The Pragmatic Bookshelf

PragPub

The First Iteration

IN THIS ISSUE

Issue #45
March 2013

Contents

FEATURES

DEPARTMENTS

Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail support@pragprog.com, phone +1-800-699-7764. 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

Up Front

Passion and Pragmatism

by Michael Swaine

Passion and pragmatism is the theme of this issue. Turns out they can go hand-in-hand.

Pragmatism—well, yes, this magazine is published by the Pragmatic Bookshelf, but we're specifically talking about making a living as a programmer. Johanna Rothman looks at one side of the process of hiring, particularly finding people who fit into your agile team. And Andy Lester looks at the same situation from the perspective of the programmer looking to be hired.

Matthias Günther offers up some pragmatic advice, too, in his series on the Pomodoro Technique.

But for most of you, programming isn't just a paycheck, it's your passion. Jonathan Rasmusson and Steven K. Roberts both weigh in on the side of following your passion even when the world says you're crazy. Jonathan articulates the crucial difference between personal startups and speculative ones. You shouldn't make decisions about a personal startup in the same way you would make decisions about a speculative one. And Steven, whose own projects include building a technobike and riding it across America, continues his series on following your passion through gonzo engineering.

Where the magic happens, though, is in that spot where passion and pragmatism come together. So Steven reveals all of his funding secrets—ways he's found to get strangers to give you the wherewithal to pursue your personal dream. Jonathan reminds us that fortunes are sometimes made by people who had the courage to follow their passions. And Andy coaches you on how to find the work that you can be passionate about.

Elsewhere in the issue, Paul Callaghan continues his exploration of his passion, functional programming, and the irascible John Shade shows his poetic side. Spoiler: it isn't pretty.

Choice Bits

What Twitter Is Good For

We follow Twitter so you don't have to.

What's Hot

Top-Ten lists are passé—ours goes to 11. These are the top titles that folks are interested in currently, along with their rank from last month. This is based solely on direct sales from our online store.

1	NEW	Programming Ruby 1.9 & 2.0
2	NEW	Lean from the Trenches
3	NEW	3D Game Programming for Kids
4	4	Agile Web Development with Rails
5	NEW	101 Design Ingredients to Solve Big Tech Problems
6	5	The Definitive ANTLR 4 Reference
7	2	Good Math
8	9	Practical Vim
9	11	Core Data
10	10	Programming Ruby 1.9
11	NEW	The Cucumber Book

What Twitter is good for:

Self-assessment

- *Found a new power law distribution today. It's hopelessly geeky that it excites me so much but I'm learning to accept that about myself.* — @KentBeck
- *Felt out of place on a flight w/ 80% consultants. Am I a free spirit or just a child with my farm-animal decal laptop?* — @verbicidal
- *Tabs in JavaScript make me sad.* — @tswicegood
- *I'm personally comfortable with the armadillo races, but appalled by one of the other entertainment options: a gunfight (ugh!) #cscw2013* — @asbruckman
- *I'm having a day where I actually feel like I know what I'm doing.* — @mtnygard
- *Estimated I've spent ~500 hours making cappuccino. I should be an expert by age 165.* — @jbrains

Sharing the Secret to Your Productivity

- *Drinking a Howard Street IPA by ThirstyBear Brewing Company.* — @pleia2
- *It's amazing how many questions can be answered with Laphroaig.* — @scottdavis99
- *Today's motto: Powered by Girl Scout cookies. #NICAR13* — @MacDiva

- *If you put a million monkeys at a million keyboards, one will eventually write a Java program. The rest of them will write Perl programs.* — @barse
- *Today I'm going to do all of the things.* — @SaraJChipps

Sharing Your Insights

- *Forget 3D printing. MIT has already moved on to 4D. 4D refers to 3D items that transform themselves over time.* — @sara6633
- *In customer acquisition, your churn rate is your most important metric. Trust me.* — @HackerChick
- *The startup (revenue) hockey stick is really just a mullet: business in the front, party in the back.* — @HackerChick
- *To people at lectures/conferences everywhere: please turn off the sound on your phone while you take photos. Fake shutter ≠ real shutter.* — @natalierachel
- *I wear a bow tie now. Bow ties are cool.* — @neilhimself

