The Pragmatic Bookshelf

PragPub

Keeping It Light

IN THIS ISSUE:

* Marcus Blankenship on distributed teams
* Chris Adamson on punk rock languages
* Eric Redmond on artificial intelligence
* Charlie Martin on cargo cult Agile
* Carmine Zaccagnino on Flutter

Not to mention... Antonio Cangiano gathering all the new tech books, Mike Swaine reading a Stephen Wolfram mystery, plus the PragPub puzzle!

Issue #123
September 2019

Contents

FEATURES

COLUMNS

BOOKS

DEPARTMENTS

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

Punk Rock Languages

In which our authors go mobile, get technical, and rock out.

As summer winds down up here in the northern hemisphere, we've harvested a bountiful crop of articles for the feast we call the September issue of *PragPub* magazine. Artificial intelligence, cross-platform mobile app development, the spirit of agile development, distributed team leadership, John Shade's minority report, and a celebration of punk rock languages: there's something for every taste and for hearty appetites.

Last month we began a series on artificial intelligence by Eric Redmond. In his first article he reviewed the history of AI and surveyed its modern applications. In this issue, he examines the technologies and other factors behind its current success.

This year we are celebrating ten years of *PragPub* by revisiting some of the classic articles of the past 120 issues. We're replaying an old favorite each issue, to let you read some of the classics. This month, it's Chris Adamson's paeon to what he calls punk rock languages, from way back in 2011. It's one of our favorites, and a fun read whether you agree with Chris or not. In true punk spirit, he'd probably prefer that you not.

Carmine Zaccagnino is writing a book on Flutter, Google's framework for multiplatform mobile app development. This month he discusses Flutter in the context of mobile development through the years. And Charlie Martin shares a tale of Cargo Cult Agile.

And there's more! Marcus Blankenship interviewing Katie Womersley, VP of Engineering at Buffer, plus a puzzle! We hope you enjoy this fat fall feast of *PragPub* fare.

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

back-of-book crew: BoB Crew

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

data protection officer: Michael Swaine mike@swaine.com

You can download this issue at any time and as often as you like in any or all of our three formats: pdf [U1], mobi [U2], or epub [U3].

Swaine's World

Ex Libris Alan Turing

by Michael Swaine

Stephen Wolfram plays detective.

Amazon, knowing me so well, just recommended $233 worth of books on quantum computing. I can be Gently Introduced to quantum computing, led to Mastery of quantum computing, advised on how to make Practical use of quantum computing. If I buy them all I'm sure I'll get a lesson in how to Make Money from quantum computing (write a book). I'd feel special to be singled out in this way except that MIT Press tells me that quantum computing is For Everyone. So far the discussion at our neighborhood potlucks hasn't stretched to include qubit twiddling, but I guess it's only a matter of time.

Books have been on my mind lately, sort of a recurring theme in my conversation and thoughts. It may be because I'm currently editing a half dozen of them, but I notice that I'm not the only one with a recent book fixation.

Stephen Wolfram, who celebrated his 60th birthday on August 29, has a blog [U1], and two days before his birthday he posted a long story about a book. It was a delicious narrative for anyone who loves the romance of books, or the decadent intellectual pleasure of following a thread of a puzzle beyond the point where most people would have given up.

Two years ago, Wolfram was given a book. It was a German-language edition of Paul Dirac's groundbreaking *The Principles of Quantum Mechanics*. Dirac was an English theoretical physicist who made fundamental contributions to quantum mechanics and quantum electrodynamics, and shared the 1933 Nobel Prize in Physics with Erwin Schrödinger. This German edition of Dirac's book had come to Wolfram through a circuitous route from the executor of the estate of Alan Turing, and the person who ultimately gifted it to Wolfram just thought he was the right person to own a book once owned by Alan Turing.

Wolfram already owned a copy of the book in the original English, and he didn't read German, but he nevertheless started looking through the book, page by page. And he began to come across marginal notes and underlined passages. Were these clues into the thinking of Alan Turing? He continued turning pages.

And then the scribbled notes began falling out.

At that point Wolfram put on his deerstalker and began tracing clues, trying to discover who had written the notes, and when, and what it all meant. He even enlisted the aid of a handwriting expert. You can follow the rest of the story here [U2]. It travels through some interesting territory, including lambda calculus, the price of books in Germany in the 1930s, the masthead of a German chemistry journal in 1933, the exact year that Cambridge phone numbers switched to four digits, the watermark on one of the notes, and why Paul

Graham named his startup accelerator "Y Combinator." Take a look. I think you'll enjoy it.

In other book news that caught my book-obsessed attention this month, Mike Elgan reports [U3] on two book-related software developments that have him worried.

First, software can now write books. That's bad, Mike says. "[T]he biggest threat to human intelligence is software that writes. The most efficient way for AI to make us dumber is to take the task of writing away from us. Our critical and creative faculties will atrophy. Our minds will become dull."

But his other scare story is actually about software *helping* us to write those novels. And it creates an experience that is anything but dull. Here's how he describes it:

"The Most Dangerous Writing App is a cloud-based word processor designed to end writer's block via the miracle of panic. You choose how long you're going to write (between 3 and 60 minutes) or how many words you're going to write (between 150 and 1,667). Once you've set your goal, you click the button and start typing. If you stop typing, your words will fade and vanish beyond recovery and you'll lose what you've written.... Their slogan is: 'Because 'tis better to have written and lost, than never to have written at all.' Hahaha!"

And one more thing: our brains don't care whether we're reading a book or listening to an audiobook [U4]. The content lights up the same patterns in the brain. Good news for my partner Nancy, whose new career is as an audiobook narrator.

New Manager's Playbook

An Interview with Katie Womersley

by Marcus Blankenship

Buffer is a company with a fully distributed team. Katie Womersley is its VP of engineering. Marcus interviews her about managing programmers.

Katie Womersley is an executive at Buffer, a fully distributed team.

mb: Tell us a little bit about you and maybe your journey to Buffer. And what is Buffer anyway, besides it's just a great company name, but what is it and what do you do there?

kw: Thank you. Yeah, Buffer is a great company name. I am VP of Engineering over there. We are a fully distributed team. We have people all around the world. We're currently 90 folks in the company overall, 35 in engineering. And we champion transparency and remote work in how we work. That's what we're known for. What we do is social media marketing. So for folks doing their marketing on social media, online-first consumer brands, most typically. We make it really easy to schedule all of your posts, engage with your audience, get deeper insights and analytics into how your efforts in social media have been doing. We create a whole suite of products that help make this a really easy and fun experience for people that are doing their marketing, advertising, engagement on social media networks.

mb: I am really excited about what we're going to talk about today because it is near and dear to my heart, and we're going to talk about how companies, and particularly how your company, develops managers and leaders. When I proposed this topic you said it was something that was exciting to you. One of the first things you told me was, "Well, there's a difference," and we should talk about that difference. Can we start there? How do you view the difference between managers and leaders?

kw: Well, Marcus, I would say that, ideally, in an organization everybody is a leader. Leadership is something that you embody in the way you take ownership of your work, the way you go about your tasks, the way you engage with your peers, with people in the organization beyond your team. Showing leadership is something that I want to see everybody do in every interaction.

Management is a very specific type of job. It's not about being a leader at all. It's about having a very specific set of duties where you are responsible for making sure that people's role expectations are clear. You're responsible for providing performance feedback, for providing coaching. You're responsible for sponsoring employees on project opportunities, making sure that they get promotions when they're deserving of that work. Management is a very … it's a nuts and bolts,

very specific job that needs to be done in an organization to make sure that people are aligned with the organization's overall goals in their work and that they are growing and getting recognized for the great work that they do. Or if there are people that need some corrections, that those are being done in, you know, a kind and in an effective manner. Management is a very specific, narrow role. It's a type of leadership. One type. If there's everybody in your organization, only a very, very small proportion of your leaders are going to be actual managers.

mb: Where do you think that leadership comes from? I hear a lot of people who just say, "Well, they're a natural leader."

kw: No, I actually don't agree with that, that there are natural leaders. I think that's something that we get this idea, perhaps in school, that some people are more assertive, and these people sort of have natural leadership. There's actually a lot of research that the best leaders, once promoted into typical leadership positions in very hierarchical organizations, like the military, the people who were actually more successful were not the ones that stood out as being loud, sharing their opinions, taking initiative in the previous position. They were actually the best followers, so this idea that leadership is something that you either have or don't have, I really question that.

I think it's a skill set like anything else. I think anybody can develop the skill set of leadership. Perhaps it might be that some people naturally gravitate to being more outspoken of their ideas, and as a society, there are certain traits that we think look "leader-y", like outspokenness, assertiveness, a manner of speaking, a sort of way of confidence. We think these things look like leadership, but I really don't agree that there's any such thing as intrinsic leadership; you're born with it or you just don't have it. I think it's a skill set like any other. I don't think anybody these days would say, "You're either born with design skills or not." No, you learn to be a designer, or you learn to be an engineer.

mb: I agree. I didn't come into this world knowing how to code or ride a bike or drive a car.

kw: No.

mb: Most things I didn't come in knowing how to do, so I had to learn them. For some reason, I do hear that idea that leaders are born is sort of sticking around in people's minds, and sometimes I think it bleeds through when they use the phrase, "Well, they're not cut out for it," is one I hear, as though we're sort of "cut out" in a particular shape.

kw: Right.

mb: Versus not being "cut out" for something, so I really agree with you. Now, that also means, though, that if leadership and leader skills are learn-able, how can companies promote the idea that everyone is going

to be a leader and that we can actually equip people to learn those skills?

kw: Well the first thing I think you need to do is be very specific about the actual skills and behaviors you want to see from folks. Sort of, a word like "leadership" means something different to everybody, so saying, "Everyone can be a leader. We should all embrace leadership," is not very helpful because people will just stare at you and think that sounds nice, but how?

One thing I do at Buffer is, I have a career framework for engineers. It starts right at Engineer 1, the sort of junior, entry-level engineering position, and there is an entire pillar called "Leadership." It'll have, for each level of engineering, what I expect from leadership skills at that position, so I have very clear leadership expectations for a junior engineer. That expectation is: you are able to give feedback privately to your manager when you notice something that doesn't make sense to you. You actually say that. It's a very concrete action. It's not something that you're cut out with, like speaking.

Everyone can surface to a manager, and that's, sort of, the junior engineer level there because we're not asking you to speak up in front of a group. We're not asking you to surface problems that'll challenge the organization six months down the line. We're just saying, "Raise to your manager when you notice something that doesn't make sense to you." And managers will prod. They'll say, "Look, I haven't noticed you doing this. It's an expectation we have under leadership that you point out to me when something doesn't make sense. You know, we know that there are things in our organization that don't make sense. I'd like to see you do that. Like, next week, please can you come to our one-to-one with one thing that you thought 'Hmm. I'm not sure that makes a lot of sense to me.'"

Being really, really, specific with what it is you want to see is the first thing because then people have a skill to actually build because leadership is not a skill. Leadership is an entire collection of behaviors and abilities that we put in this box, and the first thing you need to do for your organization is unpack that box and actually label each thing. For all the things you think of as "leadership," what is the exact skill? What is the exact ability or behavior that you want to see from people? And that does differ from organization to organization, depending on your values.

---

mb: So one of the things I think you mentioned, if you didn't mention it here, and maybe you did, was the idea of transparency. This is one of your company values. Do you feel like that ... ? I love that example you gave of a leadership skill, very concrete, that a junior engineer could start to practice with. Does that support the idea of transparency?

kw: Absolutely, because it's difficult to encourage people to do behaviors we associate with leadership, speaking up when things don't make sense, for example, when they're not seeing what's going on. You need to be transparent with what's actually happening in the organization in order

for people to start building that question-asking skill, that "Hey, I don't understand this" skill. They need to know what this is that's happening.

