The Pragmatic Bookshelf

PragPub

The Second Iteration

IN THIS ISSUE

* TENDING YOUR CODEBASE
* Jim and Clarisse Bonang on teaching kids to code
* R. Michael Rogers on mutation testing
* Rachel Davies on making time to tend code
* The objc.io guys on functional programming in Swift
* Plus Rothman and Lester on learning your strengths, Marcus Blankenship on letting go, Antonio Cangiano on tech books, our monthly pub quiz...
* ...and John Shade on the Internet of Things.

Issue #73
July 2015

Contents

FEATURES

DEPARTMENTS

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

ISSN: 1948-3562

On Tap

Tending Your Codebase

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], epub[U3]. If you subscribed through an app and want to also get the monthly file download links, just ask[U4].

Sharing Your Experience

This month we have the first installment of a series on teaching kids to code, written by Jim Bonang with an assist from Clarisse Bonang, an actual kid. It's really filled with insight and case-study experience, and if you've ever taught or wanted to teach or imagined yourself teaching someone of the kid persuasion how to code, you'll get a lot out of it. Jim found himself thrown into an intense teaching situation, and drew inspiration and guidance from a variety of sources, including a special issue of *PragPub* on teaching kids to code.

You can download that special issue for free right here[U5]. And here's what's in it:

- "The Hour of Code" by David Bock. David reflects on a week teaching kids to code, and asks why we can't make every week Computer Science Education Week.
- "Giving Back" by David Bock. Teaching kids to code is one of those powerful levers by which you can move the world. David shares his experience and invites you to join in.
- "The Anti-Cosby Approach to Teaching Kids Programming" by Chris Strom. Cliff Huxtable said you have to struggle through the fundamentals before you get to the fun stuff. Nonsense, says Chris. Start with dessert and save the green beans for later.
- "Constraints and Freedom" by Jimmy Thrasher. Freedom blossoms under constraints. So does that mean you should give your kid an obsolete computer to learn on? Maybe.
- "Lego League: Lessons Learned" by Seb Rose. Choose the right sort of problem, focus on strategies for solving the problem, and the code will come easily.
- "Coding Unplugged" by Fahmida Y. Rashid. When you're first learning to code, a programming language can just get in the way.
- "The Selfish Teacher" by Michael Swaine. How teaching a kid to code could be a wonderful thing to do — for yourself.

But back to *this* issue. We do have a few more goodies, such as these:

R. Michael Rogers shares his experience in learning about mutation testing, and why he has become an evangelist for this testing tool. If you've ever had the feeling that you're refactoring without a safety net, you need to know about mutation testing.

Rachel Davies shines light on a perennial problem in software development: how do you explain to management the importance of those hours you spend — or need to spend, anyway — in activities that don't move the needle in any way they can measure? How do you justify taking time away from adding features of upping the count of lines of committed code, to spend it on cleaning up the code you have, tending the garden that is your codebase? Is there an answer? Rachel says yes, but her answer may surprise you.

What else? There's the next installment in our series designed to help you get a grip on functional programming in Swift. Rothman and Lester have been helping developers with their careers for years. This month's lesson: understand what sets you apart from the crowd. Marcus Blankenship shares the one thing you have to learn when you make the leap from lead programmer to owning your own agency. John Shade opines on the Internet of Things. And there's more, including a puzzle and Antonio Cangiano's list of new tech books. I hope you enjoy the issue.

Who Did What

editing and production: Michael Swaine

editing and research: Nancy Groth

customer support, subscriptions, submissions: webmaster@swaine.com

staff curmudgeon: John Shade (john.shade@swaine.com)

Swaine's World

Belly Up to the Bar

by Michael Swaine

"It never rains in a pub." — Traditional Irish saying

Here are a few choice bits to nosh on. Tweets. A bill of fairly interesting sites. Our Pub Quiz. Or sudoku/anagram mashup, really. A List. That sort of thing. Cheers.

Time and Temperature

- *I think my iPhone works slower in the heat… could be my imagination… maybe my brain works slower in the heat!* — @ultrasaurus
- *In four hours there will be only three more hours until I can say there is only one more hour until I can go home for the weekend.* — @estellevw
- *Can't hack Standard Time. I'm permanently on DST -- Diet Starts Tomorrow.* — @pragpub
- *The cycle continues as it has been for time immemorial: tech wisdom being handed down from a younger generation to the elder.* — @jamesiry

Pub Crawl

Your one-stop celebtech shopping list.

- Dr. Dre's headphones [U1]
- Neil Young's music player [U2]
- Neil Young's model train controller [U3]
- Hedy Lamarr's secret communication system [U4]
- James Cameron's submarine [U5]
- Steve McQueen's bucket seat [U6]
- Ashton Kutcher's venture capital company [U7]
- Bono's venture capital company [U8]
- Penn Jillette's hydro-therapeutic stimulator [U9]
- Will.i.am's future of Fashionology [U10]

Crash Reports

- *That sinking feeling when you make your n-th (or was it the (n-1)-th, or maybe (n+1)-th) off by one error. #Hackile* — @headinthebox
- *Cracked 7 ribs, but… Weir suggests I didn't see "ascending rivers of light," 'cause "you weren't dead enough." Always this criticism…* — @JPBarlow
- *The sun has NOT exploded. Repeat, the sun has NOT exploded* [U11] *Or maybe it has. We're always a few minutes behind.* — @jamesiry

- *Never work with animals, small children, or live demos of software.* — @lxt

The Pub Quiz

Here's this month's brain-teaser.

		H			B		D	S
A	D							R
				I			Y	
	R				D	A		
B		Y				H		
		S	A	Y	I			
R					T			
T						I		

It's a sudoku, but with letters. Using only the nine letters that appear in the grid, fill in the empty cells so that every row, every column, and each of the nine 3x3 boxes contains all nine letters.

And it's also an anagram. Properly arranged, the nine letters in the puzzle will spell something that certain programmers are celebrating this month.

Solution further on in the issue; don't peek!

Advice

- *Telling a story, painting an exciting vision of the future, showing what you can do, works better than telling people they are wrong.* — @storming
- *Time-saving app idea: Every so many days, tweets "apostrophes don't make words plural."* — @tottinge
- *In retrospect, I'm thinking Smalltalk was a mistake.* — @marick
- *Your password must have at least 8 characters and if one of them has a gun then they must use it to shoot another by the 2nd or 3rd chapter.* — @jamesiry
- *There will always be room in the industry for the folks who don't want to talk or collaborate on code. But the mainstream is moving on.* — @sarahmei
- *If you ever find yourself Google image searching "Girls with computers" make sure to specify "with clothes on" as it can be confusing.* — @SaraJChipps
- *Observation: exercising your powers to their full extent is a good way to get fired.* — @lxt

The List

- Scott Adams

- Marc Andreessen
- Julian Assange
- Dan Bricklin
- Mark Cuban
- Benedict Cumberbatch
- Donna Dubinsky
- Chris Pirillo
- Sir Clive Sinclair
- Nicola Tesla

Observations

- *What I like about C++ is that it doesn't even try to pretend it cares about you.* — @matthew_d_green
- *It's crazy that once personal video recorders became ubiquitous UFOs stopped visiting Earth and cops started brutalizing people all the time.* — @stephenjudkins
- *it's hard letting a domain name expire. it's like saying "Hey dream, never mind… something else came up."* — @bokmann
- *Remember the first time you discovered View Source and it felt like you hit the jackpot but really your life was ruined forever after?* — @brittanystoroz
- *Nice. MT @sly_wit Best thing I learned last week is that the French term for "mansplain" is "mecspliquer" #tropbeau* — @inevernu
- *Canadian Uber does not offer a "dogsled" option.* — @rit

The Punters

Here's the list of folks I followed this month: Sarah Allen, John Perry Barlow, Bokmann, .DS_Storoz, Matthew Green, James Iry, Stephen Judkins, Brendan McAdams, Brian Marick, Sarah Mei, Erik Meijer, Tim Ottinger, SaraJo, Stormy, your humble editor, Patrick Tanguay, Laura Thomson, and Estelle Weyl. But fair's fair. You can follow me at www.twitter.com/pragpub [U12]. Or visit my blog at swaine.com [U13].

Rothman and Lester

Learn Your Strengths

by Johanna Rothman and Andy Lester

Managing your career is a job in itself. Fortunately, Johanna and Andy have seen all the career mistakes people make and can steer you past the hidden rocks. This month's lesson: understand what sets you apart from the crowd. What makes you you is your edge.

Andy: What strengths and knowledge make you the worker that you are? What makes you different from everyone else in the job market? What do you bring to your company that nobody else does? And, can you explain it clearly? It might make a difference in finding a job, or keeping the one you have.

We human workers are not interchangeable parts, but we don't often think about what makes us unique. Each of us brings different talents to a business. If we were all the same, companies could just hire the cheapest. Sometimes we need to be able to tell others what makes us different, in a very specific and unambiguous way.

Johanna: As you prepare for your job search, inventory what makes you unique. We all know that few job descriptions actually ask for the human part of you. But, that human part of you is what will land you the job.

You might use a career timeline [U1] to understand where you were successful and what made you happy. One way to do that is to ask yourself, "Where was I successful? Why was I successful there?"

Once you know what makes you unique, you can look for companies that will help you build your strengths.

Andy: Start by looking at your skills as described on your résumé. Maybe you'll see Oracle and Ruby. Those are high level, and plenty of people know Oracle and Ruby, so dig deeper. What do you know about Oracle? Maybe you've become expert in tuning databases, and you understand when and when not to use optimizer hints in your queries. What about your Ruby skills? What have you done that not everyone else has? Have you written code and packaged it in a gem? Have you released it on GitHub?

These sorts of things are part of what you do, and you might not think much of them, but they are skills and experiences that not everyone has. To you, query optimization is just part of day-to-day life, but it might not be so to others. Are you the person in the department that people turn to for help when their queries are slow? Do you find yourself following the "sql-optimization" tag on StackOverflow and helping out newbies? Then you may have found an area of expertise that sets you apart from others, and that might just get you your next job.

