The Pragmatic Bookshelf

PragPub

Keeping It Light

IN THIS ISSUE:

* Launching our tenth year!
* Russ Olsen asks Fermi questions
* Marcus Blankenship micromanages programmers
* Jeff Langr reflects on the meaning of pragmatism
* Novi Milenkovic challenges best practices and tames complexity in UX design
* Michael Swaine reboots the Harvard Mark I computer
* and John Shade hates driving

Plus all the new tech books, a puzzle, and the launch of our new science fiction department!

Issue #109
July 2018

Contents

FEATURES

The Pragmatic Path 12
by Jeff Langr
Jeff reflects on what pragmatic programming means as this magazine enters its tenth year and The Pragmatic Programmer enters its twentieth.

When Best Practices Can Hurt You 19
by Novi Milenkovic
Best practices, followed blindly, can give you false confidence. And the average user does not exist.

Tom Watson, Howard Aiken, and Who? 24
by Michael Swaine
Last month we attended a soirée at the home of Charles Babbage. This month we attend a festive event in the basement of Harvard's Cruft Laboratory at 6 Oxford Street in Cambridge, Massachusetts.

COLUMNS

Swaine's World 3
by Michael Swaine
In which Mike loses his job and offers you career advice. Originally written Tuesday, May 12, 2009.

Technically Awake 6
by Russ Olsen
How many M&Ms does it take to equal the weight of the Sun?

New Manager's Playbook 9
by Marcus Blankenship
If it was good enough for Henry Ford...

Shady Illuminations 45
by John Shade
We need to lock up Jerry Seinfeld and James Corden.

BOOKS

Antonio on Books 29
by Antonio Cangiano

The Rust Programming Language. Plus 19 other new tech books.

From The Pragmatic Bookshelf 32
An excerpt from Programming Phoenix 1.4 plus the top sellers and author events from The Pragmatic Bookshelf.

Beyond the Fields We Know 41
by Michael Swaine

RIP Harlan Ellison.

DEPARTMENTS

On Tap 1

The PragPub Puzzle 11

Solution to Puzzle 43

The BoB Page 44

Published monthly in PDF, mobi, and epub formats by The Prose Garden, LLC, Cave Junction, OR. E-Mail webmaster@swaine.com. The editor is Michael Swaine (michael@pragprog.com).

ISSN: 1948-3562

On Tap

What's Up

What's here and who did it.

With this issue, *PragPub* is entering its tenth year. Friend of the magazine Jeff Langr takes the occasion to reflect on the past nine years of *PragPub*, the 19 years since the publishing of *The Pragmatic Programmer*, and the nature and meaning of pragmatism.

Your editor is also in a reflective mood, and offers up a moment in the history of the computer, focusing on three individuals — Tom Watson, Howard Aiken, and Norman Bel Geddes. Each had an important part to play in the creation of an important early computer. Grace Hopper and Charles Babbage make cameo appearances, too.

Novi Milenkovic makes his first appearance in our pages, but not his last, with a thoughtful essay on agile practices, UX design, and the management of complexity. Best practices, he decides, can sometimes lead you astray.

Russ Olsen writes about Fermi questions this month. These are questions like, how many taxi cabs are there in New York City? Or how many M&Ms does it take to equal the weight of the Sun? Or how many pennies would you have to stack to equal the height of Mount Everest? Crazy questions for which you would think you lack the data to even make a plausible guess. But it turns out, you can often do better than that. Estimation on the basis of excruciatingly limited data is not an uncommon situation in software development, and sometimes this seems so impossible that we just guess. But you can, Russ explains, do better than that.

Marcus Blankenship, in his regular column on managing programmers, takes a surprising stance this month. I'll leave it at that. Antonio Cangiano surveys all the recent tech books and is particularly taken by one on the Rust language. We have an excerpt from the upcoming Pragmatic Bookshelf book, *Programming Phoenix 1.4*. John Shade returns to the topic of self-driving cars. And this month's puzzle is a sudoku plus an anagram plus some really lame puns on the names of Turing award winners.

Oh, and we're launching a new feature focusing on science fiction.

We hope you enjoy it!

Who Did What

writing, editing: Michael Swaine mike@swaine.com

editing, markup: Nancy Groth nancy@swaine.com

customer support, subscriptions: mike@swaine.com

submissions: Michael Swaine mike@swaine.com

extreme curation: BoB Crew

reclusive curmudgeon: John Shade john.shade@swaine.com

data protection officer: Michael Swaine mike@swaine.com

Photo Credits

Cover and page 1: “Fireworks Art” [U1] by nickgesell is licensed by Creative Commons 2.0.