Just Sharing

- *Moved the dog's water dish to the bathroom in hopes of keeping him from drinking from the toilet. Now he has toilet water for dessert. Ugh.* — @storming
- *Mtn View Voice "Toxic vapors found in #Google offices" (Whisman area, new offices). So much new development there. :(* — @LatinasInC
- *I have been told that I fail at RSS, and need to declare RSS bankruptcy. (I have 14k items unread in NetNewsWire.)* — @snipeyhead
- *I'm gonna start looking up and to the right all the time so when I get Google Glass you won't be able to tell if I'm ignoring you.* — @seldo
- *The 1px problems will be the death of me.* — @zahnster

Second-Guessing Marissa Mayer

- *Technology allows millions of people to work from home. Yahoo, a big tech firm, is trying to stop them.* — @TheEconomist
- *FWIW, I never doubted Yahoo's new policy about working in office. Companies can gather lots of dead wood. Let's get to know our colleagues.* — @davewiner
- *What JC Penney has in common with Yahoo is that people tend to goof off when they fall into despair.* — @dhh
- *Next time we second guess a decision from Marissa Mayer we should remember she's data driven and doesn't make decisions like this wildly.* — @kellabyte

Who Are Those Guys?

First, they're not all guys. Second, we have to confess that we cleaned up their punctuation and stuff a little. OK, who we followed this month: Kent Beck, Natalie Be'er, Amy Bruckman, Sara Chipps, Sara Czyzewicz, Scott Davis, Andrew W. Deane, The Economist, Abby Fichtner, Neil Gaiman, DHH, Elizabeth Krumbach, Latinas in Computing, Michael Nygard, Terri O, Thomas Powell, J. B. Rainsberger, Jade Rauenzahn, Lars Sjögreen, snipe , Kelly Sommers, Amy Stephen, Stormy, Travis Swicegood, verbicidal, Laurie Voss, Dave Winer, and Chrys Wu.

Fair's fair. You can follow us at www.twitter.com/pragpub [U1].

What If You Don't Want to Pivot?

Sometimes It's Personal

by Jonathan Rasmusson

Sometimes when the world tells you to pivot, you really need to stay the course.

Anyone who has read Eric Ries's excellent book *The Lean Startup* knows that as soon as you get a better idea what your customers want, you should "pivot" and act on your learnings.

Except it isn't that easy. What if the world says pivot but you don't want to? Does that mean you're wrong? Too married to your idea? Are you going to be one of those entrepreneurs who wastes their time building something nobody wants?

Here I would like to share some thoughts around pivots. Specifically when entrepreneurs should pivot, when they shouldn't, and why personal ideas need to be treated differently from speculative ones.

The Last Thing the World Needs Is Another Book on Agile

That was the advice I got from an ex-colleague of mine when I asked if he would consider writing the foreword for my book The Agile Samurai [U1]. From his point of view there were already enough books on agile and the last thing the world needed was another. And of course, from his point of view, he was right.

- A ton of books had already been written on agile.
- Agile had been around for a long time.
- Everyone was doing it (or so it seemed).
- And there wasn't anything new that really needed to be said.

Taking these data points, and naively applying the concept of the Lean Startup pivot, it would have made sense to pivot the focus of my book to something sexier or new.

Except I didn't want to.

I was married to the concept of this book. A simple 200-page distillation easy enough for management yet deep enough for aspiring teams. I didn't particularly care if no one was going to read it. This was the book I wished I had when learning agile. It didn't exist. So I was going to write it.

It was after writing *Samurai* that I read Eric's book. And it was here that I realized that if I had blindly followed some of the Lean Startup practices (which I mostly agreed with), *Samurai* would never have been written. What bugged me was that I couldn't explain why.

I know writing a book isn't the same as forming your own full-on startup, but the question remained. Why did some ideas seem easier to pivot on than others?

It wasn't until I came across a presentation by Adrian Smith that I found the "ideas" I was looking for.