Johanna: Andy's talked about some technical skills. Notice what he said about following a tag and helping newbies. Those are your qualities, preferences, and non-technical skills [U2].

For example, I have a bias towards people who have released something — preferably more than one thing. That's me. If you work in an academic environment, maybe that's not important to you or is not one of your strengths.

I want to see how you collaborate, how you treat people, what you do when no one is looking, all those kinds of things. Do you like to experiment with designs and see which one might be best before deciding on the Way It Will Be? You can be yourself, and you will see who values that.

If you are considering a "what are your strengths" assessment, be my guest. I have not found them to be that useful once you've been working for a while. I definitely do not find personality tests/assessments useful.

We are talking about your accomplishments, not your tendencies here.

Once you know what you like to accomplish, you can filter the open job descriptions you want to answer and decide how to tell the stories of your career.

Andy: I think both skills and accomplishments are important. Your skills lead to accomplishments, but some of your accomplishments might relate to skills that no longer matter. Your experiences are as much a part of who you are as a worker.

It's also important to think of both tech and soft skills when you think about your strengths. One doesn't trump the other, certainly in our tech industry. As much as we might think we're all techies, it's really more of a 50/50 split between hard and soft skills. You might be able to write code and deploy servers, but you need to be able to work with others to make these things happen.

When you consider soft skills, look at everything you do that involves another person, whether it's in person or remotely, whether with a co-worker or someone online. Have you run a software project of any type? It doesn't have to be a software project at work. If you've helped run an open source software project, that can be a distinguishing experience as well. Again, you might not think it's that big a deal, but to a potential employer, it's gold.

Johanna: I like to think of collaboration, problem-solving skills at minimum. I also like to know whether people can focus on one thing at a time, but many organizations make that impossible. Have you ever helped someone who was new in the organization? I bet you coached that person first, provided feedback, and maybe moved into more of a mentor role later. If you can ask for help and be adaptable, those are great skills to discuss, in the context of your technical skills.

Start with your most recent project or assignment at work. What can you say about it, to showcase your technical skills or non-technical skills?

Andy:: Here's a list of questions to get you thinking about what makes you different:

- What are you known for at work? For what kinds of problems are you the go-to person?
- What do people ask you for help with?
- What functions do you serve at work that nobody else can?
- What thorny problems have you solved lately?
- What recent project was a great success? What did you contribute to that success?

- What code/tests/requirements/whatever do you love working on the most?
- What was the last "a-ha!" moment you had?
- What did your boss last compliment you on?

Johanna: Start here and add whatever else you see helps you show what makes you a strong employee. Now you have data that you can use when you edit your résumé or go on interviews.

About the Authors

Johanna Rothman helps leaders solve problems and seize opportunities. She consults, speaks, and writes on managing high-technology product development. She enables managers, teams, and organizations to become more effective by applying her pragmatic approaches to the issues of project management, risk management, and people management. She writes the *Pragmatic Manager* email newsletter and two blogs on www.jrothman.com [U3].

Andy Lester has developed software for more than twenty years in the business world and on the Web in the open source community. Years of sifting through résumés, interviewing unprepared candidates, and even some unwise career choices of his own spurred him to write his nontraditional book Land The Tech Job You Love [U4] on the new guidelines for tech job hunting. Andy is an active member of the open source community, and lives in the Chicago area. He blogs at petdance.com [U5], tweets at @petdance, and can be reached by email at andy@petdance.com.

External resources referenced in this article:

[U1] https://pragprog.com/book/jrsearch/manage-your-job-search

[U2] https://pragprog.com/book/jrgeeks/hiring-geeks-that-fit

[U3] http://www.jrothman.com

[U4] http://www.pragprog.com/refer/pragpub51/book/algh/land-the-tech-job-you-love

[U5] http://petdance.com

New Manager's Playbook

Letting Go

by Marcus Blankenship

You're a developer, but you're at a point in your career where you find yourself managing others. Marcus shares tips on how to be as good at managing as you are at your "real" job.

This month he tells you the one thing you have to learn when you make the leap from lead programmer to owning your own agency.

Making the leap from lead programmer to owning your own agency is exactly what you want, right?

Be careful what you wish for. The transition from full-time coding to shouldering a company can be filled with all sorts of gut-wrenching moments and doubts.

Get your feet on solid ground with some solid advice about what to expect and how you can move into confidence with your new responsibilities.

Fantasy vs. Reality: When Running a Dev Shop Isn't What You Imagined It Would Be

Congratulations! You're on your own, calling all of the shots!

In words of Dr. Phil, *How's that working for you?*

Or maybe you're still working for someone else, but looking forward to the day when you finally can do things *your* way.

Perhaps you're a valued programmer who's worked his way up the ladder, and you have your eye set on doing more than pounding out code all day, as much as you love that job.

When I imagined my transition from lead programmer to head of my own agency, I had all sorts of fantasies about how my daily work would change, and how I would suddenly be in command of a loyal, efficient group, conquering deadlines and intricate projects.

Man, was I wrong.

Fantasy

With very little hands-on management experience, I developed the most lovely misconceptions about becoming a manager. Do these sound familiar?

- My authority would be immediately recognized and embraced by everyone.
- I'd still be a programmer and just add in *a few* additional responsibilities.
- Owning my own company equaled raking in profits.
- I'd have the pick of the best jobs coming through the project pipeline.
- I would have *so much more* control of my time.
- I'd be the in the captain's chair when it came to decision-making.
- My team would boldly (and automatically!) follow me into battle, ready to conquer the next project.

A management position often sneaks up on you when you least expect it. One day, you're working solo or with a partner to build a tiny consultancy, and when you pick your head up a couple of years later, you've taken on two or three additional staff who are looking to you for direction.

My advice? Be smart. Don't be ambushed by your own success.

Promotion Reality

When I finally started my own business, fantasy gave way to reality so fast it made my head spin. And trust me, there was no fanfare and no applause.

Just my name on a website and business cards and a whole new workload that I wasn't entirely equipped to handle at that point.

Here's what my first few months as a manager were really like:

- I had lots of freedom — to make lots of new, very public mistakes.
- I created confusion when I committed to delivery dates without consulting my team first.
- With each new project that crossed my desk, I was convinced that *this time, I was going to do everything right*.
- I allowed myself to be interrupted for any reason at any time in the name of being available for the team.
- I refused to delegate [UI], and I took on coding repair jobs that kept me at work nights and weekends. Let's just say I missed a lot of family time.
- I was overwhelmed. All. The. Time.

After several rounds of workload adjustments and still feeling stressed, I got to the core of my anxiety. *I kept trying to be a manager without letting go of my programmer identity*.

Like most coders, I prided myself on being an uber-brain who could design my team out of any problem with finesse. However, after becoming a manager, I saw my coding skills slipping away at the same time as I was feeling completely inept about my new responsibilities.

It was terrifying.

I struggled along for quite a while, finding few role models or advice other than, "You'll find your own style."

Real Solutions

I finally got to the point where I couldn't pedal any faster. I was burned out and on the edge of collapse.

So I designed a radical experiment: I decided to drop all of my coding responsibilities and focus on managing production. After three months, if my team wasn't more productive, then I would go back to part-time coding.

I was terrified to start this counterintuitive approach. What if I failed? What if company utilization fell even further behind and my confidence took another hit? If this bombed, what else could I try to solve my overload problems?

In spite of my fears, I moved ahead.

It Got Better

At the end of three months, the results were clear. Less coding meant a better team.

My team had more on-time shipments. Clients were happier. Morale was up. And incredibly, production was the same or better because I wasn't doing a half-assed job of delegation.

Sanity for the win!

Save Yourself: Survival for a New Manager

Even with all of the best advice and intentions, you will probably reach that point where you're drowning in projects and trying to hold on to your coding skills.

When you get to that place, do yourself a favor. Don't cling tighter to the idea that you and your fabulous geekified brain can do it all.

Let go.

Accept the fact that your coding skills will erode, but as they diminish (slightly), you will gain perspective. After a few months of seeing production from a new perspective, you will have amazing moments facilitating some serious wins.

Will you lose contact with coding forever? Not a chance.

Software programming is full of circumstances where glitches, bugs, or an impending deadline require all hands on deck. And you can hop into pair programming with your employees if they need a little guidance.

It is completely normal to feel shaky and lost as you make your way through the first few months walking in management shoes. But when you find yourself taking on more work to prove your supercoder status, step away from the computer … and think about what you can turn over to your team.

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 [U2].

External resources referenced in this article:

[U1] http://www.slideshare.net/marcusblankenship/slideshare-novmbdelegation-manual

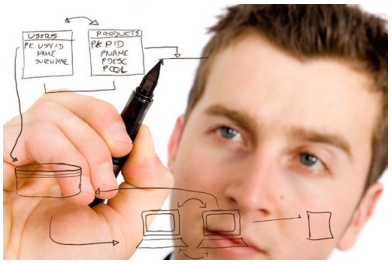
[U2] http://marcusblankenship.com

Making Computer Science Insanely Great

Part 1: Better to be a Pirate than Join the Navy

by Jim Bonang and Clarisse Bonang

Jim teaches kids to code the *PragPub* way in the most unlikely of circumstances with help from his daughter Clarisse, and explains why teaching would be great for you too.



I wanted to introduce computer science at my daughter's high school.

While her school provides an excellent computer laboratory equipped with iMacs, like most U.S. schools it offers no computer science courses. If "software is eating the world" [U1] as Marc Andreessen contends, why wouldn't my daughter's high school teach computer science? Indeed, why doesn't every U.S. school offer computer science classes? Hadi Partovi, the founder of Code.org [U2], a Seattle-based nonprofit organization that makes available ready-made lessons for kindergartners, grade schoolers, and high schoolers, asked this same question [U3](*Seattle Times*, March 4, 2015). He believes that every student in every school deserves the opportunity to learn computer science, as he explains in his TED talk [U4]. I agreed, and set my plan in motion. I'd start by teaching a single introductory class, much as David Bock described in "Teaching Kids to Code: Reflections on an Hour of Code" [U5] (*PragPub*, January, 2014).