At the, sort of, more advanced levels, when you're asking people to anticipate challenges that are going to face the organization six months down the line, so for a senior engineer that's a behavior I'm expecting, right? But now how can a person possibly anticipate challenges if I'm very secretive with what's going on in the organization? If they don't have that context of what's happening today, how can they extrapolate to the impact of what we're doing today six months down the line? People can't be leaders, they can't practice these behaviors if they don't have the information about what's happening right here, right now.

mb: I love that. When you said the word "transparency" originally, my mental model was immediately top-down transparency. People at the top will tell the people at the bottom what is happening in a more transparent way, and that might be because most organizations struggle with that. But I also love that what you said was what you expect from your junior engineers in the area of leadership is bottom-up transparency. That when something, we can build a habit, that when something doesn't make sense, and I also like that you said, "And there's a lot, there is a lot of stuff that doesn't make sense. Let's not pretend, right?" We want to get people talking about it rather than pretending that everything makes sense.

kw: Absolutely, and it's really fascinating how, in a lot of cases where you see *Harvard Business Review* case studies of things that have gone really horribly wrong in organizations. Like Enron, what happened? How did this organization implode? The people at each level in the organization, the people doing the work knew what was going wrong. People typically are very aware of the massive problems facing your organization, and often it's executives that are the last to know what's going on.

We focus a lot on, as executives, sharing context, sharing vision, get everyone in line on strategy, but we don't focus on making it truly safe, and also clarifying the expectation, that we want everybody else in the organization to share, "How are things looking from your perspective? Because you are actually in the code. You are talking to the customer. You are out there making the sales. What are people saying about our product? What are the sort of problems that you're dealing with every day that we really probably shouldn't have anymore at this stage?" Those are things that executives don't know and often are the last to know, and they're often things that'll end up crippling a company down the line.

mb: Yeah, I was reading Amy Edmondson's book *The Fearless Organization*, which is about psychological safety. Highly recommended, but it uses a lot of those same ideas, and one of the thing it states is that executives are always wondering, "What don't I know?" Because while they're doing a lot of talking, and they may want to do a lot of listening, it's sometimes really hard to get real truth, real information about what's

happening. I just love that that is one of the things you expect from junior people because having been someone who held the title, frankly, of *Junior Engineer*, I didn't know they still handed those titles out, but I had that title, and I remember really feeling the title pressure that it was not going to be a good idea for me to ask those kinds of questions and bring up where I was confused.

kw: Right, exactly. I think there's an idea there that everybody who's been here longer knows more, and everybody who's more senior than me knows more and has seen more, and we miss out on the fresh eyes of people new to the organization and their sort of curiosity and perspective, the beginner's mind of people early on in their careers. I've found that, as managers, the nuts and bolts, you manage people, not leaders. We're all leaders. As managers, a lot of people these days are afraid of the words "I expect." I've found I say "I expect" a lot because it actually builds psychological safety.

It's not some junior engineer going out on a limb, being very unbelievably, riskily bold, saying, "Hey, I'm confused." We say, "I expect you to share with me regularly stuff that's confusing. It's there for sure. I expect this from you." So they don't have to then go out on a limb and be like, "Oh, I don't know if this is okay, but I had a question, is it all right for me to ask?" They'll know they're expected to ask. It's safe.

Having those expectations and clearly saying that they are expectations, putting them in a career framework somewhere, using the words "I expect" makes it safe for people to just do their job. Then they're actually leading. They're doing these behaviors, but we think of leadership as when you really push the boundaries, go outside your lane, and you're very brave and it's very risky. If you expand the boundaries of the job, if you expand those expectations and say, "Hey, we expect you to do all those things as part of your job," you remove the fear of people needing to bust out of their lane and be very, very bold. They're then able to say, "Oh, okay. This is expected of me," and rise to the expectation because it's just been clearly, humbly stated.

---

mb: I really like that, and now I'm curious about something. I noticed that when I was a junior person in an organization, even if no one said to me, "We want you to ask these questions," that having the novice mind and recognizing that I was new, somehow made it easier to ask questions. Sort of, I could scratch my head and say, "Maybe I don't understand, in parentheses, because I just got here, but what does this mean or why are we doing this?" Do you find that it can be harder the longer people are with an organization or the higher you go in title and responsibility to ask the dumb questions?

kw: You know, I really do think so. I think there's two things going on there. There's a sense that the higher you go in an organization, the more you should know, so people become embarrassed about asking questions because they start to think, "Everyone else is looking at me for answers. At this point, I should be the expert." There's almost the

fear that comes from thinking, "Everyone's going to look at me, and maybe I shouldn't be a senior engineer if I need to ask such an obvious question."

The second thing is just tenure, not necessarily seniority, but if you've been in an organization longer, you're more likely to have had a hand in creating whatever it is, the thing that doesn't make sense. As humans, we get defensive about our mistakes and we want to maintain consistency with our past selves. That's not effective, but it's very natural, it's very human. As we've been with a place for longer and longer, we start to see the messes, our personal mess, and can almost be a little bit defensive about how things are because we've been trying really hard, and we've been here really long. There's almost that kind of very human tendency to double down on things that we know really don't make sense because we did them.

mb: Yeah, now as the VP of Engineering, that's the top of the engineering organization, is what I'm guessing, do you ever struggle with wondering if you should know things and if it's okay to admit that you don't?

kw: Well, yes, I do. In my private moments, and I try not to let that get the better of me, to be honest. That's usually a sign, in myself, when I notice, "Oh, I feel like I should know this," or "This feels like a question I'm a bit embarrassed about." I take that as a sign of, "Interesting, there's a slight status threat I'm perceiving here. I am, at some level, worried about a challenge to my authority." That's the point when I know what I need to do is double down on the actual kind of leadership I want to role model in my organization and really go in and ask those questions and say "I don't know," and work through what it is that I'm worried about. Why is it that I'm suddenly worried about status or authority? Those are things that I think leaders don't usually like to admit that they think about.

I think, we all of us, as humans think about that, from time to time. I think it's just really important to admit that to yourself and be very honest. Yeah, it's scary to not have the answers, and then just go for it. Be like, "Okay, I don't know what we should do about this," or "This code review process isn't making a lot of sense to me right now. Is this the best way to do things? Maybe it is, I just feel unsure," and admit that.

In doing that, you role model to other leaders in the organization and people that have the sort of manager job description, that's the behavior you want to see. When it gets hard personally, I like to think about "Well, what would I want to see in an engineering manager?" And then go, "Okay, think about it as role modeling vulnerability." Then that makes it a little easier because then I can think of myself as teaching, even if what I actually right then is know the answers and be the best. We all know that's not true. We all know that voice needs to shut up and go home.

mb: I think that there … boy, there's a lot … thank you so much, first of all for letting me spring that question on you. It wasn't planned.

kw: Spring away.

mb: I recognize that it might be a bit difficult. I think that that's where so often there's this tendency for me to engage in perception management and really being concerned with how other people perceive me. I've seen leaders change the subject, begin yelling, walk out of a room, get really busy, quote, unquote. Often times, they will not say those things that you just said. They will not simply say, "I don't know what the right answer is. I don't know why this isn't working." Because when you admit you don't know, I feel like you open up new possibilities that everybody could start contributing. Maybe together we can *know* versus having to *be*. And especially as engineers, I think we have a tendency to love the facts and to love to know things and to feel like we're in command and control.

Well, let's move back to your career framework because I'd love, if you wouldn't mind, one of the questions I get a lot of is "How many levels do most of these career frameworks have?" I've talked to people who say three, and I've talked to other people who say 57, so I bet the sweet spot might be between three and 57 or maybe I'm completely wrong. Can you give us a sense of how many steps you use in your career framework?

kw: Right now, for makers, individual contributors that aren't managing people, we have six steps that are in active use. We have Engineers 1, 2, 3. That's junior engineer to our normal, professional level, like you're a great engineer. Then we have, you're actually taking on technical leadership responsibilities. Again, leadership is something everyone does, but there's a specific set of accountabilities that you have for the outcomes of your whole team. That it was makes a senior engineer role, and for those folks we have two levels. We have Senior Engineer 1 and Senior Engineer 2.

Then we have a further step in responsibility and accountability. At this point, you're taking on technical leadership accountability for decisions that'll impact groups of teams, perhaps all of product engineering, all of engineering overall. This level we call Staff Engineer. We do have a concept of Principal Engineer in our organization as well, but that's not one that we currently have at that level.

If it helps to think about it in terms of manager parity, which can clarify how advanced are these levels, engineering managers map to Senior Engineer 2s. Staff Engineer is equivalent to a manager who is a director. A Principal Engineer would be equivalent to a senior director level. Then we have a CTO and a VP role, where I'm a people-management track person and then we have a CTO who is an architecture-code, technical-decisions leader.

mb: Thank you so much. It sounds like, if I were to go back to being a Junior Engineer, and I worked at Buffer, at some point, I might be given an opportunity to go left or right. There'd be a fork in the road?

kw: Yes.

mb: Where we say, "Do you want to join the managerial track or do you want to stay on the technical track?" Am I getting that correctly?

kw: That's absolutely correct. Yeah, so you would go through Engineer 1, 2, 3. Senior Engineer 1. After you're a decent Senior Engineer, it would be a case of, "Do you want to become an engineering manager if there's an organizational need?" Of course, we need people that need a manager. Or do you want to become a Senior Engineer 2, a Staff Engineer, a Principal Engineer, and carry on with that track?

mb: Do you have folks who move back and forth between the tracks?

kw: We have had that. We have had somebody go from Staff Engineer to an engineering manager, so actually proceed on the technical track and then change to the management track later on in his career. Having been an engineer for 30 years and, having seen everything under the sun, wanted a different challenge, and is an absolutely amazing engineering manager. We've also seen folks go back and forth. I've seen a Senior Engineer go to engineering management, go back to the engineering track, and is now a Staff Engineer. You know, realized, "This just isn't for me," and that's fine.

We emphasize that, at Buffer, people management is a different role. It's not a promotion over and above engineering or out of individual contributed work. It's a different type of job. If you realize this job isn't for you, that is completely fine. You can switch back, and a lot of our most successful managers are people that do switch back between the tracks. A lot of really successful engineers have had a brief stint as an engineering manager, which maybe they hated, but it gave them an appreciation for the needs of product, the needs of the business, how to coach others in their career. That made them better Senior Engineers. We encourage that kind of switching back and forth.

The last thing we want to do is trap people in people management when they really don't like it and they're going to end up bad managers because if your person manager actually hates their job, they're not going to show up really engaged for their one-to-ones and their team planning meetings and the rest.

mb: Absolutely, and I've worked at those companies that said, "Well, if we move you into management and we have to move you back out, you'll never …" You get this sort of branded, like the Scarlet Letter, "You'll never get another chance." It's a career limiting move. It's very frustrating. I would love to hear, maybe I can ask you about your own journey into management. Where did you start and how did you end up as VP of Engineering at Buffer? Maybe just a quick overview.

kw: Well, the very quick soundbite would be hard work and well-placed organizational chaos at the time. To be entirely honest, I think that's the case for many of us that end up in leadership roles. I joined Buffer as an engineer at a time when we were just coming out of self-management. We had no managers at all. We had tried out something that sounds a little like holacracy. It was a bit different. I had joined this company that didn't believe in managers at all, and I joined as an engineer.

In my first six months at Buffer, I worked on every single team the company has. I changed teams about every five to six weeks. I started off writing a ton of code, trying to fix all the bugs. We had a bunch of quality problems. Writing lots and lots of features, working tremendously hard and realizing that, on these times I was joining, there were really fantastic engineers and the problem wasn't that we had engineers that couldn't solve their problems and needed more and more coders. We had problems like people feeling frustrated because their feedback wasn't being heard or an unstable product road map that kept changing. Lots of half-boiled features that would get abandoned and something else would get done. We would have a lack of clarity on how people can get promoted, so they weren't sure which efforts were going to be rewarded, and I found myself working with the engineers on my team and working with the product managers and trying to align these basic things into teams that made a little more sense.