Speculative vs Personal Ideas

There are two types of ideas when it comes to startups. Speculative and personal.

Speculative	Personal
In it for the money	In it for yourself
Would abandon quickly	Would never abandon
Have a choice whether to build	No choice—have to build
Hard work	Hard play
Requires resources of others	Requires only you
Objective, data driven	Emotional, logic doesn't apply

Speculative ideas are all about the money. You see an opportunity and you go for it. Customers aren't responding? No problem, pivot and experiment until you discover what they do like. Pivoting, objectivity, competition and hard data rule here and this is where the concepts of The Lean Startup and other formal methods shine. It's nothing personal—it's just business. This is the realm of the speculative idea.

Personal ideas are the exact opposite. You aren't in it for the money. You pursue them because you want to. No one is paying you. Most likely no one ever will. It's your passion. Your love. It may be your obsession. You don't care if anyone is looking. It's that itch you have to scratch. That mountain you have to climb. It's not work—it's play. And you can't wait to get home from whatever it is you do during the day so you can immerse yourself in it for a few hours before you collapse and go to bed.

Danger Zone

And this is where people can get hurt: treating speculative ideas as personal, and personal ideas as speculative.

Treating speculative ideas as personal ones is exactly what Eric is trying to shield us against with the principles outlined in *The Lean Startup*. He nails it when he says there is nothing more wasteful than companies building things nobody wants. We become so enamored of our idea that we think if we simply build it others will come. Then, of course, they don't and the rest is history. The principles of *The Lean Startup* serve us well here.

But the Lean Startup pivot can be applied equally naively too. If you've got a personal idea, don't pivot just because the world isn't with you. Some of our greatest inventions and creations have come about precisely because founders saw things differently and refused to pivot when the world was telling them to.

They couldn't find validation for what they were doing because it didn't exist. Even if they could have shown their product or idea to others, few would have got it because no one would have understood (e.g. Twitter).

And this is the most dangerous time for a work of art. Most are killed just before the finish line because we become afraid. We are afraid of what it will mean to finish. We are afraid of success. We are afraid of failure. We are afraid of what others will think. We are afraid of not following the tenets of a popular book.

And it's this fear that kills our most cherished ideas. And I fear that when applied naively, the pivot will become another excuse to quit before our ideas see the light of day.

If this topic fascinates you I recommend Steven Pressfield's *The War of Art*.

Is Your Idea Speculative or Personal?

Look—startups are hard. I don't pretend to have any magical insight or formula for making them easier. Eric's book is excellent and I have personally failed in two startups that would have definitely benefited from his advice.

But if it's personal, gives you great joy, and you can't wait to get home to work on it, enjoy it while it lasts! For you have something special. Something that most haven't experienced since childhood. The sheer joy and flow that comes from creating. And this is something that should never be abandoned.

"We think the Mac will sell zillions, but we didn't build the Mac for anybody else. We built it for ourselves. We were the group of people who were going to judge whether it was great or not. We weren't going to go out and do market research. We just wanted to build the best thing we could build."—Steve Jobs

Thank you Adrian Smith for this presentation on Agile for Startups [U2].

About the Author

Jonathan Rasmusson is the author of The Agile Samurai [U3]. As an experienced entrepreneur and former agile coach for ThoughtWorks, Jonathan has consulted internationally, helping others find better ways to work and play together. When not coaching his sons' hockey teams or cycling to work in the throes of a Canadian winter, Jonathan can be found sharing his experiences with agile delivery methods at his blog, The Agile Warrior [U4]. For more information on the Agile Inception Deck check out The Agile Samurai [U5].

Send the author your feedback [U6] or discuss the article in the magazine forum [U7].