But fate intervened, in the form of a letter from the U.S. Navy (actually, an encrypted email with an attached PDF file) informing me in the subtle, understated, ALL CAPS way characteristic of armed forces message traffic, that my life was about to change dramatically:

MOBILIZATION ORDER:

OFFICIAL RECALL TO ACTIVE DUTY …

YOU ARE HEREBY ORDERED TO REPORT FOR ACTIVE DUTY …

I didn't take Steve Jobs's figurative advice on preferring piracy to the Navy (Levy, Stephen. *Insanely Great: The Life and Times of Macintosh, The Computer that Changed Everything* [U6]), and served as a reserve officer. It was a rare case where I ignored advice from Steve. I was off to points unknown, my opportunity to teach programming and computer science indefinitely postponed.

Philanthropy?

But wait. Why the interest in teaching young people computer science anyway? Why teach kids to code? Of course, you'll benefit your students, perhaps greatly, and these kids may even be your own. But still, why should you, a software professional, go to the very considerable trouble of teaching programming? What might you gain?

First of all, you may enjoy it. A tough job to be sure, but fun awaits you. (Remember, programming is tough, and fun too.) You can show off (yet preserve plausible deniability). On rare occasions, your students may actually

appreciate your efforts — and the intense afterglow of that will last until the next rare occasion.

Then, too, the programming field walls you off from friends and family. In "The Selfish Teacher: Teach a Kid to Code — for Your Own Good" (*PragPub*, April 2014), Michael Swaine maintains that "If children are in your life, you need to teach them to code." To break down those walls "You need to share what you do. And you need to do it for yourself. It will make you a happier and richer person."

You'll also accrue practical benefits too. Teaching will perfect your presentation skills on a demanding yet forgiving audience in a benign environment — rather than in front of your colleagues, or your boss. Sooner or later, you'll need to teach new techniques to your development team; you'll also need to present your ideas to management. To persuade them, they first need to understand; you need to teach them. You may be a new engineer, fresh out of school, with fresh new ideas. You'll need to teach the old dogs your new tricks. You may need to explain your product idea to a patent attorney, or at a technical conference. You may even need to persuade venture capitalists of the viability of your new startup idea. Your fate rides as much on how well you present the idea as on your idea's intrinsic viability. If you've polished your skills by teaching, your ideas will get due consideration.

To be sure, teaching kids to code will help your career. But it will change you at a more fundamental level: teaching will make you better. A better engineer. A better leader. A better presenter. A better person.

I was convinced that teaching kids to code was something I wanted to do. But I wasn't going to be allowed to do it, at least not for a while. And then fate intervened again.

The Opportunity

The heat shimmer off the tarmac greeted me as I descended from the transport aircraft to begin the next leg of my deployment at an overseas naval base. The base included a small high school serving the children of U.S. service personnel run by the U.S. Department of Defense Education Activity (DoDEA) [U7]. Despite its diminutive size, with fewer than 100 students, the school's dedicated teachers provided a very rigorous curricula including numerous advanced placement courses in science and mathematics. The school often used volunteer service personnel, of which there is an abundance, with specialized skills to assist with extra curricula events and to tutor students. Science, Technology, Engineering, and Math (STEM) activities night was coming up and I submitted my application.

There's no question about it: "Irony is Fate's most common figure of speech" [U8] (*Shibumi* by Trevanian, Ballentine Books, 1979. ISBN 0-345-31180-9). An email message notified me I was selected for the STEM computer science activity and was to introduce students to programming.

The lesson had to be exciting. It had to be interesting. It had to teach an immediately useful skill and give the students a sense of accomplishment. No previous programming experience needed — at all. And it had to be under

thirty minutes. Thirty minutes! Meeting these requirements — especially the need to make it exciting and interesting — would be tough; was it even feasible?

Breaking It Down

Tough? Understatement. In graduate school, I taught a programming class for electrical engineers; they were already interested in the subject and didn't need encouragement. Teaching an introductory course for high school students was another matter entirely. I needed help and turned to the recent series of articles on "Teaching Kids to Code" in *PragPub* for firsthand, expert advice on crafting an introductory lesson for kids. I also solicited advice from an expert in what teens find interesting about technology — a real live teenager no less. Although hands-on coding may constitute the bulk of my lesson, I realized that the hardest part would be my presentation. To ensure mine was engaging, I secured advice from a legendary presenter, a master at making technology interesting and exciting. (You may have seen him in action.) I also obtained help from an expert in designing presentation materials. Her guidance left me astounded and I never looked at a presentation the same way again. I've distilled this guidance, and my own hard-won lessons, down to three essential skill sets you'll need to succeed at teaching an introductory coding class:

1. How to craft your lesson
2. How to choreograph your presentation
3. How to design your presentation materials

I want to share what I learned over the next three issues of *PragPub*. First, I'll show you how to craft an introductory lesson suitable for children or teenagers at a STEM or similar event. You'll discover how to choose a topic that is interesting and that will encourage students to explore computer science, and one that's easy for students to get started with. You'll learn how to turn the lesson into play. Later, you'll learn how to choreograph an exciting presentation that will keep your audience interested. Then, you'll learn how to design clear presentation materials. From start to finish, you'll learn how to make computer science insanely great.

That's my plan. Let's see how it works out.

People Do Not Pay Attention to Boring Things

Now, it may seem counterintuitive, but when you craft your lesson for play, and the students are having fun, it can lead to a more thorough understanding of the fundamentals.

Chris Strom advises in the "The Anti-Cosby Approach to Teaching Kids Programming" (*PragPub*, May 2014) that "optimizing teaching for play rather than for fundamentals leads to more learning — and eventually to more understanding of the fundamentals." Could this be true? He goes on to observe that "none of us ever got excited about the fundamentals [of programming] and then created something amazing. So why teach kids like this?"

Long before I'd ever heard of software engineering (and very nearly before the term was coined), I went on a field trip to the California Museum of Science and Industry [U9] in Los Angeles. One exhibit in the Mathematica room [U10] captured my attention — I'd never seen anything like it. It was simple and

familiar, yet completely new and other-worldly: the Tic Tac Toe computer. It would beat me, or least tie me, every game. I was stunned; how could a machine do this? I wanted to beat the computer and began devising strategies that would allow me to win or at least always achieve a draw (I discovered a draw was the best attainable — admittedly not hard, but when you're five …). I was learning about algorithms, but there was no mention of this term in the Mathematica room; I was just playing. (Much of the Mathematica exhibit lives on at the New York Hall of Science [U11].) That recollection settled it; for my lesson I'd follow Chris Strom's approach.

Here, the high school teachers came to my aid. They suggested the computer science session have students construct customized, animated speaking characters, or avatars, using the Voki web service [U12]. Students select the facial features, hair color, hair style, and clothing worn by their avatar. They also type in text for their Voki character to speak, in any of several selectable voices. Vokis emit speech output in a variety of selectable languages too — English, Russian, Spanish, and many others, rather like Google Translate [U13]. I built my own Voki and that same spark was there — just like the Tic Tac Toe computer. Tempered by age and experience to be sure, but it was there!

To focus the exercise, the supervising teacher wisely recommended that the students have two Vokis debate whether school uniforms should be mandated. The Voki exercise demonstrated speech synthesis and waded ankle-deep into graphical programming, but I wanted to teach something more, something big, a really big idea.

There's something we do as programmers every day, something we don't notice but that is very strange to the non-programmer — outside of their experience. We type in text, and it makes the computer do something. The text has to conform to a certain format, a very precise format, a syntax from which no deviation is allowed (at least if the computer is to do something). And what the computer does is what we want it to do; the text has meaning and the semantics can be divined by the machine. The computer performs syntax analysis on the text we type and determines what it should do — performs semantic analysis. These notions are no longer obvious.

With the ubiquity of the graphical user interface, students press buttons to make computers do things. They may type in text to write papers, but to format the text, they press buttons (or select from GUI controls). They don't type in commands. They've never used MS-DOS, the windows command prompt, or the bash shell. Even a markup language is alien.

I decided to convey this notion: you can type something in a very precise way to get the computer to do exactly what you want it to do. Tokens, lexemes (lexical elements), syntax, semantics, and runtimes, all in one go, but without the terminology, just intuition, just play.

I was following Chris Strom's advice. When you teach, optimize for play, but subtly raise questions that will eventually lead students to fundamental understanding. And, no matter the assignment, no matter the prescribed lesson plan, tailor it until you think it's interesting and exciting to you. Your enthusiasm will come through as you teach it.

That sounded good, but how would I do all this in thirty minutes? And satisfy the lesson constraints? And have it ready on schedule, in time for STEM night? Oddly enough, the answer turned out to be "by applying more constraints."

Counterintuitive Constraints

"To achieve great things, two things are needed: a plan, and not quite enough time." — Leonard Bernstein

Would some creative teaching material emerge from what appeared to be very tight constraints? In "Constraints and Freedom: "29638 Bytes Free"— What a Gift!" (*PragPub*, March, 2014), Jimmy Thrasher asserts that "Constraints enable freedom and creativity." He's on to something. He focuses on the accessibility of the computing environment. Typically, the more powerful — less constrained — an environment, the more complex it is to use. More specifically, the more complex it is to *start* using. I needed the simplest programming environment possible, even though it might be constraining.

I decided the students would build a web page using just a few lines of HTML, and that they would use the ubiquitous (and, as it turned out, familiar) Notepad editor, and they would include their Voki avatars on their own web page. The two Voki characters would be positioned one atop the other; the students would then click on each Voki's play button to have it present its pro or con argument in the school uniform debate. The Voki website provides a downloadable code fragment to embed the Voki Shockwave video within a web page. Perfect!