I eventually went to the then-CTO and said, "I just want you to know, you hired me as a developer to write a bunch of code. I'm not really doing that terribly much, so I just want to tell you this because I know it's my job. I'm technically not doing my job, but what I am doing is this other thing, and it's really working. The different teams I'm going around to, they're a lot more successful after we sort out these kind of basic organizational issues." And the CTO was like, "Interesting," I couldn't quite tell what he thought, and then we chatted again. He said, "You know, this is something called Engineering Management." A lot of companies, they have this as an actual role. We then decided that I would become Buffer's first Engineering Manager, and I was the first person to then be an Engineering Manager. Then I took on that role, and we weren't quite sure what it was. I was then in a position of, you know, Googling a bunch, reading books, and reaching out to other people: "Do you have these things called Engineering Managers and what exactly do they do?" My first task as an Engineering Manager was to create the first career framework that we have. I did that. I put that in place. I created that. I did a lot of one-to-ones with engineers.

I kind of did that job, and then after doing that for about a year, our CTO moved on from the company, and there wasn't an immediate need to kind of replace that leadership vacuum, so myself, another person that was then an Engineering Manager, and a couple Staff Engineers, we all got together in a sort of coalition, and we led engineering somewhat by committee just trying to figure out: "How do we do things that make sense in this transition until such time as we get some kind of leader from the outside world?" We sort of expected

that someone would come in and, sort of, save the day. We were just going to do stuff that made sense up until that point and then just make sure that the future VP doesn't think, "Geez, that's a terrible decision." You just, sort of, do reasonable things, get some advice.

We did this for about six months, and then the CEO, our founder, Joel, had said to myself and one of the Staff Engineers at the time, "I think this has gone really well. You two have showed a lot of leadership. I'm going to make you both directors with different kind of focuses, and we're just going to keep doing this because this is actually working quite well, but there is no leader coming. This person you think might come fix it, that's not going to happen. We're not going to do that," which I found very alarming. I was excited to get the title, and I thought that was great, but I was very fearful inside because I kind of banked on I just sort of needed to keep things together for a little bit and then someone else that knew what they were doing. You know, the grownups would arrive.

mb: Right.

kw: I mean that still hasn't happened. We kind of did that. I think Dan and I just sort of stepped up. We're like, "Okay, directors." Did that role for, sort of, another year, and towards the end of last year, realized we actually have sort of a well-organized engineering team right now. We've managed to grow. We've managed to hire. We're at 35 people engineering. A bunch of other Engineering Managers that are all working well. Product is good, growing. Retention is good. It's like we think we've done a good job, and we sat down with the CEO and sort of officially moved to VP Engineering, CTO titles, and it was just based on, well, when you have a team and it is succeeding and these are all the things that we have happening, we think we've done the job. Our CEO agreed.

It was a case at every point there of just doing what I thought the company needed done, not really trying to carve out for myself a path. I think if I had sat down and thought, "I'm going to join this company, and in three and a half years, I want to be their VP Engineering. What am I going to do? How am I going to get rid of the current CTO?" I don't think that would have worked at all. It was just a case of "What's happening with this team right now? What does this team need?" And at each point, just providing that. It was like what the team needed was initially an Engineering Manager. Then they needed some kind of director-level. The team grew, and needed another Engineering Manager. Eventually, we needed a VP.

At each point, just working really hard, staying humble, asking my teammates, "What should we do? What are the problems?" Trying to listen a lot, and just trying to do the work to solve those problems. When I got stuck, asking for advice. I'd say the most useful thing I've done is just going outside of my company, getting coaches, looking to other leaders and saying, "Hey, I have this problem. How do you solve it?" Sometimes I'd hear, "No, everyone has that problem. That is a

normal problem to have." Sometimes I would hear how to solve it. Either way, it was useful.

mb: That is a wonderful story. I'm curious, as you went through the process, and maybe even as you reflect back on having now brought lots of other Engineering Managers into their new position at Buffer, do you see some pretty common bumps along that path that are difficult for engineers who step into that management role?

kw: The biggest bump is: Should I still keep coding and, if I don't, how am I going to still relate to engineers? What is my value if I'm no longer producing work, and how do I know that I'm productive? A lot of people that are driven to grow and progress along the path and then want to become managers to make the team more effective, they value productivity in a very fundamental way, and it's very difficult to let go of your own output as a person and rely on the output of others. That's a big bump I initially see.

Another big bump is not wanting to be authoritarian and then finding it very, very difficult to be clear. It's that kind of wanting people to read your mind, very indirect communication, trying to get your direct reports to guess where you're trying to go with something, and they're just getting really confused because you don't want to be that manager who says, "This is a terrible job. You should have done it this way. Bad job." You sort of ask them, "Do you think this code is good?" And then it's like, "I guess," and you're like, "Okay, well, do you think …?" And it's just sort of like Socratic methoding in this very awkward way. That's another big bump I see where it takes a while to build up that radical candor skill where you sit down to be like, "Hey, this code is not your best code. Here's what I see happening. What do you think?" It's not a big deal. It's not a whole emotional showdown. It's just candid exchange.

mb: I love that. I've heard that many times. "Do you think this is good?" And …

kw: "Well."

mb: "I don't know. Should I?"

kw: Yeah, it's like, "Are you going to praise me? Are you going to fire me?"

mb: Right, exactly. I know that when I became a manager, and I talk to lots of other folks who go through this transition, and they say the transition is like an identity crisis because they've been coding since they were in high school, or many years or something, and they've identified as this maker. I think the other thing that I heard from some people, and I'm curious if you've seen this as well, is if you've never had a good manager, someone who you really connected with, you might have the impression that no one does and that becoming a manager is fundamentally becoming like going to the Dark Side almost.

kw: Yeah, it's like people are very worried about that. I do see that. Also, so overwhelmed with all the negative patterns that they've seen that their mind is just so taken up with everything not to do, everything they don't want to be, that they're very paralyzed with, "Well, what can I do?" Because they've just never seen a manager that they thought was a good person and did a good job. They can almost be kind of paralyzed by it.

mb: Yeah, a few years back I realized that, I think there's a lot of bad manager prototypes in the media. You have Dilbert's pointy-hair boss, and you have the Office Space … I mean, we laugh at them, but I was trying to think, are there any great manager prototypes that we see? I just don't see very many, so that's one of the reasons I appreciate talking with you, as someone who's a great manager and leader. Even though …

kw: Thank you.

mb: I think that it's important to not only help people understand that the growing pains are real, and that they're common, and they're not alone. I just love the way you began our episode that it is a skill, it can be learned, and I'd even like to go a little further and just applaud you because I think organizations should be, and I think in particular engineering groups, should be teaching these things, rather than trying to hire from organizations that do teach them well. I think we need to be producing leaders and managers in our own organizations.

kw: I completely agree with that, Marcus. I think for the ecosystem it's important that we educate leaders and managers within our organizations, but purely from a self-interested perspective. If you're listening to this and you're thinking, "No, no. That's too risky. I'm just going to hire somebody who's a great leader," and you don't have the ability as an organization to grow them, you're always capping out at the level you're hiring into. You're always limiting yourself. You always need to over-hire for what you need because there's just no room for you to go with this person if you're not developing the skill set. And you'll get lucky. Some people will just develop themselves. That will happen, but you're going really miss out because most won't. Most people will do the job you hired them for, and then when you need the next level job done, you'll have to hire for that. It's very difficult if you're constantly hiring people over your existing team. You start getting morale challenges and retention challenges, and those are expensive problems.

mb: I was actually just going to bring up that I think a lot of engineers leave when their boss swaps out because they had this relationship, they trusted them, and now there's a new boss. They also take that as an opportunity to say, "Is this the right place, or is this the time for me to go find someone else to work for at a new place?" I think that every time you swap out a manager, you do run that risk that the people that

work for them … we're not just cogs, right? We build real, strong, trust relationships with the people we work for, and I think those are probably the most important relationships we have at work in a lot of ways.

kw: Right, I agree.

---

mb: Well, thank you for your time. Let's end with this. Do you have some favorite books or resources, things that are your go-to about leadership and management that maybe we could include in the show notes?

kw: Absolutely. Right now, a really great place to start is Camille Fournier's *The Manager's Path*. I'm sure you recommend that in many of your shows. It's very actionable, and it's got it broken down in each level, and it's specific to engineering organizations, which I really like. Going back to the old classic *High Output Management*. That's a great place to start. Andy Grove does, indeed, know what he's talking about. A book I really think everybody should read is *Radical Candor* by Kim Scott or just watching the TED Talk. I think if you're going to do one thing for your management style and for growing managers, find a way to put into words the things you know are holding your teammates back, and then help others to do the same. Create an environment of candor on your teams.

Then, Marcus, there's another book coming out that I'm very excited about. Laura Hogan has a new book, *Resilient Management*. I've read an advance copy of that, and it's really fantastic, so quick one for that. I've pre-ordered that for all my managers. Having seen a PDF, it's going to be a great resource.

About the Author

Nearly 20 years ago I made the leap from senior-level hacker to full-on tech lead. Practically overnight I went from writing code to being in charge of actual human beings.

Without training or guidance, I suddenly had to deliver entire products on time and under budget, hit huge company goals, and do it all with a smile on my face. I share what I've learned here in *PragPub* and here [U1].

The PragPub Puzzle

Vaping Venture

An alphabetic sudoku-slash-anagram poser.

IT'S PUZZLE TIME! Here's this month's Sudoku/Anagram brain-teaser.

It's a Sudoku, yes, but with letters. Just use the nine different letters in the grid to complete it so that every row, column, and small square contains each of the nine letters. When you solve the sudoku, you can rearrange the nine unique letters in the grid to form vapes' DeVilbliss Venturi devices.

T			E					
Z						O		
					R	S		I
		Z	A					
		E				Z		
A			O	M			S	
					I	T		E
M	S			R				A

Punk Rock Languages

A Polemic

by Chris Adamson

Chris wrote this for us over eight years ago. If he were writing it today he might change a few details. But it remains a truism that sometime you want a tool that's fast, messy, dangerous, and perfectly willing to kick your ass.

The year is 1978, and the first wave of punk rock is reaching its nihilistic peak with infamous U.K. band the Sex Pistols touring the United States and promptly breaking up by the time they reach the West Coast. Elsewhere, Brian Kernighan and Dennis Ritchie are putting the finishing touches on their book The C Programming Language [U1], which will become the *de facto* standardization of the language for years. While totally unrelated, these two events share a common bond: the ethos of both punk rock and C have lasted for decades, longer than anyone in 1978 could possibly have imagined.

And in many important ways, C is the programmer's punk rock: it's fast, messy, dangerous, and perfectly willing to kick your ass, but it's also an ideal antidote to the pretensions and vanities that plague so many new programming languages. In an era of virtual machines and managed environments, C is the original Punk Rock Language.

Lightning Strikes (Not Once But Twice)

This is a chord, this is another, this is a third. Now form a band.–Cartoon in *Sideburns* magazine

C has all the power of assembly language combined with all the elegance and poetry of… assembly language.–Erica Sadun [U2]

Punk rock emerged as a reaction to the increasingly self-indulgent and misguided musical trends of its time, with progressive rock insisting on merging in influences from jazz and classical music (typified by Emerson, Lake and Palmer performing Mussgorsky's "Pictures At An Exhibition" as a rock album), and second-rate guitarists noodling away in awe of Hendrix and Clapton with what Joey Ramone called "endless solos that went nowhere." Punk stripped away the nonsense of the times by focusing on faster, shorter, simpler songs. Preparing America for the Sex Pistols, *Time* magazine wrote [U3] "in the U.S. the movement is more purely musical: groups like the Ramones, Talking Heads, Television and Richard Hell and the Voidoids have rejected the rococo sophistication of much 1970s rock and turned back to basic buzz and blast."

C's development was, if anything, the opposite: it was meant as an alternative to programming the PDP-7 in assembly. In Dennis Ritchie's history of the language, he describes it as the result of several attempts to provide higher-level languages for the platform, a successor to BCPL and B (which Ritchie describes as "BCPL squeezed into 8K bytes of memory and filtered through [Ken] Thompson's brain"). But this development wasn't driven by academic niceties or intellectual noodling. It was about getting stuff done. As Ritchie recalls [U4]:

"By 1971, our miniature computer center was beginning to have users. We all wanted to create interesting software more easily. Using assembler was dreary enough that B, despite its performance problems, had been supplemented by a small library of useful service routines and was being used for more and more new programs."