External resources referenced in this article:

[U1] http://pragprog.com/refer/pragpub45/book/jtrap/the-agile-samurai

[U2] http://www.slideshare.net/adrianlsmith/agile-for-startups

[U3] http://www.pragprog.com/refer/pragpub45/titles/jtrap/the-agile-samurai

[U4] http://agilewarrior.wordpress.com

[U5] http://www.pragprog.com/refer/pragpub45/titles/jtrap/the-agile-samurai

[U6] mailto:michael@pragprog.com?subject=Rasmusson

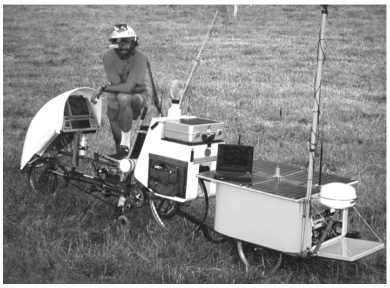
[U7] http://forums.pragprog.com/forums/134

Launching a Gonzo Engineering Project

III: The Business Angle and Getting Educated

by Steven K. Roberts

The infamous 580-pound, 105-speed BEHEMOTH, with Mac, SPARC, and DOS environments as well as satellite datacomm, HF/VHF/UHF ham radio, heads-up display, head mouse, handlebar keyboard, 6-level security system, speech synthesis, 72 watt solar array, and deployable landing gear to keep the monster upright on killer hills. The bike now resides in The Computer History Museum.



This is the third installment in a series of articles unlike anything we've ever published. Steven K. Roberts has figured out how to live passionately, pursuing crazy dreams and building fantastic machines (like BEHEMOTH and Microship, both of which are pictured and briefly described here) and going on amazing adventures. He calls what he does Gonzo Engineering, and in this series he tells you everything you need to know in order to pursue your own crazy gonzo engineering dream.

The Business Angle

Before your eyes glaze over at the word "business," let me hastily say that there is nothing in here about setting up a company! Lots of people can tell you more about that than I can, and although this discussion covers a lot of territory, you won't find me talking about how to run a corporation, nor even a sole proprietorship with shoebox accounting. I find that stuff hopelessly confusing, and am thus perennially at the mercy of people who are comfortable with financial and legal matters.

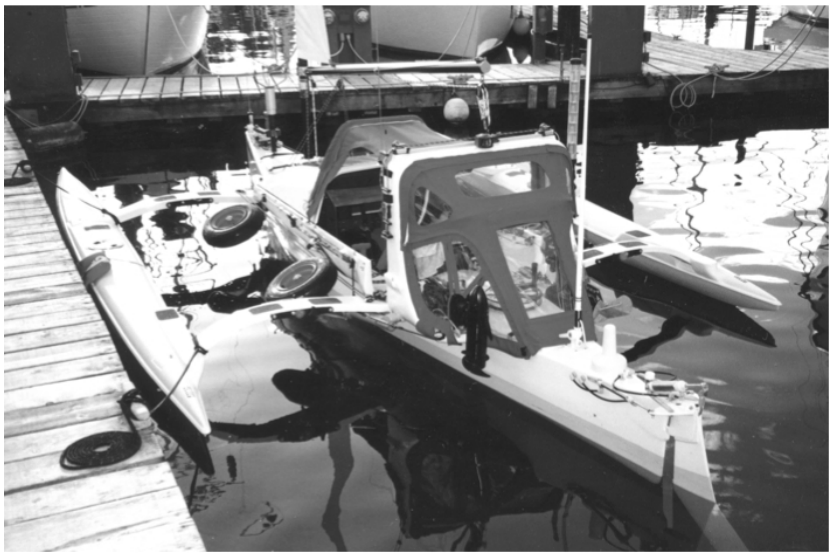
Still, you can use the system to your advantage (even *without* running a shell game, which I don't recommend... stress and creativity don't mix very well). An obvious requirement of managing a huge under-funded project is to minimize your costs, and the first step in that direction is to create a *bona fide* business entity encompassing the field in which you are working—an enterprise that actually shows cash flow and can withstand an audit. In my case, since I don't want to be a boat builder, I have to come up with something (besides manufacturing Microship clones) that renders all project expenses honestly tax deductible. Fortunately, I've been a writer for a long time, and with an endless stream of articles and the occasional book about these projects, they can legitimately be defined as "research." The only tax-law requirement is that I show a profit occasionally, and somehow, bad work habits notwithstanding, I manage to do so now and then (with a little help from an online store, affiliate links, and spin-off nickel generators).