Embedding the Voki in a web page by splicing in a code fragment had another advantage. It conveys the notion of using a function, or more specifically a segment of code. You may not understand how the code works but if you understand *what* it does and its *interface*, you can use it. A very powerful notion.

So: apply enough constraints to the environment so that students can easily get started. But constraining the programming environment to allow students to start quickly gets you only partway there. You also need to choose a project for your lesson that lets students get results quickly — results they can see.

Choose Wisely

In "Advice from Lego League: Lessons Learned Teaching Kids to Code" (*PragPub*, February, 2014), Seb Rose observes that "some approaches to teaching kids to code work better than others." He found that to keep kids interested and searching for ways to learn more you must choose a good project, one that has room to expand, and that allows students to achieve visible results quickly. I divided the hands-on work into two nearly identical major parts (one "pro" school uniform and one "con"), with each part consisting of three segments. For the first part, the students would initially create a Voki avatar and test it on-line. This first Voki would present the "pro" side of the uniform argument. The students would have something working very quickly. Next, they would create a simple HTML file and display this in a browser. Again, immediate visible results. Finally, they would insert the Voki code to display the Shockwave video into their HTML file and display this in the browser. These three segments would complete the first major part. Since time was so limited, the second part, identical aside from presenting the "con" side of the

uniform argument, could be completed outside of class. The students had all the tools needed: a web browser, Notepad, and Voki accounts. They could even use Vokis to spice up their own class presentations; the lesson had room to grow.

You'll likely have much greater latitude in your choice of projects than my circumstances permitted. For instance, I designed my lesson for beginners; your students may be further along. Fortunately, there's no shortage of great lesson ideas and one is sure to suit your audience.

Informed Sources

Sources abound that suggest topics and offer lesson plans ranging from books and magazines, to educational websites, to specialized programming environments with visual drag-and-drop block languages. The *PragPub* articles I've mentioned are a great place to start. Additionally, for younger students, consider the approach of Fahmida Rashid in her article "Coding Unplugged: A Case Study in Teaching Kids to Code" (*PragPub*, March, 2014). She uses a game where one child plays "the robot" who is "programmed" (told what do to) by another child, the "programmer"; no computer is used. She then advances the students to coding in a graphical programming language.

There are plenty of books too. Bryson Payne's *Teach Your Kids to Code: A Parent-Friendly Guide to Python Programming* [U14] (No Starch Press, 2015, ISBN 1593276141) includes intriguing lessons on topics such as decoding secret messages. (Antonio Cangiano points out in his column "Antonio on Books" in the May, 2015 issue of *PragPub* that the book is a guide for parents with programming experience, just as the title implies, and suited mainly for high school-level students due to the mathematical nature of the examples.) Consider also Chris Strom's *3D Game Programming for Kids: Create Interactive Worlds with JavaScript* [U15] (The Pragmatic Bookshelf, 2013, ISBN 978-1-93778-544-4) and Andy Hunt's *Learn to Program with Minecraft Plugins (2nd edition): Create Flaming Cows in Java Using CanaryMod* [U16] (The Pragmatic Bookshelf, 2014, ISBN 978-1-94122-294-2).

Educational websites offer great ideas too. Udacity [U17] offers free on-line courses for beginning programmers including "Introduction to HTML and CSS," "JavaScript Basics," and "Java Programming." Udacity courses use an enticing combination of graphics, videos, and hands-on exercises; all of which might stimulate ideas for your lessons. (My daughter, Clarisse, loves the Udacity Java Programming class and is working through it in her spare time.) And don't forget Code.org [U18] and Khan Academy [U19]; Khan Academy provides a series of introductory programming tutorials [U20]. If these resources don't meet your needs, consider using a ready-made educational programming environment.

Educational Programming Environments

Many programming environments for children and teens use visual, drag-and-drop block languages such as Scratch [U21], Hopscotch [U22], Alice [U23], and Lego Mindstorms NXT-G [U24]; you'll find these fascinating. Much like Windows Forms drag and drop GUI controls in the Microsoft Visual Studio toolbox, visual blocks in these environments implement language constructs such as statements, conditionals, and loops that may be dragged from a pallete,

placed into a program, and configured graphically. For instance, you might set the number of loop iterations by clicking on one of the arrows of a Numeric Up Down Control within a graphical loop block. Bill Gates illustrates loops and conditionals in a visual block language in this video [U25]. The language and environment make things easier on new programmers by preventing syntax errors. (Semantic errors remain a problem.) These visual block programming languages offer many interesting projects and lesson ideas. But remember, whatever topic you choose for your class, make sure that it's interesting to *you*.

And these languages engender much interest indeed. Each focuses on different skill levels and application areas. NXT-G differs from the rest in that it is an embedded systems programming language, used to program Lego Mindstorms robots [U26]. It's based on the National Instruments LabVIEW [U27] programming language. The name Mindstorms honors MIT computer scientist and educator Seymour Papert [U28] whose 1980 book *Mindstorms: Children, Computers, and Powerful Ideas* [U29] (2nd edition, Basic Books, 1993, ISBN 978-0465046744) proposes a computer-based learning environment.

You may recognize Papert as one of the developers of the Logo [U30] educational programming language and its well-known turtle graphics [U31], an on-screen relative cursor that produces line drawings based on programmed movement and drawing commands (turtle graphics, incidentally, appear in Payne's *Teach Your Kids to Code*). Contemporary drag-and-drop block languages build on Papert's original concepts, allowing you to create interactive games, stories, and animations. One of Papert's doctoral students, Mitchell Resnick [U32], leads the MIT Media Lab's Lifelong Kindergarten group [U33] in developing the Scratch programming environment [U34]; think of Scratch as a programmable multimedia authoring tool [U35]. My daughter enjoyed using it to create several animated stories. Dr. Resnick's superb TED Talk Let's Teach Kids to Code [U36] will provide ideas on how to incorporate Scratch into your lesson plan, and will also demonstrate how to give a very engaging presentation. For additional ideas, consider Majed Marji's *Learn to Program with Scratch: A Visual Introduction to Programming with Games, Art, Science, and Math* [U37] (No Starch Press, 2014, ISBN 1593275439). The Alice programming environment resembles Scratch and allows you to create computer animations using 3D models; my daughter found it engrossing. While Alice is more complex than Scratch, Hopscotch is a simplified version of Scratch available on the iPad and intended for younger programmers. They all offer constrained environments, will give immediate visual results, and give student projects "room to grow." You'll find all of them interesting and one will be sure to suit your audience. Of course, Vokis, HTML, and Notepad work just fine too.

Know Your Audience

While your topic has to be interesting to you (otherwise you won't be excited about it when presenting), you also need to ensure your lesson will be interesting to your audience. I decided to consult an expert on what would be interesting to my target audience — teenagers. And who better to ask than a teenager? My daughter, Clarisse, offers the following advice:

Iron Man, the Marvel Comics superhero, possesses only one superpower — his ingenious engineering skill. All of his other super abilities come from the armored

flying suit he designed. Coding ability is like having a superpower; you can create websites, games, movie effects, animations, and more. During the summer after seventh grade, my parents sent me to a computer animation class for kids at a local college; the class used a programming language called Scratch. At first, this computer language seemed strange and perplexing, but I was astounded that kids my age could create characters and bring them to life on the screen by writing programs. I was determined to develop this superpower. Soon, I created a short film with animated sea creatures swimming along to the tune of "Under the Sea" from the Disney movie The Little Mermaid. Every teenager wants a superpower.

After my summer school classes ended, I became a bit of a couch potato and would play the online Disney Fairies [U38] game Pixie Hollow [U39] (secretly) for hours. Sometimes, I would be irritated by the limited number of locations to visit in Pixie Hollow and would actually yell at the computer (which remained politely silent). I thought, "Wouldn't it be cool if I created my own world." I consulted my dad with my ever-so-brilliant idea and he introduced me to Alice. Alice is a programming environment that lets students learn fundamental coding concepts by creating 3D animated movies. It doesn't take a genius to figure it out, just curiosity, a willingness to explore, and a little imagination. Using Alice, I created a better (animated) ending to the movie Mean Girls; well, at least an ending more to my satisfaction. Teens find this delightful; they can become the director of their own movie (or their own world). Also, they can use their imagination and their skill to alter things to the way they like them — to set things right (like Iron Man does).

I never thought Alice would come in handy for school though. During my freshman year in high school, I created an animated "movie trailer" summarizing the short story "The Most Dangerous Game" [U40] using Alice. The animated summary was shown during my book report presentation in English class. It helped me earn an 'A' on the report and it showed off my programming abilities and creativity. Teenagers can't resist showing off — especially when they have a superpower.

Every teenager wants a superpower and they want to show it off — I knew just how to work that into the lesson. (You'll see how in Part III.)

Intermission

My lesson plan was now in place and I had an outline of the exercise. And, it conformed as close as I could come to the teaching advice from the *PragPub* authors. The table below summarizes the *PragPub* techniques and how I did.

Table 1: The *PragPub* Teaching Kids to Code Techniques

Technique	How I Did
Design hands-on lesson for play; the lesson should raise questions that lead to understanding the fundamentals.	Check. Build a Voki avatar — pure fun — and embed in HTML code. Illustrate interpretation of code; fun with a touch of fundamentals.
Choose a topic that is interesting to you; your enthusiasm will show and help to instill interest.	Check. Voki triggered that same spark as that old Tic Tac Toe computer — it's interesting.

Constrain the lesson so that it's easy for students to get started.	Check. Only Notepad, Voki website, and Web browser needed.
Get visible results quickly and ensure the exercise has room to grow.	Check. Visible results with the very first Voki and the exercise could be expanded outside of class.