Pages 11 and 43: “Puzzle” [U2] by voyeg3r is licensed by Creative Commons 2.0.

Page 12: “Birthday Cake with Candles” [U3] by Keith Hinkle is licensed by Creative Commons 2.0.

Page 19: “TC40 TechCrunch 40: More Powerset Test Tubes with Vodka” [U4] by Brian Solis is licensed by Creative Commons 2.0.

Page 24: “Harvard University Campus Harvard Mark I Computer” [U5] by Ted Eytan is licensed by Creative Commons 2.0.

Pages 29 and 32 and 41: “Books - Colored” [U6] by frankes is licensed by Creative Commons 2.0.

We take seriously our responsibility to safeguard your personal data. Our privacy policy is here [U7].

You can download this issue at any time and as often as you like in any or all of our three formats: pdf [U8], mobi [U9], or epub [U10].

The Pragmatic Path

Heading into Ten and Twenty Years of Pragmatism

by Jeff Langr

Jeff reflects on what pragmatic programming means as this magazine enters its tenth year and *The Pragmatic Programmer* enters its twentieth.

With this issue, we head into the tenth year of *PragPub*, a celebration of pragmatic software development. We also head into the twentieth year after publication of *The Pragmatic Programmer*, the groundbreaking 1999 book by Andy Hunt and Dave Thomas that remains a favorite of many developers, even after two decades. We are in the era of pragmatism!

But what does it mean to be pragmatic? The philosophical tradition of pragmatism promotes acting on ideas in order to vet them with actual human experiences. [1] This modern pragmatic movement dates back to around 1870. However, as *The Pragmatic Programmer* tells us, the original notion of pragmatic derives from Greek word *pragmatikos*, meaning "fit for business." Or in rawer form, simply "do."

The word was in use in French during the 15th century, where it was used for public decrees of power — *pragmatic sanctions* — which were backed by the force of law. For a time, "pragmatic" had a very negative connotation — "meddlesome" or even "fussy" — the negativity likely emphasized by folks who received the business end of the sanctions.

Today, our simple interpretation of the meaning of *pragmatic* comes without negative connotation. It can be associated with notions of "practice" and "practicality." It is probably the best antonym for "idealistic," although, Andy Hunt tells me that "idealist" isn't the best antonym for pragmatic, "dogmatic" is.

The Pragmatic Programmer helped refuel the flame of pragmatism for the next generation and more. Hunt and Thomas summarized the key qualities of a pragmatic programmer as someone who:

- Is an early adopter
- Adapts rapidly
- Embraces inquisitiveness and critical thinking
- Is a realist
- Is a jack-of-all-trades. [2]

Of these qualities, "realist" most directly aligns with the classic notion of pragmatic. "Adapts rapidly" and "realist" also work as characteristics a dogmatist would not have. And all but one necessitates hands-on exploration and practice.

The landmark book is chock-full of simple techniques and approaches that aid in building quality software. Hunt and Thomas touch upon just about everything a 1999 developer should know about how to succeed in a pragmatic fashion: testing, assertions, design, debugging, using an editor, Unix commands, writing an email, distributed objects, source control, software process and team dynamics, big O notation, writing specs, problem solving, and of course … all that and more! Hunt and Thomas also introduced us to two extremely useful concept inventions of theirs — tracer bullets and DRY ("don't repeat yourself") — that remain powerful today.

Who's Not Pragmatic?

But wait … whoever said programmers *shouldn't* be pragmatic? Were we not pragmatic in 1999? What does it even mean to *not* be pragmatic?

If you built software before the 1990s, you grew up in an era where computing resources were scarce. Every CPU second mattered. You were taught to work up your designs and even your code on paper first, before you dared offer them to the computer. Feedback cycles were long; as a result, the lesson you learned from an error was to invest an even larger amount of up-front thinking time. The natural result was an increased use of, and more formalization of, speculative design method, which in turn set the seed for more purist, or idealist, approaches to programming.

If you built software in the 1990s, you learned about a number of ideas for programming, many of them predicated on the fact that software was hard to change. Proponents of things like "late design" suggested that the best thing we could do is to invest the vast majority of our effort in getting the design right, and only then constructing the code. (I recall being told that 90% of the development effort should be design.) Maybe this approach was the right track: Where we were headed was a magical world where we could draw pictures (models) that represented the system, and the code would be generated for us.