I want to give you a couple of basic ideas that can help cover the costs while wrapping your pet obsession in a cloak of business legitimacy.

It has become axiomatic that a key component of a large independent project is "buy-in" by the public (and, as we shall see shortly, a volunteer community). This has certainly been the case with the Microship and its land-based predecessors, but you also see it in open-source development communities such as those hosted on Sourceforge: there is a sort of "critical mass of publicity" that has to be achieved for a project to take on a life of its own. One of the best ways to accomplish this is to build a related business based on information and spin-off dissemination... magazine writing, self-published technical monographs, system documentation, shareware authoring, project kits, or

anything else that leverages the work you actually *want* to do in order to simultaneously generate income and build exposure. There are very few professions in which you get paid to advertise, but that's basically what publishing is all about—the more you do, the more you create brand recognition and demand (assuming you're reasonably good at it). Magazine articles can lead to columns, which in turn can lead to books, which eventually, if you're lucky, can lead to a lucrative speaking career and a royalty stream.

Alas, many technical people either can't write very well or just don't want to; an old wag once said that writing is accomplished by staring at a blank page until little drops of blood appear on your forehead, and I know a lot of brilliant geeks who just don't have the patience or skills to make a go of it. But there are other means to the same end. Let's talk about consulting.



The Microship, the result of an 8-year development project involving extensive sponsorship, students, and volunteer teams. This is an amphibian pedal/solar/sail micro-trimaran with retractible wheels, hydraulic systems, 480 watts of peak-power-tracked solar panels, and zippy performance under sail. BEHEMOTH, the Microship, and the later Nomadness project are all documented at microship.com.

The best generalization I can give you is that the boundaries between specialties are where it's at. It is no accident that most projects in the domain of gonzo engineering are, by their nature, comprised more of new ways of combining existing technologies than of linear envelope-pushing; the latter, while honorable and necessary for ongoing industrial progress, is less likely to yield the kinds of breakthroughs that make the media flock to your door. It's not that there's anything wrong with it, it's just that individuals have a much harder time with "straight ahead" advances in the state of the art than do well-funded companies... that sort of work *does* lend itself well to structured engineering methods and thus tends to be the most likely course of corporate product development (think Moore's Law).

But an obsessed individual making leaps of intuition in the middle of the night is almost inevitably looking at new interactions between existing ideas—making novel connections across great chasms (like finding a mathematical hack to avoid the rotation and scaling problem associated with image recognition, based on the logarithmic polar mapping discovered through end-to-end electron

microscopy of the optic nerve). So it is likely that a gonzo engineering project already has some of this cross-boundary action happening. Why is this relevant?

Because the hottest consulting action is almost never within your own specialty; it's when you take your accumulated knowledge, cross over to a client who speaks a different language in an unrelated field, and solve Major Problems. If you know a lot about spread-spectrum data transmission, do you try to sell yourself to radio manufacturers that already have cubicles full of experts? Or do you become the hero of the hour to some company that has run into a wall trying to move data between ceramic foundries, deploying a few off-the-shelf radios and beam antennas along with a nice fat invoice for your trouble?

Note the parallel here. The best consulting opportunities feel a lot like a gonzo engineering project, and, if you're clever, can even use the same components and tools. So at the beginning of the mad quest to build your system, whether it's a Microship or the world's first standalone *Refrangible Densiosity Enhancement Device*, try to associate a business model with the project that can realistically turn all your R&D into a valid business expense and let you depreciate capital expenditures. If the coupling between the business and the project is close enough to stand up to the scrutiny of a bean counter, you not only save a ton of money but also create an aura of professionalism that will pay off again and again. Bankers, insurance companies, vendors, and even the trade press will take you much more seriously if that contraption you're building looks like part of a business instead of a "hobby."

Only *you* have to know the truth about the underlying motives... and won't you be surprised when the business angle is self-fulfilling and you really DO find yourself making a living from what you love?