Not bad! There were just a few problems. I'd need to provide "just-in-time" instruction: each student would need to learn the Voki website interface and then create a Voki. Then, they'd learn the rudiments of HTML and write their own web-page. All within thirty minutes. As a STEM event, the lesson would have to be inspirational and illustrate role models too: I'd have to "market" computer science. And I'd have to do that while serving in a deployed environment with very little free time. For high school students, I'd need an *insanely great* presentation. Fortunately, I had a new Apple MacBook Pro with me to help build the presentation. But how to design the presentation? How to organize it? How to make the presentation interesting and exciting. Where to find a good exemplar? Who's good at doing this? I stared at the MacBook, watching the thin blinking cursor in my notes file, thinking. It didn't take long.

We've created a special issue of PragPub containing all of our articles on teaching kids to code that are referenced in this article. You can download the issue in any or all of our three formats here: pdf [U41], mobi [U42], epub [U43].

About the Authors

Jim Bonang has spent over twenty-five years developing software for a wide range of systems, including satellite terminals, document management systems, medical devices, and many others. He's also worked on several compilers.

Clarisse Bonang just completed her sophomore year in high school, and is halfway through the Udacity introductory Java course. She programs her Lego Mindstorms robot, Probie, whenever school science projects permit his use and she created an animated book report using the Alice programming environment for her English class.

External resources referenced in this article:

[U1] http://www.wsj.com/articles/SB10001424053111903480904576512250915629460
[U2] https://code.org
[U3] http://www.seattletimes.com/opinion/why-we-need-to-teach-all-students-computer-science-skills/
[U4] https://www.youtube.com/watch?v=m-U9wzC9xLk
[U5] http://issuu.com/presspad/docs/i4319
[U6] http://www.amazon.com/Insanely-Great-Macintosh-Computer-Everything-ebook/dp/B006ZA7E6M
[U7] http://www.dodea.edu/aboutDoDEA/index.cfm
[U8] http://www.amazon.com/Shibumi-A-Novel-Trevanian/dp/1400098033
[U9] http://californiasciencecenter.org
[U10] http://www.whmsicl.cnc.net/CMSI3.html
[U11] http://nysci.org/mathematica/
[U12] http://www.voki.com
[U13] https://translate.google.com
[U14] http://www.amazon.com/Teach-Your-Kids-Code-Parent-Friendly-ebook/dp/B00WJ049XI
[U15] https://pragprog.com/book/csjava/3d-game-programming-for-kids
[U16] https://pragprog.com/book/ahmine2/learn-to-program-with-minecraft-plugins
[U17] https://www.udacity.com/courses/all

[U18] https://code.org

[U19] https://www.khanacademy.org

[U20] https://www.khanacademy.org/computing/computer-programming

[U21] https://scratch.mit.edu

[U22] https://www.gethopscotch.com

[U23] http://www.alice.org/index.php

[U24] https://en.wikipedia.org/wiki/Lego_Mindstorms_NXT

[U25] https://www.youtube.com/watch?v=m2Ux2PnJe6E

[U26] http://www.lego.com/en-us/mindstorms/?domainredir=mindstorms.lego.com

[U27] http://www.ni.com/labview/

[U28] https://en.wikipedia.org/?title=Seymour_Papert

[U29] http://www.amazon.com/Mindstorms-Children-Computers-Powerful-Ideas/dp/0465046746

[U30] https://en.wikipedia.org/wiki/Logo_(programming_language)

[U31] https://en.wikipedia.org/wiki/Turtle_graphics

[U32] http://web.media.mit.edu/~mres/

[U33] https://llk.media.mit.edu

[U34] https://en.wikipedia.org/wiki/Scratch_(programming_language)

[U35] http://web.media.mit.edu/~mres/papers/Scratch-CACM-final.pdf

[U36] http://www.ted.com/talks/mitch_resnick_let_s_teach_kids_to_code

[U37] http://www.amazon.com/Learn-Program-Scratch-Introduction-Programming/dp/1593275439

[U38] https://en.wikipedia.org/wiki/Disney_Fairies

[U39] http://fairies.disney.com/pixie-hollow-games

[U40] https://en.wikipedia.org/wiki/The_Most_Dangerous_Game

[U41] https://s3-us-west-2.amazonaws.com/pp-x1/Special.pdf

[U42] https://s3-us-west-2.amazonaws.com/pp-x1/Special.mobi

[U43] https://s3-us-west-2.amazonaws.com/pp-x1/Special.epub

Mutation Testing — Totally a Thing

Is This Line of Code Really Necessary?

by R. Michael Rogers

At first, Mike couldn't figure out why he needed mutation testing — and then he talked to someone who uses it effectively.

Mutation testing is a way to verify that your code is covered by tests.

"What, like code coverage?" you may ask. Or, you may say "you want me to test my tests? That's stupid. You're stupid!" — which are not dissimilar from my reactions to mutation testing when I first heard of it.

Or maybe you're not as vitriolic as I am. Moving on.

Let's take as an assumption that you don't want unnecessary code in your application, because every clause that isn't 100-percent necessary is a new vector for bugs, cruft, and misdirection. I say 'misdirection' because if you're trying to refactor a piece of code and you spend an hour trying to figure out why that particular clause is there and it turns out "no reason" you're going to feel a little lied to. I know I feel that way sometimes.

So mutation is a great way to test that things like this...

```
if (someVariable != null && someVariable.hasValue())
```

...are necessary. Do you actually know if you need the null check? Is that something you did reflexively? That null check could mean you're not 100 percent sure what the contract is for the someVariable variable, and might warrant looking into. You may want to clarify your intent with the Java 8 Optional <> type.

Or not! But you can't have that discussion without catching the problem first!

How It Works

The idea is pretty simple.

A mutation is a very specific yet slight alteration of application source code. You can think of them the same way you think about Software Patterns in that each type (which is concrete and well described) can be applied in many places. For example, a mutation exists that changes the return type of a function to void. You run that mutation everywhere it can be in your source code, at each point re-running your tests. If the test fails, that mutation is said to be 'killed' and your code is safe from that mutation. If the mutation makes it through your tests, it's said to have 'lived' and your code is not mutation-proof.

There exist a number of mutation testing libraries, each of which have their own sets of mutations. The grunt-mutation-testing plugin for JavaScript has a nice list[U1]of mutations applicable to JavaScript, and many mutations overlap between libraries/platforms/languages.

The output you get won't differ much from the code coverage tools you're used to. You'll get a percentile of 'killed mutations' versus 'total mutations' and it

will be a lot lower than you'd prefer. But hold on to your pants, you don't want to use the metric wrong and start adding tests willy-nilly. Keep reading!

My First Experience

Here's why I initially disliked mutation testing.

My first experience was with a .NET application written for the configuration of a medical device, so we're talking Regulatory Overhead, with I-triple-Es and FDAs and so forth. A combination of a massive, massive codebase and correspondingly huge test corpus meant that the mutation testing had to be distributed and it still took over a day.

That's a long time to wait for feedback, but if it were a valuable metric I could wait. A misconfiguration of the medical device could result in a human death. I'll take a longer feedback loop to avoid that.

But it was frustrating to me because I couldn't understand the value of the metric. It just seemed like Yet Another Measurement to abide by (although we never got so far as to add it to the process) without an explanation of how to use it properly.

My second experience was in the JavaScript space, where my team was asked to adhere to 100-percent mutation-free code. This seemed less valuable initially because human lives weren't on the line (to be trite). Also, I personally dislike JavaScript so the more I have to play in it, the less comfortable I am.

So mutation testing was a hard sell for me.

How the Community Does It

And then I reached out to the wonderful Lisa Crispin [U2]to see if I was doing it wrong (hint: I was). My initial concern was that the cost/benefit of practicing mutation testing was out of whack and we weren't getting the value we desired. Her response was pretty enlightening:

"My experience working with teams doing TDD over the past 15 years leads me to believe that in most cases, mutation testing would be expensive and not add much of a safety net. *Of course it depends on how good your programmers are at TDD*, but if they're poor at TDD then they might also be poor at implementing mutation testing." — Lisa Crispin (emphasis mine)

So if you're doing Test-Driven Development *right*, your code should be mutation-proof to begin with and your test coverage should be solid.

And from her twitter followers we heard the same thing over and over:

- *@jbrains @lisacrispin I've never seen a need for it. Maybe for embedded stuff hard to end to end test? But I have seen it as /helpful/.* — @mheusser
- *@lisacrispin Things like Jester? I never used it seriously. I do mutation testing when I don't see a test fail first.* — @jbrains (J.B. Rainsberger)
- *@lisacrispin So far, I haven't been able to come up with a good use for it though.* — @_prakash (Prakash Murthy)

… said the experts at TDD. But what about us normal humans?

- *@lisacrispin I find it most helpful when adding tests to legacy code. Which includes the times when I've gotten sloppy with TDD* — @patmaddox

- *@jbrains @lisacrispin @patmaddox agree. i mutation-test mostly manually and mostly cuz i'm irritated. same as firing up the debugger.* — @GeePawHill (Michael D. Hill)

So what I was seeing was that mutation testing bridges the gap between less-than-perfect TDD and really solid code. But how do you use it pragmatically?

- *@lisacrispin It for me is the technique closing the TDD cycle.* — @_m_b_j_ (Markus Schirp)
- *@lisacrispin Without mutation testing a TDD cycle can add more semantics to code than the new test asked for.* — @_m_b_j_ (Markus Schirp)
- *@lisacrispin It reduces the code / specification gab. Disclaimer, no silver bullet.* — @_m_b_j_ (Markus Schirp)

"I have to try this." I said to myself, stuffing my face with buttered pop-tart.

Personal (Better) Experience

I had some refactoring I was going to do on a Jenkins plugin [U3] I've been working on. It was a perfect opportunity to try to use this new tool "correctly" (for certain values of 'correctly').

I focused on PITest [U4] because that's what was encouraged by Markus Schirp (@_m_b_j_). I ran it initially and was not disheartened to see that my code was only 50 percent covered from mutation, because when I forked the repository it didn't even have a 'test' directory and I was actually proud that the coverage was so high. Go me!

