], page 43: "Ukrainian Cafe" [U15] by Mark is licensed under Creative Commons [U16], page 50: "Runner (IMG_2615)" [U17] by Richard Ash is licensed under Creative Commons [U18],

You can download this issue at any time and as often as you like in any or all of our three formats: pdf [U19], mobi [U20], or epub [U21].

Half of a Third of a Century in TDD

Reflections and Recommendations

by Jeff Langr

One of the leading practitioners of test-driven development reflects on his career.

I built software the old-fashioned way — the tried-and-turbid, non-TDD way — for the first sixteen and 2/3 years of my development career — until I discovered TDD in 1999. Sixteen and 2/3 years since (the second half of my career), I've ingrained and practiced TDD as my preferred programming approach. I'd like to share how I've benefited from TDD, a practice well beyond the span of being a fad, in the second half of my development career.

Debating TDD

It's easy to wage attacks against something you don't personally care for.

- You can point to lack of research studies. For TDD, a handful of studies exist. Most show similar results: Quality improvement of around 15% and some comparable increase in initial development cost (the studies don't consider long-term benefits of costs). But really, most people aren't going to be persuaded by reseach to embrace a practice. Ardent academics might. The consistent research shown for TDD *should* at least convince people that TDD can generate tangible benefits.
- You can use the "all or nothing" argument. "TDD can't drive the development of complex algorithms," for example. Experienced TDD practitioners (TDDers or test-drivers) will admit that TDD isn't a technique that magically imbues you with the key insights needed to derive a particularly interesting algorithm. Thinking is still a key part of what is really a simple practice — a habit that you ingrain partly to afford regular reflection on what's the best thing to do next. And TDD *will* help you derive 99.9% of the algorithms most of us code day-to-day. It will also help immensely by giving you the confidence to move code bits around in order to derive a better algorithm.
- You can suggest those promoting it are out to make money, or that they don't build real software, or you can find other personal mud to sling. In my case, yes I make money off of teaching TDD, but I also make sure I build "real" software as well in real development teams.

Oh, that's enough already.

There are probably dozens more excuses for not wanting to practice TDD. Initially I thought I would address the more common ones using logic and anecdotes (mine and others). But I've gotten to the point, now as I write this and now in my career, where I simply don't want to waste my time. We're all adults. We all work in development teams, and there are gobs of kinds of teams to choose from — the number of people practicing TDD is still a considerable minority of all developers. We all have the right to choose, and the right to promote our preferred practices. If you don't like TDD, find a team that doesn't

want to do it. If you're intrigued by some of the benefits we've seen, explore it and find a team that embraces it.

Ultimately, I practice TDD because I enjoy it: I find it continually gratifying, and I appreciate how it's helped me build better quality code. It's that simple. I've also greatly appreciated seeing the light bulb turn on in other peoples' eyes once they get it (and get how to do it well). I see little point in arguing beyond that — you either accept that others have found success with TDD, or you don't. And if you don't, I'm perfectly happy with your choice to build software differently. Just don't ask me to maintain your code — I've seen it.

Instead of debating more "points against," I'll talk about my observations regarding doing TDD for this past sixth of a century.

Almost Everyone Writes Tests

In 1999, the prevailing attitude was, "I'm a programmer, not a tester. Why should I be writing tests?" Thankfully, developer testing at least is no longer a shocking concept to most. Questions like "how do you test your code?" are commonplace in interviews. A perusal of sigificant projects in GitHub suggests that a large number of them contain automated unit tests, integration tests, or acceptance tests.

Granted, most of these tests are written after the fact, not by doing TDD. But the general awareness of the value of developer tests is important, and hopefully leads more programmers to a test-before technique.

Today, I wouldn't consider hiring a senior developer who thought they were above programmer tests. I don't care as much whether or not they practice TDD, but the attitude that programmers can integrate untested code is thankfully antequated.

TDD Is Just How I Build Code: Simply

I've ingrained TDD as a habit. I am "test infected," to use an old but accurate term. It's hard for me to not first think about how to express a small bit of behavioral need in the form of a test. Occasionally, I'll write code without test-driving it, but it's usually an unhappy experience; I'll talk about that in a soon-upcoming section.