Punk is famous for needing little more than three chords. C basically has three types—int, float, and char—embellished only by being able to increase their bit-length as longs and doubles. It's gallingly simple to modern eyes, but hones quite closely to what's actually on the CPU. Look through the x86 or ARM7 instruction set and you'll find no references to Unicode strings or "objects" of any kind; what you get is simple arithmetic and logical operations that work on operands of 8, 16, and (maybe) 32 bits. C is simple because, at their core, computers are simple.

Of course, you can build any amount of complexity you like atop this simple foundation. The American National Standards Institute (ANSI), spent most of the 80s codifying the C Standard Library, the collection of always-available libraries to perform essential operations like memory management, string manipulation, and networking. And you can then build your own libraries atop these. But as long as you're still working in C, it's fairly easy to keep your code running close to the metal, working with simple structures and function calls. Ritchie again:

"Despite some aspects mysterious to the beginner and occasionally even to the adept, C remains a simple and small language, translatable with simple and small compilers. Its types and operations are well-grounded in those provided by real machines, and for people used to how computers work, learning the idioms for generating time- and space-efficient programs is not difficult. At the same time the language is sufficiently abstracted from machine details that program portability can be achieved."

With the rise of object-oriented programming, new languages co-opted C to provide OO features: C++ has proven the most popular over time, while the Smalltalk-flavored Objective-C might well have remained a footnote had it not been adopted by NeXT, which led to its use in Apple's popular products. Even so, within these languages, you still mix procedural C calls with the object-oriented additions.

Police On My Back

Ever get the feeling you've been cheated?–Johnny Rotten, January 14, 1978

Segmentation fault–Command-line output of a C program with a broken pointer.

The largely unmanaged freedoms of C, C++, or Objective-C provide the opportunity to cause extraordinary havoc with your code, intentionally or not. What's most distinctive about C compared to other popular languages today is the exposure of memory pointers, and the disasters that come with their misuse. But then again, this is an intrinsic part of how computers work: you have a big block of memory (real or virtual), containing the system, your program, and other programs. If you can address any point in memory, and you read or write to an address that your program doesn't own, what should happen? One option is to allow programs to gallivant through each other's

memory space. As entertainment, this is "Core Wars"; as a coding error, it's a security nightmare. In the real world, such mistakes are prohibited by operating systems, and the offending program is terminated.

A lot of people find this price of failure too high, even though the fix is to *just write better code*, and instead opt for an even more draconian option: taking away the keys. Most modern languages provide their own memory management paradigms, and never allow the developer to see a pointer. Take a look at the monthly TIOBE Programming Community Index [U5], and you'll see three kinds of languages in the top 10:

- *C and its OO derivatives:* C, C++, Objective-C
- *Interpreted languages:* Python, PHP, Visual Basic, Perl
- *Virtual Machine languages:* Java, C#

The latter two groupings add profound new layers to the programming model: interpreters to parse and interpret program code, or virtual machines that create an entirely new execution environment for bytecode. Importantly, in both of these cases, the programmer's code is not the executable: run `ps` and you'll see that the interpreter or VM is what's running, not your code *per se*.

As we build abstraction upon abstraction, we get further and further away from what's really going on in the CPU and memory. In 2011, we now have popular scripting languages like Scala and Groovy whose interpreters run in the Java Virtual Machine. We've gone from working with the same data types that the CPU processes to an elaborate stack of layer after layer of artifice and illusion. If C is a punk band, blaring out the two or three basic types that the CPU understands, this new style of programming is Rick Wakeman hiring the London Symphony Orchestra to play backup for his synthesizer.

Now we have an Anti-IF Campaign [U6], which vows we will get more agile code by casting off `if` statements, replacing them (or at least moving the problem) through the use of elaborate subclassing and other language novelties.

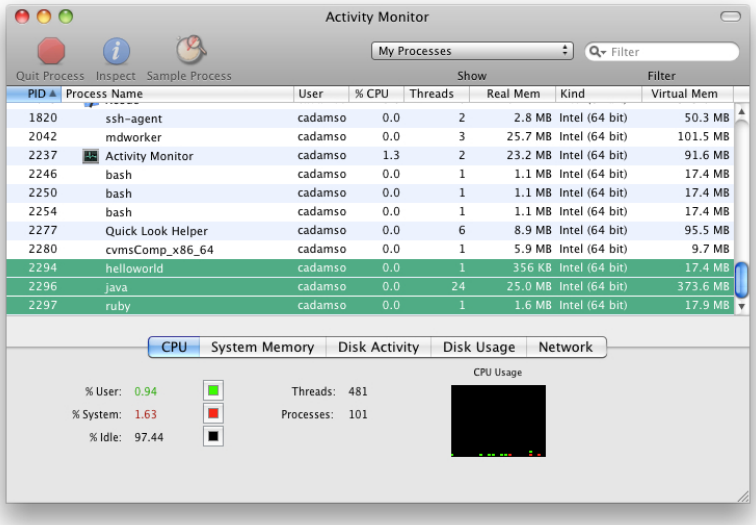
There's a price to pay for all this, the "abstraction cost" of creating general-purpose interpreters and virtual machines. Consider "Hello World" in C, Ruby, and Java, with an added sleep so the program doesn't immediately terminate:

```
#import "stdio.h"
int main(int argc, char *argv[]) {
  printf ("hello, world!\n");
  sleep (30);
}
```

```
puts 'Hello world'
sleep (30)
```

```
public class HelloWorld {
    public static void main (String[] args) {
        System.out.println ("Hello World!");
        try {
            Thread.sleep (30000);
        } catch (InterruptedException ie) {}
    }
}
```

Run all three of these and take a look at the resources they demand for this trivial program. As the figure illustrates, on Mac OS X the C program takes 356 KB, the Ruby interpreter 1.6 MB, and the Java Virtual Machine 25 MB. Java also starts up 24 threads for this trivial program.



Nobody Likes You

As someone remarked: There are only two kinds of programming languages: those people always bitch about and those nobody uses.–Bjarne Stroustrup

We are the cries of the class of '13 / Born in the year of humility / We are the desperate in the decline / Raised by the bastards of 1969—Green Day, "21st Century Breakdown"

Fans defend scripting and VM languages with arguments based around the idea of "programmer productivity": that faster CPUs and cheaper memory make it more economical to accept these abstraction costs, in the interest of getting working code out the door faster. Java has long been accused of tolerating bad programmers gladly, but that knock against it applies equally well to all the scripting and VM languages.

Furthermore, look who's making these languages popular: it's the web developers. All of the scripting and VM languages in the TIOBE top 10 are primarily associated with developing web applications. Yet when we cast our gaze over to the desktop and the mobile device, the scripting and VM languages largely disappear. PHP is clearly meant for web use, but Ruby and Python are general-purpose enough that they could be lashed to a UI framework to power desktop apps. In fact, Apple did just this in Mac OS X 10.5, making Ruby and Python first-class languages for Cocoa development. But in 10.6, the Ruby and Python templates have disappeared from Xcode. It appears that few developers took the bait, just as Cocoa-Java found few takers in earlier versions of Mac OS X and was retired.

(For those still willing to give scripting for OS X a go, Brian Marick's Programming Cocoa with Ruby [U7] is a fine introduction to this, wisely presenting Cocoa to the Ruby developer, instead of Ruby to the Cocoa developer.)

The VM languages have had some success on the desktop, most obviously C# in the Windows realm, where the go-to language was once Visual Basic, whose compiled P-code was also typically run by a VM. But Desktop Java's long miserable history of underachievement tarnishes the viability of desktop VMs, and with one exception (highlighted later in this article), there's little apparent real-world use of scripting languages on the desktop. Even on Linux, where scripting languages thrive on the server, GNOME, GIMP, vi, and emacs are written in C, while KDE, Firefox, Thunderbird, VLC, and OpenOffice.org are in C++.

Meanwhile on smartphones, the C languages are the only real option for Apple's market-leading iOS devices. And while Android uses the not-really-Java-nudge-wink "Dalvik" VM, it offers the Java Native Interface as an escape slide to C for performance-hungry applications like games.

The takeaway is that when your code is running in the immediate presence of the user—on his or her desktop, phone, game console, or tablet—the advantages touted by interpreted and VM languages suddenly diminish. Without network latency to hide behind, and with resources at a premium (particularly on the mobile device), the extravagances of the webapp stack suddenly becomes a lot less desirable. Users see slow, memory-intensive applications on their systems and say, as the last Clash album did, "Cut The Crap."

That C has won the end-user practicality battle is obvious to everyone except developers. Paul Graham's essay "The Hundred Year Language [U8]" manages to not mention C, despite the fact that it's already got 40 years under its belt, and has never ranked lower than #2 on the TIOBE list.

Come Out and Play

Well, you know, for me punk has always been about doing things your own way. What it represents for me is an ultimate freedom and sense of individuality. Which basically becomes a metaphor for life and the way you want to live it.–Billie Joe Armstrong

C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.–Dennis Ritchie

One of the defining traits of punk is the do-it-yourself (DIY) ethic, a rejection of the need to buy products or use existing systems, and instead to attend to your own needs. This attitude clearly suits C programming as well. Indeed, it defines the language's history, with the messy incompatibilities of early versions of the language on different hardware, and the development of wide-ranging libraries that don't always use like-minded idioms or patterns. As the decades have rolled on, a motley collection of C code has been developed by programmers around the world and across the decades, providing everything from XML parsing to 3D graphics, even if a libxml call looks nothing like an OpenGL one.

Some of these libraries are open-source and part of GNU. Others are commercial and liberally licensed. And there's an incalculable amount of

proprietary C code hidden away in businesses around the world. It doesn't matter: C has no licensing concerns of any kind. C isn't owned by a company (like Java, Objective-C, and C# effectively are), but neither is it a slave to the FSF or any other political movement. Any programmer can pick it up and use it for any cause, good or evil, that he or she sees fit. No $99 developer program memberships, no license agreements, no genuflecting to Richard Stallman... just code, compile, and run.

The Kids Aren't Alright

Punk is musical freedom. It's saying, doing and playing what you want. In Webster's terms, 'nirvana' means freedom from pain, suffering and the external world, and that's pretty close to my definition of Punk Rock.–Kurt Cobain [U9]

Having made the case for C as a sort of "punk rock language," it's worth asking what other languages share C's traits of ruthless usefulness. Let's continue the musical analogy: once the original wave of punk spent itself, its values were adopted by a number of successor genres, like post-punk, power pop, and new wave. Bands like The Clash and Television combined punk's immediacy with better chops and more outside influences. The Offspring and Green Day revived punk in the 90s with catchy melodies, launching an era of pop punk that persists to this day. In the early 90s, punk contributed to grunge, which helped end the era of glammy hair metal (sorry, Jon Bon Jovi and David Lee Roth). As long as there's a musical trend that's too full of itself, punk or one of its spin-offs will be waiting to tear it down.

What's the least pretentious language in widespread use today? Surely it has to be the much-derided JavaScript. Though it's one of many languages that inherits much of its syntax from C, it is an interpreted language, object-oriented and packed with novel features like dynamic typing, first-class functions, and closures.

It is possible to write elegant, clever code with JavaScript. In true punk rock fashion, almost nobody does.

JavaScript in anger is a brutally efficient language. Used primarily in the context of a browser, it is often used to slash out quick-and-dirty `onClick()` implementations. Despite being burdened with the confusing "Java" name, no class declarations, package and visibility statements, or other niceties are required; often enough, JavaScript code is rammed into the attribute of an HTML element.

Older developers bemoan the lost days when computers shipped with BASIC built in, or of learning scripting with HyperCard. "Where," they ask, "are today's kids learning how to program?" missing the browser that's staring them in the face. The web browser offers a JavaScript interpreter and enough toys in the DOM to keep the young programmer occupied for years. Anything that could be written in AppleSoft BASIC can just as easily be created with JavaScript and the `<canvas>` tag, and the presence of the tree-structured DOM and JavaScript's support for regular expressions give the young programmer access to more significant and interesting data structures than us old folks were ever going to develop on a middle school VIC-20. And thanks to the arms race between browser developers, JavaScript performance continues to improve by leaps and bounds.