PITest has a nice report that shows me which lines are not safe from mutation. I did not use that to back-cover those lines with tests, but instead kept in mind where the tests were weak and where they were strong.

The refactoring went over well enough, and then I looked into re-running the mutation tests. Fortunately PITest has a wonderful feature where it integrates with Maven's SCM plugin to only be exposed to files that have changed, so the rerunning of mutation only took a couple of minutes.

The generated report showed no change in coverage, which is exactly what I wanted to see. All the refactored code was covered by the existing tests.

This experience was pretty transformative. It showed me how to use the tool to gain confidence in my refactoring, and that led to a great deal of comfort in the output.

Conclusions

It would be so, so easy to fall into the trap of using this tool in a non-agile fashion, the way many of us have seen more traditional code coverage metrics used. It's a pretty solid metric, so mandating 100-percent mutation-proof code would be easy. But expensive.

I've seen that edict in practice and what I've found is that it moves the conversation away from a proactive discussion of testing and TDD and into a reactive mode of discussion. Instead of "did you consider this corner case" or "I think this refactoring would be more clean" the conversation turned to "this expression is not covered by tests," which is neither helpful nor cost-effective.

If you want to build a more proactive culture, then you should look into pairing with an eye for building TDD chops and applying mutation testing pragmatically. From Lisa Crispin:

"I agree with your ideas on the proactive measures. My teams have always done those things — making sure the team has plenty of time to learn and practice so they can master the practices that help build quality in." — Lisa Crispin

So, an answer is pairing [U5].

The fundamental shift for me came with the realization that a chunk of TDD as I was practicing it was unverified. In the traditional Red Green Refactor cycle, Red and Green are pretty easy. But how can you be sure you're not injecting something unnecessary during Refactor? That's a full third (or more, realistically, given the complexity of some refactorings) of your TDD cycle without a safety net!

Hence mutation testing.

If you want to learn more about mutation testing I found this article [U6]pretty engaging, with plenty of decent citations, or drop me a line, or just kinda hang out the window and scream really loudly.

Special thanks to Lisa Crispin and the Twitterblag in general for providing such impressive insights and feedback.

About the Author

Mike Rogers has worked for more than a decade in the software engineering field, making myriad mistakes as he goes. He's currently learning and working at Software Engineering Professionals [U7] in Carmel, Indiana, which is an amazing company and you should give them your money. When he's not working he spends time on his little farm with his wife Sara, his son, and his goats and horses. His hobbies include languages, tinkering with electronics, and being confidently wrong.

External resources referenced in this article:

[U1] https://www.npmjs.com/package/grunt-mutation-testing#available-mutations

[U2] http://lisacrispin.com

[U3] https://github.com/mike-rogers/rally-plugin

[U4] http://pitest.org

[U5] http://mike-rogers.github.io/2015/01/31/pairing-a-casual-chat/

[U6] http://accu.org/index.php/journals/1929

[U7] http://www.sep.com/

Making Time To Tend Code

Your Codebase as a Garden

by Rachel Davies

Codebases require tending, like a garden.

A question that software developers everywhere struggle with is how to make time to get things done in their codebase. Agile approaches seem to be mostly about piling in more features. Any developer knows there's a ton of work not directly tied up with these new features. Often it seems developers are left trying to beg, borrow, or steal time to take care of tasks that are hard to explain to non-technical people — with haphazard results. Being driven to deliver more features while sidelining their concerns has the effect of making developers feel like second-class citizens with no voice. Agile process being used as a stick to beat developers is a sad state of affairs, but there are ways that Agile process can also be used by developers to create protected time for work they need to get done.

Codebases are in some ways like gardens: to continue to be fruitful, they must be tended with care. Garden plants don't stay the same shape: as time passes they grow in all directions, eventually dying back to make room for others. Code also grows in different directions over time, and as patches of code start to yield less business value, we need to consider removing them. To walk easily around your garden, you have to cut back stray branches and sweep up to keep paths clear. Like gardeners, software developers need time to prune sprawling code, thin out overcrowded components, and tease out new services so they may flourish.

In my experience, Scrum-style planning tends to focus team effort on developing user-facing features that can easily be explained to business stakeholders. Extending the gardening metaphor, the items that appear in our

backlogs are like new garden features — plant a fruit tree so we have apples, install a shed so we can keep our tools safe, create a stunning herbaceous border in the front so we can make our neighbors envious, etc. Sprint planning doesn't cover time for weeding the existing flower beds, raking up dead leaves, or removing an ants' nest that's making the patio unusable. Developers need time to stay on top of technical maintenance alongside development of new features. Although a Scrum team should be able to add technical tasks to their backlog, typically the focus of a sprint is meeting the sprint goal, delivering all the shiny new features that will generate business value.

Developers often grumble that there's not enough time in a sprint to stay on top of the technical tasks. I'm talking about things that no stakeholder will ask for. For example, tidying up deployment scripts, new browser VMs for testing, upgrading to the latest release of something. Nearly always, code-tending tasks can be deferred over delivering more features. When we decide to put them off, we may also be increasing the work needed to sort them out. Anyone who has ever had a lawn knows that grass grows fast in the springtime — you need to get out with your mower every week or you'll soon regret it. Leave the grass until it's knee high, and you have to use a strimmer. Strimming long grass takes a lot longer than mowing, and leaves the grass looking roughly hacked back. Putting off care for your lawn turns a simple task into a laborious one.

To ensure more time is available to improve our software systems, developers may try to make code-tending work visible to their business stakeholders as "technical stories." We recently created a technical story to cover a rewrite of a stats pipeline to handle higher volumes of data. However, there are so many mundane tasks that it doesn't make sense to expose a large stream of technical stories to stakeholders unless it's going to seriously impact delivery of new features. Most Scrum teams have a bunch of technical tasks in their sprint backlog jostling for time alongside work on new features. When work on features and technical tasks are lumped together into a single line on a burndown chart and aggregated into team velocity, it's really hard for any team to get the balance between these types of work right.

Inspired by a Kanban technique, separating out different *Classes of Service* for feature development over support changes, I encouraged our XP teams at Unruly [U1] to separate out different types of work on their team boards. This enables our teams to balance their capacity across the different types of work: features, tech tasks, support, research. Doing this helps us maintain momentum on all of these different types of work. Sounds a bit complicated? Making different kinds of work more visible helps us not to neglect one kind of work over another.

We use different colored cards to represent different types of work. We use white cards for stories and green cards for code-tending work. We try to do refactoring as part of stories but sometimes we stumble across bigger issues that can be done later as green cards. How do we make time to do these green cards? Well the secret of XP is that we just do them. We only include work on stories in our velocity. Excluding green cards from our velocity means that it only represents our likely capacity for prioritizing the new stories. Now we have time to keep working on a steady stream of code-tending tasks. Our business stakeholders are happy with this because we keep delivering new features and they don't have to wrestle with trying to understand the priority of the technical stuff that we need to do to maintain our codebase.

All of our teams use simple tracking mechanisms to ensure that we maintain a balance between new features and technical improvement work. One team does dot counting [U2]. Other teams keep an area on their whiteboard for tracking where their time goes. They have their own system of colored magnets and hieroglyphs. These trackers are updated by the team at standup time so we spot if we're not striking the right balance. We later take photos of them to review in our team retrospectives.

Each team prioritizes their green cards weekly. To figure out which tasks are most important, we ask whether doing this work helps us deliver value to our business in the coming weeks. As one of our team leads puts it *"does this make our real days closer to ideal days?"*

Yes, there are busy times when we feel the right thing to do is deliver more customer-facing features over technical tasks. But often our system will force us to put on the brakes. Neglecting technical tasks will eventually slow us down — our tests take longer to run, it takes more time to deploy, developer motivation is sapped. We may even lose developers who are fed up with working around an aging technology stack.

Setting aside time to tend our code is essential for us to maintain fruitful software assets. Start by making technical tasks visible to your team. Prioritize these tasks so that the team works on the ones that will yield the most benefit. At standup, consider as a team, what effort should be put on code-tending tasks today. Keep track of how much time you spend on code-tending; this can help you strike the right balance against business-facing features. Reflect together as a team on how code-tending is going.

You may wonder whether putting prioritization of technical tasks in the hands of developers leads to business priorities being ignored. We prefer to consider that our developers are taking on more responsibility and accountability for protecting our software as a business asset. We also find this way of working helps developers keep their codebases pleasant and easy to work in.

About the Author

Rachel Davies coaches product development teams at Unruly [U3] in London. She is co-author of Agile Coaching [U4] and an invited speaker at industry events around the globe. Her mission is to create workplaces where developers enjoy delivering valuable software. Rachel is a strong advocate of XP approaches and an organiser of Extreme Programmers London meet-up. Follow her @rachelcdavies on Twitter.

External resources referenced in this article:

[U1] http://tech.unruly.co/

[U2] http://rachelcdavies.github.io/2014/03/03/dot-counting.html

[U3] http://unruly.co/

[U4] https://pragprog.com/book/sdcoach/agile-coaching

Functional Snippets

Applicative Functors

by Chris Eidhof, Wouter Swierstra, and Florian Kugler

Here's the next installment in our series designed to help you get a grip on functional programming in Swift.

Swift opens up a whole new world of programming. To quote Swift's creator, Chris Lattner: " 'Objective-C without the C' implies something subtractive, but Swift dramatically expands the design space through the introduction of generics and functional programming concepts."

In this series, we use small snippets of Swift code to explore this new world of programming, as well as demonstrate how Swift supports the functional paradigm.