Getting Educated

Once that tiresome money-making stuff is out of the way, we can start to play. One of my primary motives for taking on an increasingly complex series of technomadic projects has been to provide context and justification for learning curves—half the fun of this stuff is diving into something you know almost nothing about and becoming expert enough to do an end run around traditional approaches and make a contribution to the field. If you're driven by personal motives like *passion*, that can happen surprisingly quickly.

You are probably already aware of an important phenomenon: the nonlinearity of learning curves and their eventual asymptotic leveling. Since you're reading this, it's safe to assume that your brain is a rather AC-coupled affair, implying that some degree of change and growth is necessary to keep you interested. If you find yourself getting bored and restless in a job (or relationship, or town...) when the slope of the learning curve approaches zero, prompting you to migrate from one situation to the next just to prevent intellectual atrophy, then you already know *exactly* what I'm talking about.

So how do we arrange our lives to keep this from happening? Easy... just take on a massive project, then redefine it and add components as required over time to keep it interesting. A nice side effect of this is that if you play your cards right, you can keep yourself surrounded by experts who know a lot more than you do about various parts of the system, even while you become the world's leading guru on the thing you're building.

The Microship project is a perfect example. I had just spent a decade focusing my life on the bike, and for all its complexity and unfinished subsystems, it had become pretty stale in my mind. Few surprises lurked in the bicycle-touring lifestyle after 17,000 miles, and even though BEHEMOTH was intended to be a development platform, it was hard enough to hack that I didn't find myself rushing to build new bike subsystems nearly as much as I thought I would. The truth was that I was just burned out, so when I fell in love with tiny boats I was lured wide-eyed into a whole new world of unfamiliar knowledge: navigation, hull design, oceanography, harsh-environment packaging, composite materials, and more. I started reading again, finding and hanging out with experts whose language I could barely understand, slowly building a dream evocative enough to attract the people from whom I most needed to learn. This was profoundly energizing.

I subscribed to nautical magazines, loaded up on books, and hit trade shows. I posted questions to newsgroups and forums, built up a stable of advisors, and seized every opportunity to get on the water in boats of any size and configuration—learning every time. I played experts against each other by striking up 3-way email exchanges on diverse topics like HF antenna coupling to a rotating mast, participating in discussions with people who have spent years accumulating arcane wisdom far beyond my own, interjecting questions to keep things moving in the right direction. I was careful to establish engineering contacts within sponsoring companies, learning as much as possible about the constraints, problems, and unexplored potential inherent in their product design (stuff that's hard to get from marketroids). I even leveraged my bike-era notoriety to cozy up to authors and recognized gurus in these unfamiliar new fields, sometimes embarrassing myself with stupid questions but in the process building an incredibly potent database of experts—most of whom were remarkably patient with the early phases of my simultaneous learning curves because the *dream itself* had sparked their curiosity.

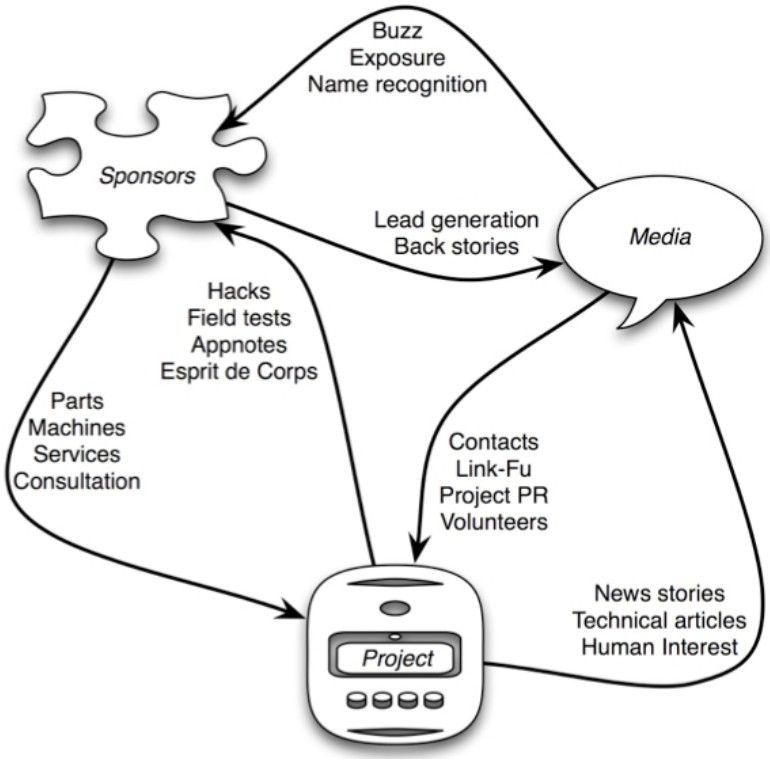
Although there is some risk in issuing "forward-looking statements," as corporate types like to say, I have also found considerable educational value in opening the design process to public scrutiny. I blog [U1] weekly about the ongoing development of my new ship (*Nomadness*), and am not at all shy about discussing plans… or admitting my uncertainty about how to address a problem.