This concept of self-coding systems didn't seem far-fetched to some. In 1995, I worked with a fellow named Jan who honestly believed we would need no programmers in ten or so years. Never mind that drawing complex diagrams around system interactions is still programming of a sort — perhaps "visual programming" — and never mind that visual interaction models aren't the most succinct or even clearest way of representing what systems need to do. (Think "mathematics," as one simple example, a peskily pervasive programming need.)

Still, the premise had a strong hold: We should be able to represent what a system must do by first modeling it … then sitting back and letting a programmer or computer (which, by the way, was another word for "programmer" in the 1950s) do the dirty work. Henceforth, came the rise of UML and model-heavy design methodologies, where a team of well-paid architects and designs could draw pictures of the system, and lower salary folks could type in code for those pictures without having to think too hard. Also came ideas like use cases, where we could model the requirements in narrative form, and another team of programmers could build to those narratives.

Scaling Everest

In 1995, I worked on a large project at the defunct telecommunications company MCI. The goal of the multi-year effort named "Everest" was to re-engineer MCI's revenue stream systems, this time capitalizing on the promise of object-oriented development. I joined about a year into the project and left a year later, one year shy of its demise. Late in my one-year tenure, we hadn't delivered a line of code (and the project never would). We had delivered a first collection of use cases — maybe a few dozen written pages' worth — painstakingly authored as a result of bickering in a closed-door meeting room for three solid weeks.

What did we bicker about? Things like "Are these the right granularity?," and "Should we use the word 'should' or 'will' here?" The use cases themselves weren't all that illuminating — basically the same CRUD patterns repeated for a couple handful of data structures. To me, it seemed an enormous waste of time. I was much happier trying to start building out the Smalltalk code that would implement them.

Construct use cases and UML — weren't themselves the problem. In fact, I still find pragmatic value in them today. The problem was the insistence that they should be produced in whole, to highly detailed depths, before software construction began — in other words without feedback to verify aspects of the proposed design. This dogmatic approach, per Hunt, was and still is the problem.

Reaching for the Horizon

As things got worse for Everest at MCI, a manager named Mike called about 20 of us into a room to brainstorm rescue ideas. He worked his way around the table, soliciting an idea from everyone there. I had no grandiose ideas; just one simple one related to a smaller but significant challenge: We were trying to consolidate half a dozen or so separate systems. Each system had its own notion of what a customer was, and the granularities were different — a single conglomerate customer in one system might appear as three separate customers in another, one for each subsidiary. We found a third-party solution that would purportedly handle the reconciliation for us — given a customer name, it would provide a unique cross-system identifier.

"We haven't produced any code," I said. "It would be nice to be able to demonstrate that we know how to build and deliver something, anything. What about the portion of the system that we'll need to reconcile customer names? It would be small, but I think it would be worthwhile to ship something, and then build on it."

"Yeah, that sounds nice," said Mike, "but it's just not going to be enough. We need something bigger." No one else came to the defense of my feeble idea. I suppose I must not have had the conviction to continue pushing it. Shortly thereafter, the project underwent yet another management upheaval and "they" replaced Smalltalk with C++. A few more management missteps and I realized the project was doomed. I also realized the biggest demotivator: to not be able to deliver anything. A year without delivery was my worst year of software development to that point, and also the worst to today.

About two-thirds of the journey through this three-year, $180 million waste, most people on the team saw the writing on the wall. If there was any doubt, we all realized the project was doomed when "they" renamed the project from Everest to Horizon. First words out of the mouths of many: "Oh yeah … that distant place you can see but never reach."

My worst year — a year without delivery — was a year lacking in pragmatism and high in dogmatic pronouncements. We had succumbed to an ideal: we could not deliver anything until the design and software was perfect and complete.

Abundance and XP

During the 1990s, Moore's law and other factors — a free market and an egalitarian attitude helped — meant that computing resources were becoming highly abundant. Hobbyists could now afford to buy modestly priced computers just to play with and explore. The World Wide Web meant that information about how to build software was becoming ubiquitous. We no longer needed to wait for those in the ivory tower to tell us what to do.

I discovered *The Pragmatic Programmer* and extreme programming (XP) at about the same time in 1999. For the most part, the ideas they promoted weren't terribly new — but many of them had been impractical up until that point. Test-driven development (TDD) is tough to do when build cycles are several minutes or worse. It's effective only when build cycles are near-instantaneous — something that Java and Smalltalk were able to provide at the time.