In last month's snippet we used a login function like this as an example:

```
func login(email: String, pw: String, success: Bool -> ())
```

Often you'll have the situation that the strings you want to hand to the login function are optional values. Then you'd have to write something like this:

```
if let email = getEmail() {
    if let pw = getPw() {
        login(email, pw) { println("success: \($0)") }
    } else {
        // error...
    }
} else {
    // error...
}
```

There's another way to tackle these situations: we can make use of the fact that optionals in Swift are an instance of applicative functors. Yes, that sounds confusing and crazy, but don't worry, we'll simply make an example. For applicative functors, such as optionals, we can define the <*> operator:

```
infix operator <*> { associativity left precedence 150 }
func <*><A, B>(lhs: (A -> B)?, rhs: A?) -> B? {
    if let lhs1 = lhs {
        if let rhs1 = rhs {
            return lhs1(rhs1)
        }
    }
    return nil
}
```

In short, this operator takes the right hand operand and applies it to the function on the left hand side if neither of them is nil. Using this operator and the curry function from last month's snippet we can write the example from above like this:

```
if let f = curry(login) <*> getEmail() <*> getPw() {
    f { println("success \($0)") }
} else {
    // error...
}
```

Much nicer, isn't it? Admittedly, it looks pretty foreign at first, but it's a wonderful opportunity to practice some out-of-the-box thinking!

Update: In this snippet [U1] we show how this can be improved with Swift 1.2's multiple optional unwrapping.

About the Authors

Along with Daniel Eggert, Chris Eidhof and Florian Kugler created objc.io [U2], a periodical about best practices and advanced techniques for iOS and OS X development. Eidhof, Kugler, and Wouter Swierstra also wrote the book *Functional Programming in Swift* [U3], in which they explain the concepts behind functional programming and how Swift makes it easy to leverage them in a pragmatic way, in order to write clearer and more expressive code.

External resources referenced in this article:

[U1] http://www.objc.io/blog/2015/02/02/functional-snippet-18-unwrapping-multiple-optionals/

[U2] http://www.objc.io/

[U3] http://www.objc.io/books/

Antonio on Books

Thirty New Books

by Antonio Cangiano

Antonio looks at all the new tech books of note.

Antonio Cangiano is the author of the excellent Technical Blogging [U1] and you can subscribe to his reports on new books in technology and other fields here [U2].

By now it should be clear to anyone involved in technology that data science and data analysis are some of the hottest topics around. Without trying to insist on the topic too much, and simply reflecting what the market of technology books has been promoting, this month's pick is once again about data analysis. Specifically with R, which happens to be one of the primary tools of the trade.

The newly released second edition of *R in Action: Data Analysis and Graphics with R* by Robert Kabacoff does an excellent job at covering the subject, by focusing on practical business problems and use cases rather than simply illustrating the programming language. In the beginning it also covers some important fundamentals of statistics. This is, I'm happy to say, an easy book to recommend.

Our Staff Pick: *R in Action: Data Analysis and Graphics with R* [U3] • By *Robert Kabacoff* • ISBN: *1617291382* • Publisher: *Manning Publications* • Publication date: *June 6, 2015* • Binding: *Paperback* • Estimated price: *$33.81*

Mastering matplotlib [U4] • By *Duncan M. McGreggor* • ISBN: *1783987545* • Publisher: *Packt Publishing - ebooks Account* • Publication date: *June 4, 2015* • Binding: *Paperback* • Estimated price: *$39.99*

Compiler Design: Syntactic and Semantic Analysis [U5] • By *Reinhard Wilhelm, Helmut Seidl, Sebastian Hack* • ISBN: *3642435912* • Publisher: *Springer* • Publication date: *June 23, 2015* • Binding: *Paperback* • Estimated price: *$65.99*

Distributed Graph Algorithms for Computer Networks [U6] • By *Kayhan Erciyes* • ISBN: *1447158504* • Publisher: *Springer* • Publication date: *June 6, 2015* • Binding: *Paperback* • Estimated price: *$79.99*

Hypergraph Theory: An Introduction [U7] • By *Alain Bretto* • ISBN: *3319033700* • Publisher: *Springer* • Publication date: *June 21, 2015* • Binding: *Paperback* • Estimated price: *$109.00*

The Hacker Playbook 2: Practical Guide To Penetration Testing [U8] • By *Peter Kim* • ISBN: *1512214566* • Publisher: *CreateSpace Independent Publishing Platform* • Publication date: *June 20, 2015* • Binding: *Paperback* • Estimated price: *$22.49*

Spring REST [U9] • By *Balaji Varanasi, Sudha Belida* • ISBN: *1484208242* • Publisher: *Apress* • Publication date: *June 17, 2015* • Binding: *Paperback* • Estimated price: *$35.24*

Pro XAML with C#: From Design to Deployment on WPF, Windows Store, and Windows Phone [U10] • By *Buddy James, Lori Lalonde* • ISBN: *1430267763* •

Publisher: *Apress* • Publication date: *June 30, 2015* • Binding: *Paperback* • Estimated price: *$43.99*

Python for complete beginners: A friendly guide to coding, no experience required [U11] • By *Dr Martin Jones* • ISBN: *1514376989* • Publisher: *CreateSpace Independent Publishing Platform* • Publication date: *June 18, 2015* • Binding: *Paperback* • Estimated price: *$9.00*

Beginning SAS Programming: a true beginner's guide for learning SAS [U12] • By *Yufeng Guo* • ISBN: *1514218992* • Publisher: *CreateSpace Independent Publishing Platform* • Publication date: *June 4, 2015* • Binding: *Paperback* • Estimated price: *$38.00*

Beginning App Development with Parse and PhoneGap [U13] • By *Wilkins Fernandez, Stephan Alber* • ISBN: *1484202368* • Publisher: *Apress* • Publication date: *June 29, 2015* • Binding: *Paperback* • Estimated price: *$35.99*

Beginning Python Games Development, Second Edition: With PyGame [U14] • By *Harrison Kinsley, Will McGugan* • ISBN: *1484209710* • Publisher: *Apress* • Publication date: *June 23, 2015* • Binding: *Paperback* • Estimated price: *$33.99*

Numpy Beginner's Guide - Third Edition [U15] • By *Ivan Idris* • ISBN: *1785281968* • Publisher: *Packt Publishing - ebooks Account* • Publication date: *June 30, 2015* • Binding: *Paperback* • Estimated price: *$44.99*

Pro Vagrant [U16] • By *Wlodzimierz Gajda* • ISBN: *1484200748* • Publisher: *Apress* • Publication date: *June 8, 2015* • Binding: *Paperback* • Estimated price: *$35.61*

Murach's Beginning Java with NetBeans [U17] • By *Joel Murach, Michael Urban* • ISBN: *1890774847* • Publisher: *Mike Murach & Associates* • Publication date: *June 30, 2015* • Binding: *Paperback* • Estimated price: *$57.50*

Elixir in Action [U18] • By *Saša Juric* • ISBN: *161729201X* • Publisher: *Manning Publications* • Publication date: *June 14, 2015* • Binding: *Paperback* • Estimated price: *$26.94*

Building Splunk Solutions: Splunk Developer Guide [U19] • By *Grigori Melnik, Dominic Betts, David Foster, Liying Jiang* • ISBN: *1512356077* • Publisher: *CreateSpace Independent Publishing Platform* • Publication date: *June 25, 2015* • Binding: *Paperback* • Estimated price: *$35.99*

XSLT Jumpstarter: Level the Learning Curve and Put Your XML to Work [U20] • By *David James Kelly* • ISBN: *0913465038* • Publisher: *Pragmatic Bookshelf* • Publication date: *June 8, 2015* • Binding: *Paperback* • Estimated price: *$21.68*

Mastering Pandas [U21] • By *Femi Anthony* • ISBN: *1783981962* • Publisher: *Packt Publishing - ebooks Account* • Publication date: *June 22, 2015* • Binding: *Paperback* • Estimated price: *$44.99*

Web Programming for Business: PHP Object-Oriented Programming with Oracle [U22] • By *David Paper* • ISBN: *0415818052* • Publisher: *Routledge* • Publication date: *June 10, 2015* • Binding: *Paperback* • Estimated price: *$49.38*

Lean Websites [U23] • By *Barbara Bermes* • ISBN: *0992279461* • Publisher: *SitePoint* • Publication date: *June 15, 2015* • Binding: *Paperback* • Estimated price: *$22.41*

Swift by Example [U24] • By *Giordano Scalzo* • ISBN: *1785284703* • Publisher: *Packt Publishing - ebooks Account* • Publication date: *June 30, 2015* • Binding: *Paperback* • Estimated price: *$39.99*

Beginning HTML5 Media: Make the most of the new video and audio standards for the Web [U25] • By *Silvia Pfeiffer, Tom Green* • ISBN: *1484204611* • Publisher: *Apress* • Publication date: *June 17, 2015* • Binding: *Paperback* • Estimated price: *$30.91*

Developing for Apple Watch: Your App on Their Wrists [U26] • By *Jeff Kelley* • ISBN: *1680500686* • Publisher: *Pragmatic Bookshelf* • Publication date: *June 18, 2015* • Binding: *Paperback* • Estimated price: *$12.77*

Transforms in CSS: Revamp the Way You Design [U27] • By *Eric A. Meyer* • ISBN: *1491928158* • Publisher: *O'Reilly Media* • Publication date: *June 14, 2015* • Binding: *Paperback* • Estimated price: *$6.00*

Web Scalability for Startup Engineers [U28] • By *Artur Ejsmont* • ISBN: *0071843655* • Publisher: *McGraw-Hill Education* • Publication date: *June 23, 2015* • Binding: *Paperback* • Estimated price: *$25.50*

Machine Learning Projects for .NET Developers [U29] • By *Mathias Brandewinder* • ISBN: *1430267674* • Publisher: *Apress* • Publication date: *June 29, 2015* • Binding: *Paperback* • Estimated price: *$39.99*

Colors, Backgrounds, and Gradients: Adding Individuality with CSS [U30] • By *Eric A. Meyer* • ISBN: *1491927658* • Publisher: *O'Reilly Media* • Publication date: *June 14, 2015* • Binding: *Paperback* • Estimated price: *$10.91*

Mastering AWS Development [U31] • By *Uchit Vyas* • ISBN: *1782173633* • Publisher: *Packt Publishing - ebooks Account* • Publication date: *June 25, 2015* • Binding: *Paperback* • Estimated price: *$49.99*

HTML5 Programmer's Reference [U32] • By *Jonathan Reid* • ISBN: *1430263679* • Publisher: *Apress* • Publication date: *June 30, 2015* • Binding: *Paperback* • Estimated price: *$45.14*

Pragmatic Bookstuff

Here's what's up with the Pragmatic Bookshelf and its authors.