Surprisingly often, this is all it takes to pull answers out of the ether. Somebody who knows way more than I do about the subject will submit a blog comment, send an email, or utter a quick tweet… basically giving me the benefit of their experience with the problem that has me stymied. Whole comment threads (and in some cases, friendships) have emerged from this, which would have been impossible had I waited until a design was a *fait accompli* before going public with the details.

It can be embarrassing, of course… a major hazard of life on the bleeding edge is that one is a newbie much of the time. But I will accept the occasional public cringe if it prevents a large-scale screwup. More than once, an idea that I just *knew* was brilliant got shot down by someone who had spent a whole career developing intuition about the subject. This was painful at the time, but I realized in retrospect that I was saved from pouring resources into any more dead ends than absolutely necessary.

The counter-argument, of course, is that often said experts are inculturated in old methods, and can't resist snorting in derision at the exuberant yammerings of a noob who doesn't even know what's impossible... but who ends up successfully breaking all the rules with a crazy design that nobody took seriously. You have to listen to even the most constructive criticism with a finely tuned discriminator, and not be afraid to stay the course if you're *really sure* you're on to something. Just don't let ego get in the way, especially early in the process when you aren't yet sure how much you don't know.

All those dynamics, coupled with IP-centric nervousness about revealing potentially valuable new ideas, conspire to enforce paranoid secrecy in the early stages of a project. But if you are serious about maximizing free education, try to publish early and often... it will attract teachers of every form.



Three-way symbiosis of project, sponsors, and media. Everybody wins as long as the process keeps moving forward, and it is up to you to drive that. New toys are inspiring; build something amazing with them, and the press will tell your story. If you craft interviews to honestly underscore the role of your sponsor(s) without blatant flag-flying, then you can keep the outer loop going indefinitely. The hard part is the leap of faith on all sides necessary to get it started. Inner, direct connections are just as important, and the most productive relationships contain all the elements shown here.

Sponsorship

This is a subject that gets everybody excited. Every few months, I receive email from somebody who has surfed across my list of hundreds of sponsors, basically asking: "How do I get companies to give *me* stuff for free?"

I've probably dashed a few hopes with terse responses that convey basic realities, the most important of which is rather obvious: the only realistic sponsorship deal is a win-win. Vague plans for media coverage are usually not enough to

convince people with budgets that they should send out free goodies, and some companies that make particularly alluring adventure tools receive so many proposals (on the order of dozens per week) that they have instituted formal procedures and tough filters; others have adopted a firm policy of only donating to nonprofit corporations.

Yet, there are times when it feels like Christmas every few days around here, with the UPS and Fedex guys dropping off new toys and a half-dozen new proposals always in the works. How do we do it?

Product versus Cash

First, let's get one thing straight. Asking companies for money is a *lot* harder than asking them for products, and if you plan to use the money to buy products anyway, you might as well eliminate those troublesome intermediate value states and their associated end-of-year implications. Besides, when you ask for money you end up dealing with the kinds of people who think in terms of return on investment, deliverables, tax laws, penalty clauses, and all that other weird stuff that causes the eyes of most geeks to glaze over. But when you go looking for goodies, you find yourself talking with engineers (sometimes even kindred spirits who wish *they* were doing something so cool), company principals (they may be suits, but they can make the instant decision to support you without having to sell the idea upstairs), and the aforementioned marketroids (hey, they may not always be the most creative folk, but they are usually friendly people who understand the value of good PR—and can be very helpful).