I view TDD as a simple cycle that accommodates thinking incrementally and continually:

- red: I think about a behavior and decision I need to capture and document. I code a test that demonstrates an example of that behavior/decision. I think hard about what it means if the test passes — does our system already provide that behavior somehow? Or did I just do something stupid?
- green: I think about how to code a solution that passes my tests. I add no more complexity than needed to support all the current behaviors that the tests demonstrate.
- refactor: I think hard about the new, small increments of test and production code I created. I bring all my design background into the foreground of my thought, and fix any deficiencies in the design or code constructs.

The ingrained cycle TDD cycle of red-green-refactor allows me to focus my thinking on creating a correct small increment. It also gives me constant opportunities to keep the design simple enough to support any forthcoming increments, foreseen or not.

Anything Can Be, and Is Better, Built in Small Increments

I used to slam out large chunks of code, sometimes hours worth, before building and testing it out (a slow process). My feedback cycles were long! It sometimes took me 15 minutes to find out that a tiny code change was horribly wrong from the get-go. Using TDD, I find out constantly and rapidly the moment I introduce defective code. My tests tell me whether or not I coded logic defects; these never get integrated into the team's common source repository.

Most of us practicing TDD are also practicing iterative-incremental development in the form of agile. The short, high-feedback cycles of agile mesh well with the short, high-feedback cycles of TDD. Both support the mentality of producing just enough code, given a list of the most important things to do. Both promote the notion that you should be able to stop development at any time, and still have high-quality, potentially shippable software.

Writing Tests After Is Harder, And Not Much Fun

Believe me, I've tried to build code without the use of TDD. See "My Ruby Regrets" [U1] for a candid story about writing tests after-the-fact for something we deemed a small, two-day effort. It seems fast at first to slap out code, but my point of regret is usually only a few hours into crafting a solution. Regarding the above linked story, we quickly built a code base that was overly complex and rigid — we couldn't change it easily without worrying that we'd broken something, and it took us a painful couple minutes to find out if we had. Not even a full day into a two-day project, we dearly wished we'd test-driven the dumb thing.

Pairing Makes It Even Better

Doing TDD on my own is a reasonable exercise, but it tends to leave me doubting my choices. This feeling is no different than how I feel when I just whack away at code. However, "just whacking away at code" doesn't accommodate pairing as readily as does TDD. In fact, the ping-pong pairing model is expressly built around TDD: I write a test, then my pair gets it to pass (and refactors). We then switch: My pair writes the next test, which I get to pass. And so on. This lively programming model bifurcates thinking a little: at any given point, one of us is thinking more about the definition of the problem, and the other is thinking more about its solution. And I no longer second-guess myself.

Refactoring Is Fun, And I Do It Without Fear

Well before practicing TDD, I learned to enjoy cleaning up my code, always looking for ways to make it more expressive and better designed. This refactoring was a slow, high-risk activity, however. I had to go through the

tedious manual test cycle: build, execute, exercise the code, look for problems (and sometimes miss them). The more I tanked myself with defective refactoring sessions, however, the more I moved toward an "it ain't broke, don't fix it" mentality. Net result: My code got worse.

With TDD, I worry little and refactor at will every few minutes. Every piece of behavior I drive into the system is well-tested, by definition.

Design Is Important, and I Consider It Constantly

Design is so important, in fact, that I consider it malpractice to create an expensive, comprehensive, speculative design model at the outset of a project initiative. The expense — or is it the arrogance — leads us to think that we must ossify the design and subsequently find creative ways to force-fit features into it.

The reality is simple. Needs change, knowledge increases, and our speculations are almost always wrong to some degree — particularly as the level of detail increases. This doesn't mean we eliminate up-front design completely — we simply do it with a lot less detail, and with a lot less certainty. Design should work its way into every fiber of a well-executed agile process: during planning discussions (sprint, daily, story), and during every piece of the TDD cycle — particularly the refactoring step.

Success with TDD requires continually applying everything you can know about crafting a quality design.

My Systems Are Smaller

I've seen development teams shrink a couple reasonably-sized (> 100,000 LOC) systems to less than half their size, by virtue of adding unit tests and continually factoring away duplication and unnecessary code. These systems ended up being a pleasure to work with for their clarity and lack of clutter.

Yes, with TDD, you must maintain test code in addition to production code! It *is* additional work. You'll probably write about as much test code as you write production code. But, if my contention that continual refactoring can halve the size of your system, then in the long run, the total amount of code you write (tests + production code) will be about the same... and you'll have all the benefits that the tests can provide.