The core technical elements of XP seemed liberating to me. Yes, design is extremely important. It's so important that we need to find a way to think about design (and ensure it's kept clean) continually. XP offered several controls, both technical and supporting, to ensure that we could safely tweak the design as needed, at any given moment: TDD (with the ability to continually address design through refactoring built into its cycle), simple design, continuous integration, metaphor, and continual review via pairing.

Once you realize that it's possible to continually shape the design with low levels of risk and fear, you realize that producing a single up-front, speculative design is insufficient and even borders on malpractice.

XP seemed the very embodiment of pragmatism. It promoted a very code-centric, hands-on approach that allowed for continual design exploration. I thought of the TDD cycle as supporting a long chain of very short experiments.

XP was largely understood as a collection of practice, backed by a set of core values. That definition also led to its being perceived as overly prescriptive and thus dogmatic. The backlash, along with the rise of Scrum, led to folks not talking much about XP by the late 2000's. Was the pendulum swinging back to less pragmatism?

Wise teams continued to adopt many of XP's technical practices in order to help their codebase survive over time. They just stopped calling it XP.

Agile With a Big A

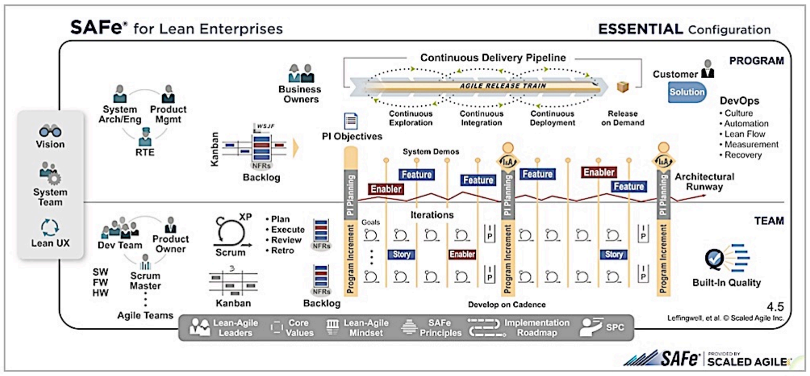
XP was one of the methodologies covered under the agile umbrella at the time of the signing of the Agile Manifesto in 2001. To support a larger umbrella, the manifesto's core values and principles assimilated ideas from other anti-heavyweight processes such as Scrum and Crystal. But that meant the umbrella had to push XP's technical practices out into the rain.

While Agile is largely a set of values and principles, teams still tend to gravitate to the ceremonial practices promoted by many of the processes under the umbrella. The daily standup (or "daily scrum") is a perfect example.

Daily standups were intended to kick off the day with a brief, pragmatic plan for how we'd work together. What they became in most organizations was evidence that dogma will rear its ugly head in the absence of continual vigilance. The typical standup looks like a glorified status report in three-question inquisition form, not a planning meeting.

In 2000, we used index cards to track things. I'm not recalling the price in 2000, but you can still get a pack of 100 cards for about a buck and a half. It was practical and prudent. Today, you can also choose to pay $245 per month for a team of ten users for one of the major "agile project management" tools. Even without a bulk discount, that will net you over 12,000 index cards per month.

You can also invest heavily in a framework that will help you scale your agile efforts. Here's a picture of one; it's small, because I'd rather not risk bogging you down in the considerable amount of depth required to understand this simplest-possible version of the framework.



In my estimation, high-performing Agile teams are fairly rare. A practical, sensible approach would suggest that we ensure we can get at least one team really working well before we sink a whole company into a scaling framework (SAFe, LeSS, DaD, LeadingAgile, etc.) and all the support it demands.

Over the years, as Agile moved across the chasm to mainstream success, Agile became co-opted by ideologues, dogmatics, and big-dollar products. Index card sales didn't increase much (you can't even find them in Europe!).

PragPub and Pragmatism

Hunt and Thomas published the first issue of the monthly *PragPub* in July 2009. Hunt kicked it off with the following summary: "*PragPub* is a look into the pragmatic world: it's a peek at what we're excited about, what our authors are excited about, and what we hope you can get excited about." Michael Swaine took on the editing role, and made sure it was clear that "this monthly publication is all about doing."

With that first issue, we were in the midst of a financial crash, so the publication led off with Andy Lester's article providing numerous tips on how to avoid and survive layoffs. After that, we read an interview with Rich Hickey, creator of Clojure — the Lisp-and-JVM-based functional language that had seen its first stable release only a couple months earlier. Stuart Halloway, author of Programming Clojure — also published only a couple months prior — immediately followed up with a very hands-on article about handling exceptions in the language. We were on track with the "early adopter" characteristic of pragmatic programmers! Dave Thomas wrapped up the feature articles as the subject of an interview on ebooks and "the future of publishing."