Shouldn't beginning programmers learn from a more cleanly structured language? That's always been the complaint, but the slop of BASIC didn't harm my generation of developers (and mandatory Pascal didn't clean us up), so why should we fear that JavaScript will pollute the next generation? If anything, JavaScript's freedom to be as messy and unstructured as you want is a gift: once the mess gets out of hand, the young developers will have to figure out how to bring in classes, structure, OO, and the other tools that JavaScript provides to set things right. But it's DIY: JavaScript lets you be a good programmer, but doesn't force you to. How you structure your code and how you deal with problems is up to you.

Institutionalized

Don't do anything by half. If you love someone, love them with all your soul. When you go to work, work your ass off. When you hate someone, hate them until it hurts.–Henry Rollins

I've probably written an ungodly amount of C code, but I hate C with a passion. I've been writing a lot of Objective-C code lately… which just fuels my hatred for C. .h files are absolutely the work of the devil.–James Gosling [U10]

Not every language can be "punk." Not every language should—sometimes you do want the protections and comforts of the Web scripting languages, just as sometimes you'd rather listen to The Beatles than Black Flag. So, can we decide where to draw the line? Since C and JavaScript seem to have become punk rock languages through their evolution and use, not from their design, it might be more useful to list the traits that we associate with punk rock languages.

Here are some signs that you're working with a punk rock language:

- *Owned by nobody*—As pointed out before, C isn't owned by any company, nor is it controlled by the FSF or any other part of the free software movement (though it was clearly instrumental in the development of Unix, Linux, and the GNU software stack). Similarly, JavaScript is nobody's property. It's OK for these languages to have formal specifications, and to accept contributions from all sources, but no one body controls them, and that's a great thing.
- *Allows the user to apply as much or as little structure as he or she chooses*—It's not punk to keep the training wheels on all the time.
- *Is used in real-world systems or applications*—Perhaps we can create a category of "garage band languages" for those that aren't ready to play real gigs yet. But until it's useful to someone, a language with no users is always at risk of turning out to be a poseur.
- *The natural appeal of the language is to write software with it, not to mess with the language itself*—Solve your users' problems rather than indulging your own programming fetishes.
- *Inspires disdain, disgust, or outright hatred from people smart enough to know*—Any idiot with a Slashdot handle can talk crap about anything. It's when you piss off the smart developers that you know you're working with something interesting.

Feel free to come up with your own definitions and categorizations. After all, if there are too many rules, it's not punk anymore.

Good Riddance (Time of Your Life)

So, you have a choice. You can keep adopting the flavor-of-the-month language, each one giving you some beautiful new way to iterate over collections, so your method to collate all the user's previous transactions into a table can be done in 10 lines of code instead of 15. Or maybe you can use so much duck-typing that you get to add the string "4" to the number 2 (and be able to accurately predict whether you'll get the string "42" or the number 6). Or maybe the next language will do away with all the traditional branch and loop statements (`if`, `for`, `while`, etc.), in favor of some new exotic side-effect-driven style of programming that will be awesome as soon as you and everyone else "get it."

Or, maybe you can drop the pretensions, dismiss the illusions, and tear down the fake world of intents, monads, and whatever crazy new thing they come up with next.

What will be left is you, your code, and a CPU waiting to do stuff.

Kick... ass.

About the Author

Chris Adamson is a writer, editor, developer, and consultant specializing in media software development. He is the co-author, with Bill Dudney, of iPhone SDK Development [U11]. He has served as Editor for the developer websites ONJava and java.net. He maintains a corporate identity as Subsequently & Furthermore, Inc.[U12] and writes the [Time code];[U13] blog. He listened to The Clash, The Offspring, Green Day, and Fear while writing this article.

External resources referenced in this article:

[U1] http://cm.bell-labs.com/cm/cs/cbook/

[U2] http://twitter.com/#!/ericasadun/status/36176044994076672

[U3] http://www.time.com/time/magazine/article/0,9171,919281,00.html#ixzz1EcrXe4iC

[U4] http://cm.bell-labs.com/cm/cs/who/dmr/chist.html

[U5] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[U6] http://www.antiifcampaign.com/

[U7] http://pragprog.com/refer/pragpub21/titles/bmrc/programming-cocoa-with-ruby

[U8] http://www.paulgraham.com/hundred.html

[U9] http://www.nirvanaclub.com/info/nirvinfoa/quotes.txt

[U10] http://www.youtube.com/watch?v=9ei-rbULWoA

[U11] http://www.pragprog.com/refer/pragpub21/titles/amiphd/iphone-sdk-development

[U12] http://www.subfurther.com/

[U13] http://www.subfurther.com/blog

It's Just Artificial Intelligence

What's Different Now

by Eric Redmond

Artificial intelligence is not one field, but many. Eric builds a map of the whole complex landscape of AI.

Last month Eric covered the history of AI. This month he delves into the technologies themselves.

Fueled by a boom in a branch of AI called *machine learning* (ML), AI has emerged from an academic curiosity to changing the world in practice. How did we get here? A handful of shifts in the technology landscape have contributed to this AI resurgence over the past decade, and these core investments are poised to keep bringing new cognitive capabilities to bear on an ever-wider array of goods and services. These improvements have risen from three coequal changes: better hardware, democratization of algorithms and data, and increased investment in the ML space by both industry and academia. The changes in the industry are symbiotic. On March 2017, Google's CEO announced that they were now an "AI first" company, but they had invested in ML research for over a decade. Internal investment, academic partnerships, and acquisitions set the stage for its modern AI renaissance. Concurrently, they invested in an open ML toolkit called *TensorFlow* in an attempt to attract top talent and control the narrative on new applications. This coincided with the creation of custom hardware specialized to execute tensor calculations called *TPUs* (Tensor Processing Units). The world's largest collection of data didn't hurt either.

Better Hardware

While there are many ways to implement artificial intelligence, the current growth is based on machine learning, in large part because machine hardware has gotten so good in comparison to other artificial means, such as, synthetic biological research. While Moore's Law might be dead (transistor density doubling every 18 months), the kind of specialized hardware necessary for executing modern machine learning techniques continues to grow exponentially, thanks in large part to two unrelated trends: video game enthusiasts and cloud architectures.

The kind of hardware necessary for executing machine learning systems is similar to the hardware optimized for rendering video game graphics, called graphics processing units (GPUs). This is similar to the central processing unit (CPU) that has dominated the computing market for decades but optimized for the kind of math operations necessary for both use cases. With the proliferation of cheap and easily available GPUs, new life was breathed into the stagnant field of ML research. Google's play in this space upped the ante, as they rapidly prototyped, built, and deployed their first generation of production TPUs by 2015. The race was on to create specialized machine learning hardware called AI accelerators. Nvidia, the world's leading GPU

manufacturer, staged a concerted effort to dive into the greater AI market outside of Google's ecosystem.

While the AI hardware war rages, the cloud is the primary battleground, namely, Google Cloud, Amazon Web Services, and Microsoft Azure. The cloud allows anyone access to cutting edge hardware.

Democratization

While it's true that Google was an early mover in the ML space, most of the concepts and tools that they championed were developed elsewhere and, in many cases, better. What Google should be appreciated for was forcing other technology companies to open their IP. An open source ML library called Torch released in 2002 had been assisted by Facebook, IBM, and others, who contributed to the core project but were reticent to release how their particular sausages were made. But Google released more, not only the TensorFlow in 2015, but also a mountain of documentation, training videos, blog posts, and open-sourced algorithms that worked in production environments. The rapid popularity of TensorFlow ushered in an age of open, sharable machines based on a common language.

This openness in software, coupled with newly accessible powerful hardware, democratized the machine learning ecosystem. Suddenly, any smart kid with a cool idea has access to cutting edge ML research, with source code, and an environment to run it in. Another driver behind the democratization of machine learning is the flood of easy-to-use frameworks like the point-and-click *Orange* or *SageMaker*, coupled with easy-to-grasp education opportunities like *Udemy* or *Edx*. No longer would machine learning be reserved for those with a "freakish knack for manipulating abstract symbols" (via Paige Bailey of Microsoft paraphrasing Bret Victor of Apple). But open AI algorithms are only half of the story. Machine learning requires data to train with and run against, and data is increasingly everywhere.

The proliferation of open data, from public weather data, to university psychology research data, to government census and economic data (data.gov), has given many professional and budding AI engineers a set of data against which to build their own ML models. Often, those models are themselves open-sourced, allowing others to build on their work, prompting organizations to open even more datasets. This virtuous cycle of data to information has created new understandings in previously opaque industries, which prompts even more open data which democratizes machine learning even more. There is a constant stream of competition to create the best AI against a given dataset hosted on a site called Kaggle. They provide the data and the terms of the competition, and a dispersed community of data scientists compete to make the most sense of the data. These competitions often have a cash prize for the winners, and range anywhere from "Customer Revenue Prediction" to "Using News to Predict Stock Movement" to "Human Protein Image Atlas Classification," — and those were all in the same week.

Hardware as a service, machine learning lingua franca, open-sourced algorithms, open datasets, and easy-to-access education, all play a role in the democratization of machine learning. It's great that this infrastructure is opening up, but why would companies like Google give everything away?

Where does the money for the Kaggle prizes come from? Who pays for all of this?

Investment

Those of us who were around for the big data revolution quickly spotted a flaw in our operating model. While the emerging big data industry made it easier to quickly collect huge volumes of varied data (for example, by leveraging NoSQL datastores), it was not easy to make sense of the data. The general philosophy for the better part of a decade, starting around 2009, was, "collect all the data, we'll figure out what to do with it later." Turning data into information is a difficult task. Turning information into understanding at scale is nigh impossible for humans. A new, vaguely defined kind of job started popping up everywhere: Data Scientist. It was no longer good enough to hire statisticians, these unicorns also needed to be experts in large scale data management, to mine mountains of unstructured data for … something. The company made an investment in all of this data, just find something good.

Everyone had a sense that data was valuable, but there was no clear roadmap on what to do with that data. It was like a joke from the animated TV show *South Park*, where gnomes secretly stole underpants with no clear plan in sight: Step one, collect underpants, then step two, then step three is profit. There never was a step two, but they felt very strongly that it would lead to profit. We like to believe the coterie of swells slinging real money have clear goals in mind, but sometimes it's just worth taking a shot. It was highly unlikely that data would be worthless, and the emergence of ML as an increasingly popular sub-discipline of data science ended up being the missing piece to make sense of all this data sitting largely idle around corporate silos. The rapid increase in industry and government spending on limited AI resources makes sense through this lens, alongside the standard concerns about being left behind.

The symbiotic nature of academic and industry research is important for emerging technologies like machine learning. Academics conduct leading edge research across a variety of topics, and some percentage of that research ends up, hopefully, being of interest to the corporations and governments of the world. They, in turn, invest in more of the kind of research that they believe will yield better results. Right now, there's a goldmine of investment, which is only growing. Corporate investment in AI is on pace to have around 50% CAGR (compound annual growth rate) over the next decade. The general thrust of emerging AI research is maturing from machines that are better at describing the data, to predicting what's next, to the holy grail of AI, prescribing better courses of action.

Types of Machine Learning

AI continues to evolve from atavistic statistics which merely describe the world, to making sophisticated predictions, then prescribing courses of action for humans, to eventually taking action itself. The tools required to evolve these capabilities have become increasingly human-like in the manner in which they understand the world. While the perceptron was insufficient for much of anything with the technology of the 1950s, artificial neurons are the design of much modern AI via ML, in the form of deep neural networks. In a

short time, AI has moved from simple data-structures and symbolic algorithms to a complex artificial neurology, built on the rise of structures of available data and hardware.

Plain Old Data Science

In its most basic terms, statistics is about creating a model that estimate unknown parameters. Inference, probability, frequency, data science, machine learning, artificial intelligence, human intelligence … a rogue's gallery of approaches to attacking a basic problem: perfect knowledge is not possible, so how do we fill in the gaps when confronted with something new?