Here's what's happening at Pragmatic Bookshelf.

What's Hot

1	NEW	Deliver Audacious Web Apps with Ember 2
2	NEW	Beyond Legacy Code
3	1	Real-World Kanban
4	4	Programming Elixir
5	NEW	Predicting the Unpredictable
6	7	Agile Web Development with Rails 4
7	NEW	The Definitive ANTLR 4 Reference
8	NEW	Practical Vim
9	5	Clojure Applied
10	10	iOS 8 SDK Development
11	11	Ruby Performance Optimization

Who's Where When

And here's what the authors are up to:

2015-07-09 **Extensions Programming Workshop**
Chris Adamson (author of iOS 8 SDK Development [U1])
CocoaConf Columbus [U2]

2015-07-10 **Revenge of the 80s: Cut/Copy/Paste, Undo/Redo, and More Big Hits**
Chris Adamson
CocoaConf Columbus [U3]

2015-07-11 **Video Killed the Rolex Star**
Chris Adamson
CocoaConf Columbus [U4]

2015-07-20 **Mini-Munging: The Joy of Small Data. Big data is trendy, but most data is small. What's a useful toolkit for processing and manipulating data on a scale that doesn't require us to spin up vast clusters of servers, and where does Ruby fit in?**
Rob Miller (author of Text Processing with Ruby [U5])
Brighton Ruby Conference 2015 [U6]

2015-07-21 **Design Your Agile Project**
Johanna Rothman (author of Behind Closed Doors [U7] , Manage It! [U8] , Manage Your Project Portfolio [U9] , Hiring Geeks That Fit [U10] , Manage Your Job Search [U11] , and Predicting the Unpredictable [U12])
Uberconf, Denver, CO [U13]

2015-07-22 **Tuning Your Agile Team**
Johanna Rothman
Uberconf, Denver, CO [U14]

2015-08-01 **Agile Teaching - Some things I've learned teaching Ruby to beginners.**
Mark Sobkowicz (author of Learn Game Programming with Ruby [U15])
Burlington Ruby Conference, Burlington, VT [U16]

2015-09-08 **Treat your Code as a Crime Scene**
Adam Tornhill (author of Your Code as a Crime Scene [U17])
Swanseacon, Swansea, Wales, UK [U18]

2015-09-23 **Using Agile contracts in Sweden**
Mattias Skarin (author of Real-World Kanban [U19])
Upphandla IT, Göteborg

2015-09-24 **Lead organizer**
Alex Miller (author of Clojure Applied [U20])
Strange Loop - St. Louis, MO [U21]

2015-10-05 **Treat Your Code as a Crime Scene**
Adam Tornhill
GOTO Copenhagen, Denmark [U22]

2015-10-07 **Full day Workshop: Code as a Crime Scene**
Adam Tornhill
GOTO Copenhagen, Denmark [U23]

2015-10-20 **Lean Kanban conference moderator. Topic: Evolving products and organisations Fast Feedback.**
Mattias Skarin
Lean Kanban Nordic - Stop Starting 2015, Stockholm [U24]

2015-10-26 **Software Design, Team Work and other Man-Made Disasters**
Adam Tornhill
Trondheim Developer Conference, Norway [U25]

2015-11-04 **Mine Social Metrics From Source Code Repositories**
Adam Tornhill
Øredev 2015, Malmö, Sweden [U26]

2015-11-16 **Coming soon!**
Mattias Skarin
Lean Kanban Central Europe 2015

What's Happening

But to really be in the know, you need to subscribe to the weekly newsletter. It'll keep you in the loop, it's a fun read, and it's free. All you need to do is create an account on pragprog.com [U27] (email address and password is all it takes) and select the checkbox to receive newsletters.

Solution to Pub Quiz

Here's the solution to this month's pub quiz:

Y	S	R	T	D	H	B	A	I
I	T	H	R	A	B	Y	D	S
A	D	B	I	S	Y	T	H	R
H	A	D	B	I	R	S	Y	T
S	R	T	Y	H	D	A	I	B
B	I	Y	S	T	A	H	R	D
D	B	S	A	Y	I	R	T	H
R	Y	I	H	B	T	D	S	A
T	H	A	D	R	S	I	B	Y

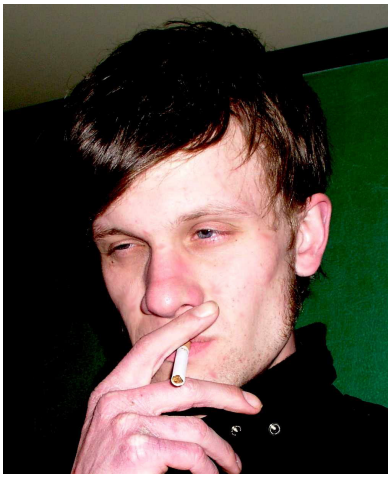
Rearranging the letters from any row, column, or smaller square reveals what some programmers are celebrating this month: BIRTHDAYS.

Shady Illuminations

You Are a Thing

by John Shade

The Internet of Things is not about what you think it is about.



Today's lesson is on the Internet of Things, or IoT, in which the unit of information is not a bit but an iot, plural iota. You probably didn't know that was where the word came from. Don't thank me, I'm just doing my job, spreading fertilizer in the garden of thought.

OK, here's what we know For Sure about the Internet of Things:

The IoT is different from the Plain Old Internet, or POI.

POI is mushy, gray, and tasteless. This we know, but why is that? Because POI is about connecting people, and people in the mass are a mess. POI is random beings, deeply connected. Everybody gets Deeply Intertwingled. A telepresent chat orgy, a Sandstone of the Mind, the new anti-intellectual home of mankind, where the math of conversational connectedness rules: exponential growth, reduced to the lowest common denominator. The ultimate purpose of POI is to spawn infinite Facebook threads, each branching fractally into a higher infinity of Godwin rule invocations.

The IoT is not that. The IoT is about connecting devices, generously including devices of the firmware or software persuasion. Letting my accounts payable talk to your accounts receivable witout the unhelpful intrusion of the whiny human element. Letting my outdoor thermometer talk to your car's GPS, so it can redirect you to abort that visit and go to the water park instead. Letting my security camera lock onto your annoying drone and trigger the sprinklers. OK, these may not be the right examples, but you get the idea. Things talking to things, with no humans involved.

This is what we know For Sure. It's a nice, simple dichotomy: POI is about people and the IoT is about things. And like most things we know For Sure, it's pasture pizza.

Don't believe me? Check your Smalltalk class hierarchy. Class Person inherits from class Thing, right? (OK, maybe from class Object. Cripes, I haven't checked my Smalltalk class hierarchy lately. Don't be pedantic.) Point is: People are Things and people are things.

The IoT isn't about things as opposed to people. People are part of the IoT. The most important part. (I hated saying that. It sounds so Up With Peopleish.)

You don't believe me, but what if you did? In that case, what is the IoT about really? I'll tell you: the IoT is about eliminating the middlething. Wherever there's a thing that's sitting on data, the IoT wants to route around it. Yeah, sometimes that's a person, but the process is cumulative. Each rerouting creates a different network and you have to recompute the efficiencies. Sometimes you'll want to reinsert a person.

In fact, I maintain, ultimately you have to bring a human into every IoT network, because IoT, like all of technology, only exists to amuse humans.

You're not going to buy that if you still cling to some antiquated notion that we're here on this planet to Get Things Done rather than to Amuse Ourselves However We Can While Waiting for the Ax to Fall, so I'll just give you an example of what I'm talking about.

There's a piece of hang gliding technology called a variometer. (I've never hang glided and know nothing about it, but I can do the google.) It exists to tell the human exactly what his vertical speed is, a bit of data that I'd imagine would come in handy. In fact it's the sort of data you'd like to feel directly and instantaneously in order to ride the air currents like you actually belonged there. And it really works that way with acceleration: human gliders are good at sensing changes in velocity when they first hit a thermal, and immediately responding to it. But they're really bad at sensing constant velocity, so they have these little devices that show them their velocity.

The IoT looks at this system and asks, shouldn't that data go to some mechanical device rather than be presented on a screen to a human? Trying to route around the weak link.

But that's the wrong question. It's undeniably true that often the human bean is the weak lunk in information-centric systems. But that's just evidence of incorrect centering. In hang gliding, as in everything that actually matters, the human is at the center, and the goal is to increase control and power for that human. It's all about the human and his or her enjoyment. So route the data to the human, but use all the available input modalities. You don't want to read your velocity on a screen, you want to feel it, just like you feel acceleration. So variometers are on the right track, they just need to be wired into the flyer's nervous system more organically. And might as well loop in the human's internal systems, too. Is your heart racing? That should be feedback to the glider itself. Are your palms sweating? Then your grip may be less precise, and that needs to be adjusted for. Your nervous twitches can be adjusted for, but also processed as useful somatofreakout data. The goal should be not to eliminate the human, but to integrate the human into something bigger.

You might think that hang gliding is one isolated and eccentric example, but I would argue that hang gliding is the perfect metaphor for human existence. Better even than surfing, it's my go-to metaphor. I recommend it to your consideration.

Ultimately, computers and other tools are extensions of the human nervous system — both sensory and motor. We talk about increasing productivity, but come on, that never happens. Technology is just about giving us superpowers. The IoT is about making us into superthings.

About the Author

John Shade was born in Montreux, Switzerland, on a cloudy day in 1962. Subsequent internment in a series of obscure institutions of ostensibly higher learning did nothing to brighten his outlook. He is currently the Resident Nihilist at Imlac Corporation.