Each issue of *PragPub* comes packed not only with in-depth, feature articles, but also with a bevy of entertaining and useful columns: editorials, puzzles, programming challenges, conferences and speaker calendars, information on new books, and more!

I'm thankful to have been part of the ten years of *PragPub*. It's an honor to have published articles alongside the many well-known pragmatic folks whose work has appeared in the magazine.

A Changing World

Things in the software world have changed dramatically since that first issue! In 2009, developers typically pledged fealty to a single language — perhaps C++, Java, C#, or even Ruby — in what I might call monolingual organizations. Today, the typical programmer tends to be a polyglot working in an organization using three or more different programming languages. Per Andy Hunt: "A pragmatic programmer uses the right tool for the job, and knows why it's the right thing to do in this situation."

PragPub helped us get to the polyglot shop, providing us with numerous articles that not only presented overviews of new or boutique languages, but in many cases explored them in considerable depth. We learned about Scala, Swift, J, Haskell, F#, Clojure, Lua, Crystal, Go, Groovy, Elixir, Wolfram, and some old standbys like C, Python, Perl, Fortran, and COBOL. We dug into JavaScript and some of its frameworks and offshoots — PureScript, Angular 2, Ember, ClojureScript, CoffeeScript. And we also learned about the most predominant languages, too, with articles about C++, Java, and Ruby.

Tools and technologies? *PragPub* covered a wide variety — Awk, ASP.NET MVC, Re-Frame, Cucumber, GitHub, Vim, CSS, Sass, JSON, HTML5, TextMate, the Apple Watch, Xcode, Hadoop, Arduino, iOS, and Android, to name a few.

Techniques and concepts? We learned about legacy code, low-code, pair programming, testing, unit testing, microservices, mock objects, recursion, retrospectives, estimation, the Mikado Method, De Morgan's law, NoSQL, technical debt, technical blogging, how to teach kids, how to write well, responsive design, presentation skills, pomodoros, management, security, distributed development, home recording, how to coach others, QA, sales, interviewing, and how to get a job. Or how to hire for one. All in a hands-on practical manner.

We also learned a lot about computing history, to help us understand how we got here, as well as the mistakes we might be able to avoid a second time around.

What all these articles have in common is, of course, their focus on pragmatism. In a few short pages, we gained wisdom from experts on the topic — not just ivory tower idealists, but people who've actually seen the technologies applied on production software and projects.

Experimental and Controversial

Where I find the true value in *PragPub* is that it hasn't shied away from publishing articles about controversial new "experiments." From pair programming to mob programming to #NoEstimates, pragmatic authors have raised hackles and ruffled feathers by daring to challenge the status quo — particularly the status quo dogmatically promoting idealistic approaches. And more often than not, these challenging topics have held up to scrutiny and gotten well past the notion of being a fad.

You might not think that some of the topics *PragPub* has covered are at all controversial today:

- agile
- behavior-driven development (BDD)
- functional programming
- diversity
- distributed programming
- Bitcoin
- net neutrality
- test-driven development (TDD)

But they were all once highly controversial (and still are today to many people). That many have crossed the chasm is because pragmatic folks have been vetting them and writing about how to succeed with them over the past decade.

And if all that tech wasn't entertaining enough: Zombies were once a rarer breed of monster, perhaps controversial due to their statement about modern society. Nowadays zombie entertainment is rampant, and so through *PragPub* we got James Grenning to tell us how to do "TDD Guided by ZOMBIES" in June of last year.

May you enjoy our tenth year of *PragPub*, and to the next twenty plus one of your pragmatic career!

[1] Gutek, Gerald. *Philosophical, Ideological, and Theoretical Perspectives on Education*. 2016, Pearson, New Jersey. pp 76, 100.

[2] Hunt, Andrew and Thomas, David. *The Pragmatic Programmer*. 1999, Addision-Wesley Professional.

About the Author

Jeff Langr has been pragmatic most of his life, which includes 35+ years of software development. In addition to being a member of the Pragmatic Bookshelf technical advisory board, Jeff has written a handful of books (including *Modern C++ Programming With Test-Driven Development* [U1]), contributed to *Clean Code*, and written a hundred or more articles. He can help you with consulting, development, training, coaching, and mentoring services provided through Langr Software Solutions, Inc. Jeff resides in Colorado Springs.

External resources referenced in this article:

[U1] https://pragprog.com/book/lotdd/modern-c-programming-with-test-driven-development