Statistics is as old as the first humans making assumptions based on observation. The Greek historian Thucydides described a frequentist method used in the 5th century BCE. One of the first modern statistical models tracked mortality by sampling for signaling of the Bubonic Plague (*Graunt*). Around the same time, the study of randomness was being used to calculate probabilities in games of chance (Pascal). Statistical work for demography and probability theory started to converge (Laplace) over the centuries, eventually consuming warring philosophies like frequentists and Bayesians into a general set of methods.

With the emergence of computers and the Big Data revolution, a new field of Data Science arose, with basically statisticians that know how to handle lots of data and can code computers a bit. Like other scientists, they test hypotheses against large datasets, using their own tools of the trade, namely, software packages like R or Python's *scikit*.

A good example of a useful statistic for extrapolating signals from incomplete knowledge is one of the simplest and oldest, called *linear regression*. Imagine you had a bunch of measurements of people's height compared to shoe size. While each of the dots in a 2D chart represent a single person, over a population of people a pattern emerges. Generally, the taller a person is, the larger his or her feet will be. Now, let's draw a line through what appears to be the average of each value. This line is our *prediction*. Say, the average 60-inch-tall person wears a shoe size of 8, while a 70-inch-tall person wears 11.

But is our prediction line good? To find out, we can measure the distance from each point to the line. That distance is called the error, because our estimate is wrong compared to the real observed value. We then square the errors (to make distant dots stick out even more) and add them all up. That total is called the sum of squared errors (SSE). What we want to do is draw a prediction line that best fits the data points we have, and this, has the lowest SSE, since that means overall our line is closest to the most points. Linear Regression is a method for calculating the best prediction line to the data set.

Although we'll never have a line that exactly predicts every new measurement, we only need one that is *close enough to be useful*. That's the crux of data science right there. We aren't looking for perfect, we're looking for useful. There's also no law that says you have to group together data points using straight lines. We can also try curved lines, a.k.a. *Nonlinear Regression*, such as would be the case when measuring average height by age. After people reach age 18, the

correlation between height and age tends to flatten out and even curve down a bit later in life.

Sometimes we want to analyze clusters of data (K-means) or reduce the dimensions or features in play (manifold learning) or convert from one type of data to another (auto encoding). Moreover, not all data is numeric. Sometimes we want to classify things, such as, is that a picture of a cat or a dog? The tools for accomplishing this feat are varied, including Logistic Regression, K-Nearest Neighbors, Support Vector Machine (SVM), Decision Trees, Random Forests and so on. How can a data scientist possibly know which algorithms to use, let alone figure out how to fit the chosen model to the data set?

The answer is, we start letting the machines do the work for us. We educate a model with a *training* data set and look to reduce its errors against a *validation* data set. Then check how generally useful that model is with a *test* data set. In other words, we teach the machine how to learn by fitting a prediction curve with the common pattern of training, validation, and testing.

Deep Neural Nets Machine Learning = Deep Learning

"Excellence is an art won by training and habituation … we are what we repeatedly do." — Aristotle via Will Durant

Artificial Neural Networks (*ANNs*) are loosely inspired by biological neocortex neural clusters. Deep Neural Networks (*DNNs*) are ANNs where there are many layers of neurons between the input and output, in other words, the network is deep. *Deep Learning* is basically training a DNN with loads of data, until the network starts to be "shaped" by the commonalities in that data set. Say you train a DNN with many images of cats. With enough images, the DNN will start to recognize the common attributes that make up a cat. When you give it a picture it hasn't seen before, it can pick out whether the image contains a cat with some level of confidence. Machine Learning is not magic, it's just multidimensional curve-fitting.

Consider an audio waveform. Since sound is just vibrations in the air, it was long ago discovered that unique sounds can be collapsed into one unique wave. Let's say we have a series of waveforms of different people saying the word "hello." Something about the waves are similar enough that any human capable of hearing it could make out the word "hello" (even if they don't speak English). While the waves might not look exactly the same, there is some commonality that can be extracted by a crafty algorithm.

Send many samples of the audio waveform for "hello" to a DNN, and it may, through a series of weights and adjustments to its internal structure, start to recognize the pattern in any waveform. Then, as you stream in a series of waveforms of, say, a conversation, it can pick out a particular wave segment as the word "hello." This vaguely mimics our human neurons recognizing a friend shouting "hello" across a crowded room.

It turns out the ability to recognize patterns in a set of noisy data is similar whether you're talking about audio, images, video, electrical pulses, financial data, or many other signals. The mathematical representation of these data is

called a *tensor* (think of a multidimensional matrix). As long as the data can be converted into a tensor, it's a candidate for deep learning, and it turns out, most everything humans interact with is.

DNNs and Deep Learning open up a world of ML techniques for various uses. The "hello" example above was a way to classify an observation and is becoming a rather milquetoast technique in the ML space. It turns out, given the right incentives, you can teach machines to do more than observe, but also take action.

Supervised, Unsupervised, and Reinforcement

The examples we've covered generally fall into a category called *supervised machine learning* (SL). The word "supervised" here means our training data has a value (dependent variable) that represents what we're training the AI to predict. If we want to train a *convolutional neural network* (CNN) to recognize images that have cats in them, we have to train it with many other images of cats that humans have already labeled as cats. If you've ever run across a modern Google CAPTCHA (security) test, you may have been provided a series of images and been told something like, "select images containing a stop sign." That's Google using you to label its images, so it can later train a DNN to recognize stop signs automatically. You, dear human, are the machine's supervisor.

This raises the question, is there an *unsupervised machine learning* (UL)? When you have a lot of data and want to detect patterns but aren't really sure what those patterns might be yet, you can't label it. Imagine you have a series of warehouses, where inbound products are scanned, and their geolocation is tracked. Providing only these locations, certain algorithms (isolation forest, k-means clustering, variable auto encoders, etc.) learn where your products are expected to be. Given a scan somewhere that's unexpected, like Antarctica, can trigger an alert that a signal is an anomaly. Unsupervised machine learning is commonly used in many kinds of anomaly detection, from logistics, to bank fraud, to consumer profiling, and recommendation systems … anytime you're looking for a good representative form from the data.

Recently, ML practitioners are playing with combining SL components with UL into a type of ML called *generative adversarial networks* (GAN). We could call GANs semi-supervised models, because they're trained by arming an unsupervised model to learn (and generate) against a known supervised model (the adversary). The details are deeper than we need to go here, but GANs are interestingly powerful for taking bodies of known works and generating outputs that are similar enough to be useful. Such as, providing a corpus of classical music, and asking the GAN to generate an endless supply of new music that somewhat "sounds like" the input. Well-trained GANs are excellent for generating new creative endeavors, such as an endless selection of custom sneakers. In early 2019, an *OpenAI* GAN was claimed to be so good at generating convincing fake news articles, it was deemed too dangerous to release to the public.

In the history of psychological Behaviorism, Pavlov's Dog tends to be the experiment that most of us would shout at a trivia night. But years before Ivan Pavlov rang his feeding bells, Edward Thorndike discovered the "law of effect,"

which states that satisfying consequences tend to be repeated, giving rise to a study called Operant Conditioning. This is the crux of a third type of machine learning, called *reinforcement learning* (RL). Unlike supervised or unsupervised machine learning, reinforcement learning goes beyond data per se, and instead focuses on training *agents* to act in a given *environment*. We train these agents in the same way that we tend to train other humans, by *rewarding* desired behaviors and *punishing* undesirable ones, in service of some *goal*. Reinforcement learning, as a computer concept, is decades old. However, RL is experiencing a renaissance, thanks to the emergence of deep neural networks, most famously Deep Q-Networks (DQN). RL tends to be the modern tool of choice for training machines to do things that have recently been done by people, such as training machines to beat world champions at Go (*AlphaGo*) with the goal of winning, or training computers to trade stock autonomously better and faster than any human could with the goal of higher profits. Of all the tools in the roboticist's toolkit, reinforcement learning may become the most disruptive.

When you tie together RL, SL, and UL, you can get a sense of the next few decades of AI research. Consider autonomous vehicles. You could use cameras and CNN (SL) models to see and detect objects, use isolation forest (UL) to judge whether the objects together make common sense — like a snowman on a palm tree — and DQN (RL) a car to react based on the best available knowledge with the goal of driving down the road without hitting a living thing or being hit. While the cutting-edge researchers continue to build increasingly sophisticated ML models, us mortals can put them together like puzzle pieces in new and interesting ways.

Next month: downsides, dangers, and work to do.

About Eric

Eric Redmond has been in the software industry for over 15 years, working with Fortune 500 companies, governments, and many startups. He is a coder, author, illustrator, international speaker, and active organizer of too many technology groups in Portland, OR.

Agile Is a Thing of the Spirit

Cargo Cult Agile

by Charlie Martin

In that project, we adopted the trappings of agile as sympathetic magic.

Almost 20 years ago, the Agile Manifesto proposed a change in the way software is developed that values:

- *Individuals and interactions* over processes and tools
- *Working software* over comprehensive documentation
- *Customer collaboration* over contract negotiation
- *Responding to change* over following a plan

Twenty years later, people *still* don't understand how radical that was — including a lot of people who think they are practicing agile.

Back in 2008, I wrote about a project which I called Cargo Cult Agile [U1].

The cargo cults arose in Micronesia during and just after World War Two. During the war, as the United States executed the island-hopping strategy, the Navy developed airfields on the islands. Along with the airfields, they brought canned food, material, built temporary housing, and passed along things like canned pineapple that the indigenous occupants of the island had never seen. And then the war ended, and the "cargo" stopped coming. To try to get the cargo back, the native occupants built their own airfields, hangars, even airplanes, all out of bamboo and palms. It was sympathetic magic.

In that project, we adopted the trappings of agile as sympathetic magic. Under the project manager, we adopted use cases for the specifications, morning stand-up meetings every day, and incremental development.

But, per the project manager, we still had to have a complete schedule to final delivery as a Microsoft Project Gantt chart, so, to make an estimate, we had to accumulate what we thought would be *all* the use cases. We set up a wiki for them, and started accumulating use cases, until we had something like 900 use cases, which then let us make the Gantt chart, out for the 13 (as I recall) months of the schedule. We started planning incremental deliveries, but for various bureaucratic reasons it was a lot of work to "deliver" an increment for testing, so what started being a planned one-month sprint very quickly was modified to be a … six-month sprint.

Of course, once we started, the requirements did what requirements always do: they changed. Some of the assumptions we made at first turned out to be wrong. This was a highly-secure application that was supposed to meet FIPS140-3 standards. No one had looked into what that would imply when we made up the use cases, and actually meeting the standard was much more difficult and complicated than anyone expected. Then we realized that we hadn't given any thought at all to how the software would be delivered, which for an appliance was much more complicated than expected.

In the meantime, the morning stand-ups changed from the traditional three questions — What did you accomplish yesterday? What are your plans today? Is there anything blocking your progress? — to an hour-long sit-down complaints and problem-solving meeting.

Obviously, whatever agile-like ornamentation we'd adopted, this was a standard old-fashioned waterfall-style project.

When the project was less than successful — as it was — one conclusion was that "agile" was not particularly effective.

In the intervening eleven years as a consultant, I've seen several other projects in which "agile" was attempted unsuccessfully. Some projects felt they needed complicated tracking, rather than the Extreme Programming 3-x-5-inch cards, and so adopted software that was to help manage the user stories (we'd replaced "use cases" with "user stories" when formal user stories turned into massive documents with rigorous formats) and then added epics and comment tracking and automated links to the configuration-management system, until managers and developers are spending hours a week simply managing the tool.

In other projects, progress didn't seem to be fast enough — so regular meetings were instituted to urge people to work faster. In my most recent experience, this turned into 8 hours of meetings a week for 50 developers.

Yes, 400 staff-hours a week. That's 10 staff-weeks.

And yes, we had to walk to work through the snow, five miles each way.

The point is that over and over again, we "adopted agile" while violating every one of the principles. Processes and tools began to dominate the work. Thoroughly documented user stories took time from development. The inevitable changes in requirements broke the plan — and then the plan struck back.