I Eliminated a Complete Class of Defects

Yeah, yeah, of course I've shipped defects. It's always embarrassing.

Here is one classification scheme for defects:

- Logic
- Integration / configuration
- Non-deterministic behavior
- Misunderstood / missed requirements

At a recent startup, I shipped enough integration/configuration related defects to make me sick. We didn't build automated acceptance tests (long story), and instead favored production monitoring to catch problems... which meant

that production integration defects were almost unavoidable. (Worse, many of the problems required massive corrective effort, due to the need to rework voluminous amounts of errant data.)

Misunderstood or missed requirements will always occur. "I thought you meant X," I tell the product owner, and I'm told, "No, I meant Y." If you think of each test in TDD as a captured decision about some small bit of behavior, though, it might help remind you to go and ask the product owner whether or not that decision is correct... before it's too late. That plus some level of spec by example can reduce defects here.

Logic defects, however, just don't happen very often with TDD. I can't remember the last time I shipped code that had a dumb coding mistake.

I Know What the Code Does

Reiterated: Each test in TDD is a captured decision about some small bit of behavior.

"What does the system do in the case of X," asks my product owner. "I don't know, let me go look," I answer. In a system built without TDD, answering this question can take a while — hours, in some cases. For me, however, I usually have the answer in a few minutes by reading through the tests. I don't waste nearly as much time.

I Deliver Faster

When I first start test-driving a system, it takes a little while to put the initial constructs in place. Continual quality takes, well, a bit of consideration continually added to each increment of code. But with the continual refactoring that TDD accommodates, things simplify. Keeping tests clean and abstract makes it easier and quicker to write similar tests. Meaningful tests and test names make it easier to find and understand code, as does continue refactoring of the production code itself. Duplication eliminated reduces maintenance costs and risk. The smaller methods and simple design that results also decreases the chance of defects. The control-space autocomplete in my IDE usually helps me out a lot more in a well-designed system, and in test-driving, I can use quick-fix constructs from my tests to minimize the amount of code I actually have to type (which also reduces the potential for defects).

Over time, the attention to cleanliness in a system minimizes the amount of degradation that's otherwise typical in a system.

Haters Gonna Hate

I was told by one twit (i.e., someone who uses Twitter, of course) that "you're doing it wrong, mate" — because I practiced TDD. Not knowing me or my background, he proceeded to claim that this was a black-and-white situation, that I just wrote about development while he actually did it, and therefore I must be wrong. It was amusing, but the narrow-mindedness of it all was pretty sad.

Then there are the folks who create frameworks that make unit testing difficult and conclude that this means TDD is a bad idea. Don't believe them.

But don't believe me either — find out for yourself (but at least make sure you follow the wisdom of those who've found success with TDD).

I'm Still Having More Fun

I've met hundreds upon hundreds of developers over the years. Many, too many, expressed constant frustration with their codebase: everything is hard to change, hard to understand, easy to break, and it's just not enjoyable working in it.

It doesn't need to be that way.

33 Years And Going Strong

If I am fortunate, I have another 16.666.. years remaining in my software development career (which, all added together, amusingly takes me to 49.999... years, just shy of a 50th anniversary — God knows I don't need to have a real celebration of this stuff).

TDD isn't a silver bullet, we all say. To that I add: Show me something better, and I'll be more than happy to adopt it and leave TDD by the wayside. It'd be nice to have a third technique to compare against the other two sixth-century chunks. Right now, the only thing better I see on the horizon is self-programming systems, and once they arrive, I know it will be time to retire.

About the Author

Jeff is a veteran software developer with a third of a century of experience. He's written five books on software development: Pragmatic Unit Testing in Java 8 with JUnit [U2] with Andy Hunt and Dave Thomas, Modern C++ Programming with Test-Driven Development [U3], Agile In a Flash [U4] with Tim Ottinger, *Agile Java*, and *Essential Java Style*. He also contributed two chapters to Uncle Bob's (Robert C. Martin's) book *Clean Code*.

External resources referenced in this article:

[U1] http://rubylearning.com/blog/2010/12/08/my-ruby-regrets/

[U2] https://pragprog.com/book/utj2/pragmatic-unit-testing-in-java-8-with-junit

[U3] http://pragprog.com/book/lotdd/modern-c-programming-with-test-driven-development

[U4] http://pragprog.com/book/olag/agile-in-a-flash