The key observation, I think, is that programmers like tools and processes and making things look like programming, and managers like to *manage*. And the key to agile is to recognize that when you have volatile requirements, and "unknown unknowns" cropping up as development proceeds, you can *never* have complete control.

Programmers like systems, methodologies, structure, and there have been profuse attempts to make "agile" into a systematic, structured, follow-the-rules methodology. This leads to "cargo cult agile." A project adopts the outward signs of agile — stand-up meetings, use cases or user stories, iteration — as a kind of sympathetic magic, hoping that if they erect bamboo airplanes, the tins of pineapple will start arriving again.

We need to — once again — break out of the "structured methodology" mindset.

- Agile isn't tools, and complicated tools can keep you from being agile.
- Agile isn't stand-up meetings and sprints, but you can spend a lot of unproductive time on them.
- Agile isn't SCRUM or Kanban or pair programming or test- and behavior-driven design, although if you *are* agile they can be helpful.

Agile is what happens when you commit to uncertainty and the inevitability of change. It means recognizing that you can't have a firm schedule with a fixed set of requirements and a fixed end-date. It means looking at every new form, proposed meeting, new tool, and last minute change, and asking "Am I really going to need this?"

"Agile" means learning to say "no" — and expecting that when you say no, you'll be supported. "Agile" means not just changing the way you program, or document, or deliver. It means changing the way you *think*. Agile is accepting, committing, to not having complete control. Agile is not a "methodology" — agile is a thing of the spirit.

About the Author

Charlie Martin has been a professional programmer since we used stone knives to punch holes in Hollerith cards made of animal skins. During and after graduate school in Computer Science (MS Duke '88), he often supported himself as an instructor in computer topics, for such customers as Learning Tree, IBM, Digital, NASA, and the Department of Defense. He has worked as a senior developer for (among others) IBM, Sun Microsystems, StorageTek, and SGI, and was CTO of an Internet marketing startup, Sumazi.

Currently, Mr. Martin is a writer with hundreds of published articles and an instructor, primarily through Wyzant [U2]. Watch for his upcoming book at PragProg.

External resources referenced in this article:

[U1] https://www.cio.com/article/2434415/cargo-cult-methodology--how-agile-can-go-terribly--terribly-wrong.html

[U2] https://www.wyzant.com/Tutors/CharlieM

The Smartphone Revolution

Making Everything Accessible

by Carmine Zaccagnino

Smartphones became ubiquitous because people found them intuitive. And that presents a challenge to the multiplatform smartphone developer.

What really made the smartphone revolution happen wasn't big screens or full web browsing, it was apps. The iPhone (just like the Android phones) did away with the previous operating systems (Symbian, mostly) that were designed for regular cellphones and that had very few apps available for general consumers.

Smartphones took off because they are so useful to so many people, and that's because there are so many applications available. Apps that you can interact with in an intuitive way while on the go. The actions that would have required a PC (checking whether a bank transfer was completed, checking email or even SSHing into a remote server) can all be done quickly with one hand while commuting on a crowded bus (or in a car if someone is giving you a lift.)

The smartphone was to the computer what the cellphone was to the landline phone: it provides the same basic functions but is more accessible and flexible. And that's the challenge for developers: both those attributes are critical for the apps developed for smartphones: they have to look and work the way the user expects, and empower users to perform everyday actions using this tiny device with (relatively speaking) not much power.

The Original SDKs and Early Competitors

When the iPhone came out, the solution was to integrate some of the navigation actions that used to be reliant on the presence of physical buttons into the apps themselves, which made it a priority to provide a consistent experience across first-party and third-party apps if there was to be any hope of third-party apps becoming mainstream. This resulted in the creation of SDKs specific to Android and iOS by Google and Apple respectively, each relying on the programming language and development style that was closest to the vision of the company releasing the SDK: Android developers had to use the Java language, and iOS developers had to use Objective-C.

This meant that for the first few years there were many instances of apps only being developed for one platform, especially those developed by small companies or independent (sometimes hobbyist) developers who, encouraged by the widespread adoption of smartphones by consumers who started downloading apps and trying to use their phone for more than just calling and texting, started creating apps that appealed to the average person.

Think of a hobbyist mobile app developer, who has a day job and has discovered the possibility of creating apps that reach a huge audience and could potentially be rewarding economically. Such a person doesn't necessarily have the time to learn how to create apps with both the iOS SDK and the Android SDK.

They probably don't even have both an Android and an iOS device to test the app on. This means they will only release the app for one platform.

Not everyone was doing that, though: a platform that worked on Android, iOS, and desktop already existed: the Web. The appearance of parallel stripped-down mobile-oriented websites helped people get some very simple tasks done on their mobile devices. Some of these websites were very usable (but still very slow when compared to installed apps), but some were either hard to use, lost too many features, or weren't updated often or at all when the full desktop website got updated.

So, in cases where it was feasible (not much access to hardware necessary, no performance concerns), many smartphone apps were really just web apps, with native interfaces that were not much more than icons on the home screen to launch a *WebView* (a web page, rendered by Chrome or Safari, that can be integrated into an app's interface) that loaded the web app directly using the device's internet connection.

But there were also tools like *Cordova/PhoneGap*, which allowed developers to write apps using HTML, CSS, and JavaScript that worked offline and also allowed access to some of the device's sensors and interfaces in a consistent way, without relying on the user's browser to support each feature. Essentially, though, these also relied on WebViews under the hood. And the general performance, look, and feel weren't nearly as good as what could be achieved with a native SDK.

Dialing Back Some of the Pain

To lure some developers back into native SDKs — and since both Java and Objective-C soon started to feel inadequate and cumbersome — Apple decided to give iOS developers the option to use a new language called Swift instead of Objective-C, and in 2017 Google decided to officially support Kotlin as an alternative to Java for Android development. These were regarded as positive changes by most developers and some people's interest in the native SDKs grew. But the need to support both the most popular mobile operating system worldwide (Android) and the most profitable for many apps (iOS, also the most popular in the United States) was made all the more apparent by the 50/50 split in OS market share in North America for the past 4 or 5 years and the continuous loss in worldwide market share for iOS.

Furthermore, UI work on both Android and iOS gave a choice between a fixed declarative UI defined in separate files from the app code, with the navigation between them defined in the app code with constructs that tended to become unintuitive and cumbersome when more functionality needed to be added, or programmatically defining the UI, which sometimes made it difficult to figure out the final UI that gets generated from the code when looking at a codebase written by somebody else and also required many lines of code for comparatively easy tasks.

One of the latest entries into the cross-platform framework market was Facebook's *React Native*, borrowing app structure and use of the JavaScript language from *React* (also by Facebook and quite popular in the web development space).

It was used by some high-profile apps like Instagram. The Instagram team themselves documented online that one of the hardest things to achieve using React was to keep the method count down by only pulling the parts of React Native they needed in order to avoid having to make the Android version of their app multi-dexed.

In case you're not an Android developer, that means the app has to be built into multiple Dalvik executable files (.dex extension, hence the single-dexed/multi-dexed name for apps that are under/over the 64K method limit for a single DEX file), which dramatically increases build time, app size, memory usage, and startup times and is generally thought to be something to avoid if possible because it also requires additional workarounds for some older Android versions. (On an unrelated note, going multi-dexed is sometimes also caused by the usage of Google Play Services which contains tens of thousands of methods, but that's very hard to avoid for some apps which make heavy use of such services.)

The advantage React Native has over the HTML frameworks is that it uses native UI elements (the same used by the native SDKs), so it looks at home on each device it runs on. But this turns into a disadvantage if you need predictability and consistency in your UI. Because React Native apps will look different on different OS versions, and you can only see the difference if you run the app on both operating systems. What was needed was a framework that gives complete control of this to the developer.

The Best of Both

Over the last few years, Google has been promoting its own alternative to the previously mentioned cross-platform frameworks: *Flutter*.

Flutter is developed by Google as an open-source project. It allows developers to build cross-platform apps with very little boilerplate code when compared to native apps, that integrate declarative UI code into the app code in a way that also makes it very easy to both interact with the UI programmatically and to understand what the app will render. Each UI element (called a widget) is defined in a class, and each widget has a build method that gets called every time it needs to be rendered and that method returns, in turn, the widgets it should contain.

This way of using composition to build UIs makes it very easy for beginners to build nice-looking and functioning user interfaces.

Flutter supports both the Material Design UI elements and iOS-like UI elements, and it allows you to automatically select the right one depending on the platform the app is running on, so your users will never know whether the app was built using the native SDK or using Flutter, except for one thing, which I'll explain now.

Flutter doesn't use native UI elements like React Native or the native SDKs do: in reality, it replicates them using their own low-level rendering engine, ensuring higher maximum frame-rates, greater control over the UI by the developer, and the same look across OS versions. Also, as proof of the complete independence of Flutter from platform-specific aspects, you can use iOS widgets on your Android app and/or Material widgets on your iOS apps, if you so desire.

This can be used to test the usability and look of both the iOS UI and the Android UI on the same physical device.

It also has built-in state management that simplifies the development process for many apps. In fact, building Flutter apps is incredibly easy and it's surely bound to ensure that no app is developed for just one platform and fewer apps are developed with the slow and out-of-place-looking web-based frameworks.

The mobile industry in general has demonstrated already that failing to adapt to new standards can lead to unfavorable results. Now, more than ever, mobile app developers need to be able to support both Android and iOS: the strengths and weaknesses of each are being highlighted continuously. Furthermore, the possible replacement of Android with Fuchsia makes Flutter all the more appealing.

Editor's Note: Clearly, Carmine is a fan of Flutter. If Flutter interests you in the slightest, there is good news: learning Flutter is made really easy by a growing community that can help at any time, and there is a book about it called *Programming Flutter: Native, Cross-Platform Apps the Easy Way* [U1] by none other than Carmine Zaccagnino. Further disclosure: I am Carmine's editor on the book.

About the Author

I am a web and mobile developer and have struggled for years building Android apps using the standard SDK and, in a lesser way, Web-based tools. My experience in development areas other than mobile development has led me to be particularly bothered by the lack of a native (or close to native) framework that can bridge together Android and iOS without losing low-level access to hardware and software, until Flutter did exactly what was needed.

External resources referenced in this article:

[U1] https://pragprog.com/book/czflutr/programming-flutter

Antonio on Books

The Rise of Rust

by Antonio Cangiano

A new edition of Antonio Cangiano's excellent Technical Blogging [U3] is now available!

Rust may never become as popular as Python or JavaScript. It's too complicated and "low level" to do so. Slowly and surely, however, Rust is carving out a fairly large community of people who swear by it. So much so that it's been voted the most beloved programming language in the StackOverflow developer survey for four years running now.

If you're curious and would like to learn more about what all the fuss is about, you need to look no further than this month's pick: *The Rust Programming Language*, published by No Starch Press. This is the official book of the Rust community and it has now been updated for Rust 2018.

The Rust Programming Language [U4] • By *Steve Klabnik, Carol Nichols* • ISBN: *1718500440* • Publisher: *No Starch Press* • Publication date: *August 12, 2019* • Binding: *Paperback* • Estimated price: *$26.91*

Programming Kubernetes: Developing Cloud-Native Applications [U5] • By *Michael Hausenblas, Stefan Schimanski* • ISBN: *1492047104* • Publisher: *O'Reilly Media* • Publication date: *August 6, 2019* • Binding: *Paperback* • Estimated price: *$34.54*

Computer and Information Science [U6] • By *Editors at Springer* • ISBN: *3030252124* • Publisher: *Springer* • Publication date: *August 6, 2019* • Binding: *Hardcover* • Estimated price: *$128.60*

5 Steps to a 5: AP Computer Science A 2020 [U7] • By *Dean R. Johnson* • ISBN: *1260454916* • Publisher: *McGraw-Hill Education* • Publication date: *August 19, 2019* • Binding: *Paperback* • Estimated price: *$11.86*

Coding Games in Scratch: A Step-by-Step Visual Guide to Building Your Own Computer Games [U8] • By *Jon Woodcock* • ISBN: *1465477330* • Publisher: *DK Children* • Publication date: *August 6, 2019* • Binding: *Paperback* • Estimated price: *$13.99*

The Art of Game Design: A Book of Lenses, Third Edition [U9] • By *Jesse Schell* • ISBN: *1138632058* • Publisher: *A K Peters/CRC Press* • Publication date: *August 28, 2019* • Binding: *Paperback* • Estimated price: *$61.12*

DevOps For Dummies [U10] • By *Emily Freeman* • ISBN: *1119552222* • Publisher: *For Dummies* • Publication date: *August 20, 2019* • Binding: *Paperback* • Estimated price: *$19.41*

Help Your Kids with Computer Coding: A Unique Step-by-Step Visual Guide, from Binary Code to Building Games [U11] • By *DK* • ISBN: *1465477322* • Publisher: *DK* • Publication date: *August 6, 2019* • Binding: *Paperback* • Estimated price: *$12.80*

The Secret Life of Programs: Understand Computers -- Craft Better Code [U12] • By *Jonathan E. Steinhart* • ISBN: *1593279701* • Publisher: *No Starch Press* • Publication date: *August 6, 2019* • Binding: *Paperback* • Estimated price: *$29.58*

Code This!: Puzzles, Games, Challenges, and Computer Coding Concepts for the Problem Solver in You [U13] • By *Jennifer Szymanski* • ISBN: *1426334435* • Publisher: *National Geographic Children's Books* • Publication date: *August 27, 2019* • Binding: *Paperback* • Estimated price: *$11.71*

Your Linux Toolbox [U14] • By *Julia Evans* • ISBN: *1593279779* • Publisher: *No Starch Press* • Publication date: *August 27, 2019* • Binding: *Paperback* • Estimated price: *$16.95*

Essential TypeScript: From Beginner to Pro [U15] • By *Adam Freeman* • ISBN: *148424978X* • Publisher: *Apress* • Publication date: *August 15, 2019* • Binding: *Paperback* • Estimated price: *$32.10*

Java in easy steps [U16] • By *Mike McGrath* • ISBN: *1840788739* • Publisher: *In Easy Steps Limited* • Publication date: *August 25, 2019* • Binding: *Paperback* • Estimated price: *$10.35*

Mastering Visual Studio 2019: Become proficient in .NET Framework and .NET Core by using advanced coding techniques in Visual Studio, 2nd Edition [U17] • By *Kunal Chowdhury* • ISBN: *1789530091* • Publisher: *Packt Publishing* • Publication date: *August 9, 2019* • Binding: *Paperback* • Estimated price: *$34.99*

Mastering GitLab 12: Implement DevOps culture and repository management solutions [U18] • By *Joost Evertse* • ISBN: *1789531284* • Publisher: *Packt Publishing* • Publication date: *August 2, 2019* • Binding: *Paperback* • Estimated price: *$44.99*

Artificial Intelligence Basics: A Non-Technical Introduction [U19] • By *Tom Taulli* • ISBN: *1484250273* • Publisher: *Apress* • Publication date: *August 2, 2019* • Binding: *Paperback* • Estimated price: *$21.66*

Beginning Oracle SQL for Oracle Database 18c: From Novice to Professional [U20] • By *Ben Brumm* • ISBN: *148424429X* • Publisher: *Apress* • Publication date: *August 6, 2019* • Binding: *Paperback* • Estimated price: *$27.32*

Super Scratch Programming Adventure! [U21] • By *The LEAD Project* • ISBN: *1718500122* • Publisher: *No Starch Press* • Publication date: *August 27, 2019* • Binding: *Paperback* • Estimated price: *$13.89*

Getting Started with Web Components: Build modular and reusable components using HTML, CSS and JavaScript [U22] • By *Prateek Jadhwani* • ISBN: *1838649239* • Publisher: *Packt Publishing* • Publication date: *August 9, 2019* • Binding: *Paperback* • Estimated price: *$19.99*

Hands-On Artificial Intelligence for Cybersecurity: Implement smart AI systems for preventing cyber attacks and detecting threats and network anomalies [U23] • By *Alessandro Parisi* • ISBN: *1789804027* • Publisher: *Packt Publishing* • Publication date: *August 2, 2019* • Binding: *Paperback* • Estimated price: *$44.99*

Introducing Delphi ORM: Object Relational Mapping Using TMS Aurelius [U24] • By *John Kouraklis* • ISBN: *1484250125* • Publisher: *Apress* • Publication date: *August 2, 2019* • Binding: *Paperback* • Estimated price: *$22.23*

Surviving the Whiteboard Interview: A Developer's Guide to Using Soft Skills to Get Hired [U25] • By *William Gant* • ISBN: *1484250060* • Publisher: *Apress* • Publication date: *August 2, 2019* • Binding: *Paperback* • Estimated price: *$21.26*

Introducing Markdown and Pandoc: Using Markup Language and Document Converter [U26] • By *Thomas Mailund* • ISBN: *1484251482* • Publisher: *Apress* • Publication date: *August 20, 2019* • Binding: *Paperback* • Estimated price: *$25.95*

Machine Learning with Python for Everyone [U27] • By *Mark Fenner* • ISBN: *0134845625* • Publisher: *Addison-Wesley Professional* • Publication date: *August 26, 2019* • Binding: *Paperback* • Estimated price: *$34.25*

What's New in TensorFlow 2.0: Use the new and improved features of TensorFlow to enhance machine learning and deep learning [U28] • By *Ajay Baranwal, Alizishaan Khatri, Tanish Baranwal* • ISBN: *1838823859* • Publisher: *Packt Publishing* • Publication date: *August 12, 2019* • Binding: *Paperback* • Estimated price: *$24.99*

A Concise Introduction to Machine Learning [U29] • By *A.C. Faul* • ISBN: *0815384106* • Publisher: *Chapman and Hall/CRC* • Publication date: *August 14, 2019* • Binding: *Paperback* • Estimated price: *$48.07*

The Hardware Hacker: Adventures in Making and Breaking Hardware [U30] • By *Andrew Bunnie Huang* • ISBN: *1593279787* • Publisher: *No Starch Press* • Publication date: *August 27, 2019* • Binding: *Paperback* • Estimated price: *$12.76*

R Data Science Quick Reference: A Pocket Guide to APIs, Libraries, and Packages [U31] • By *Thomas Mailund* • ISBN: *1484248937* • Publisher: *Apress* • Publication date: *August 8, 2019* • Binding: *Paperback* • Estimated price: *$22.09*

The Sciences of the Artificial [U32] • By *Herbert A. Simon* • ISBN: *0262537532* • Publisher: *The MIT Press* • Publication date: *August 13, 2019* • Binding: *Paperback* • Estimated price: *$32.59*

SQL for Data Analytics: Perform fast and efficient data analysis with the power of SQL [U33] • By *Upom Malik, Matt Goldwasser, Benjamin Johnston* • ISBN: *1789807352* • Publisher: *Packt Publishing* • Publication date: *August 23, 2019* • Binding: *Paperback* • Estimated price: *$29.99*

About the Author

A new edition of Antonio Cangiano's excellent Technical Blogging [U34] is now available! You can also subscribe to Antonio's reports on new books in technology and other fields here [U35].

From The Pragmatic Bookshelf

Who's Where When

What's Up with Pragmatic authors.

The Pragmatic Bookshelf is the publishing imprint of The Pragmatic Programmers, LLC. *PragPub* is associated with the company, which Andy Hunt and Dave Thomas founded in the early part of this century with a simple goal: "To improve the lives of developers. We create timely, practical books, audio books, and videos on classic and cutting-edge topics to help you learn and practice your craft."

Author Calendar

2019-09-13 **Organizer**
Alex Miller (author of Clojure Applied [U36] and Programming Clojure, Third Edition [U37])
Strange Loop 2019, St. Louis [U38]

2019-09-20 **Idempotence: Build Mission-Critical Systems *and* Keep Your Job Level up your system design skills and make your users' lives better with this one weird trick. Have you ever seen a submit button get disabled after clicking it? Have you ever seen th**
Ethan Garofolo (author of Practical Microservices [U39])
UtahJS Conference

2019-09-26 **Keynote**
Alex Miller
ClojuTRE 2019, Helsinki [U40]

2019-09-30 **Keynote - TBD**
Diana Larsen (author of Agile Retrospectives [U41] , Liftoff, Second Edition [U42] , and Liftoff, Second Edition [U43])
XA/experienceAgile, Lisbon, Portugal [U44]

2019-10-14 **A Scrum Course — the official two-day course based on "A Scrum Book." Taught by Cesário Ramos and James Coplien.**
James O. Coplien
Amsterdam, Netherlands [U45]

2019-10-20 **Intro to Test-Driven Development**
Jon Reid (author of iOS Unit Testing by Example [U46])
Silicon Valley Code Camp Agile Track, San Jose, California, USA [U47]

2019-10-25 **"A Scrum Course" — the official two-day course based on "A Scrum Book." Taught by James Coplien and Karel Smutný**
James O. Coplien
Prague, Czech [U48]

2020-01-31 **"A Scrum Course" — the official two-day course based on "A Scrum Book." Taught by James Coplien and Cesário Ramos.**
James O. Coplien
Vienna, Austria [U49]

Solution to Puzzle

Vaping Venture — Solved

Give up? Here's the solution.

Stop!

Spoiler alert!

Do not read on unless you have given up on the puzzle that appears earlier in this issue.

We are about to reveal the solution!

You're still reading.

OK then, here's the solution to this month's puzzle:

T	I	M	E	S	O	A	R	Z
Z	R	S	I	T	A	O	E	M
E	A	O	M	Z	R	S	T	I
R	O	Z	A	E	S	M	I	T
S	M	E	R	I	T	Z	A	O
A	T	I	O	M	Z	E	S	R
O	Z	R	S	A	I	T	M	E
I	E	A	T	O	M	R	Z	S
M	S	T	Z	R	E	I	O	A

The answer to the anagram part of the puzzle is ATOMIZERS. An atomizer is the part of a vape pen that produces a fine spray, based on technology invented by Thomas DeVilbliss and exploiting the Venturi effect.

The BoB Page

Get the 2018 Back Issue Bundle

The Back Channel

At The Prose Garden, we plant, water, fertilize, prune, and arrange your prose (and ours) into what we hope are beautiful bouquets of insight and knowledge. If you have a budding interest in sharing some technical insight of your own, please contact us at mike@swaine.com [U1] or nancy@swaine.com [U2]. We are always on the lookout for game-changing, enlightening, pragmatic articles and our audience is, too. In addition to publishing our monthly magazine, we also edit *and narrate* books, blogs, and articles.

Functional Programming: A PragPub Anthology

If you like *PragPub*, I bet you'll like *Functional Programming: A PragPub Anthology* [U3]. We scrutinized all the articles on functional programming from the past eight years of *PragPub* and put together a book that looks at functional programming from the perspective of five languages: Clojure, Elixir, Haskell, Scala, and Swift. You'll explore functional thinking and functional style and idioms across languages. Led by expert guides, you'll discover the distinct strengths and approaches of each language and learn which best suits your needs.

"This book is an amazing buffet of programming delicacies, all arranged around the vital and compelling theme of functional programming. These authors are great teachers. With so many expert voices presenting different aspects of the topic, this book is like *Beautiful Code* for the functional-curious." — Ian Dees

Order it from The Pragmatic Bookshelf [U4].

PragPub Back Issue Bundles

The 2018 Back Issue bundle is now available. All twelve issues from 2018, in pdf, epub, and mobi formats, just twelve dollars.

Get all 12 issues of a year of *PragPub* in all three formats — pdf, epub, and mobi — plus a downloadable index to the articles, in the Back Issue bundles. Twelve bucks for a full year of *PragPub* (2013, 2014, 2015, 2016, 2017, or 2018). And of course you can purchase individual back issues. All at The Prose Garden [U5].

For Kids

Mike recently co-authored a short biography of Steve Wozniak for kids. You can buy one for the kid in your life here [U6]. Meanwhile, Nancy (under the pseudonym Summer Jo Swaine) has been recording audiobooks for kids. You can sample her work here [U7].