There is a counter-argument that applies in some situations: a big "title sponsor" who gives you a pile of dollars translates into only *one* relationship to maintain. Seeking a separate sponsorship deal for every component can gobble months of schmoozing time, and you can even find yourself having to delicately balance relationships with companies who see each other as competitors. This can get a little tricky, especially where IP or pre-release products are involved.

Between the bikes and the boats, we've had well over 200 corporate sponsors, and with only two exceptions they have all provided products or services, not dollars. The two who provided money did so in a very structured way—one as a consulting contract with a defined (albeit very liberal) deliverable, the other in the form of covering the lease on a lab building. No company has ever just handed us a lump sum of cash, although we've had a few individual donors.

Some projects (like alpine "first ascents") seem to need direct financial support to pay for people or logistics, but from my perspective that appears to be a much more stressful relationship, both in terms of the difficulty of finding major sponsors and providing a return that fulfills contract terms. This typically involves large-scale logo display, scheduled appearances, or serious product placement. The problem with all that is that it can really define your media image… and be difficult to change without stepping on toes if you suddenly find another brand you prefer (sports stars do this dance all the time). Best to sidestep the whole issue if you can.

In the early days of my bicycle travels, I was offered a significant monthly stipend—more than I had ever made in my life, although that's not as dramatic as it sounds—to pedal an epic South American journey while promoting a

particular brand of tobacco. As a militant anti-smoker, it was not difficult to politely decline, though I was terribly broke at the time and did timidly ask if they could launder it through their beer subsidiary (they were not amused). That was an easy decision, and it's a good thing: other than the obvious hypocrisy of promoting cigarettes on a bicycle tour, the one-shot hype experience would almost inevitably have prevented my nomadness from becoming self-supporting over the long haul.

More interestingly, however, it would also not have been fundamentally different from visibly identifying with, say, Apple or Motorola, companies that make things I actually *do* use. How would the media ever take me seriously, especially if I was talking about computers and communication tools? A big logo on the side of my machine would taint every statement with obvious bias.

This is the other key reason to try to avoid such relationships: as we shall see later, the media is a critical part of a three-way symbiosis with the project and the sponsors, and it wouldn't do to look too much like an advertisement. Obviously there is some product promotion going on, but fer chrissake, let's at least try to be *subtle* about it!

If we're not going to do logo decoupage all over the hull, what *do* we offer sponsors in return for their generosity?

Win-Win Relationships

If we think creatively, there are lots of things we can do to help a company that has just donated an essential component to a gonzo engineering project. Frequently, the sponsors are not high-profile purveyors of whizzy consumer goods, but are small, specialized companies that target their wares to industry. In these cases, a little publicity goes a long way, as they're generally not used to this sort of thing and tend to be more receptive than outfits that get a stack of proposals every day. This is a key point, so I'll say it again: don't beat your head against the gates of Microsoft, offering to trade your immortal soul by clicking through the EULA on a free piece of remote-admin spyware with auxiliary productivity features, when some small outfit making valves in Mississippi will fall all over itself for the opportunity to ship you parts in exchange for a little free ink in the trade press.

Media Exposure

"Ink," of course, is the first thing people think of when considering what to do for a sponsor: name mentions during interviews, product-application feature stories in industry trade journals, photos that show their gadgets being used in an unusual way, links on the project website. We pursue every opportunity to do all these things, and some companies with particularly visible or mission-critical components have received piles of media photocopies over the years—name mentions highlighted in yellow and a nice thank-you letter paper-clipped to the top.

But sometimes this doesn't work very well. Some products are in the class of development tools or are so deeply buried inside subassemblies that they couldn't be photographed effectively if we tried, and mentioning them in general media interviews would just confuse reporters ("This is the Microship, which uses stacks of DuPont Hytrel crane bumpers from Miner Elastomer as landing-gear shock absorbers.") Other than slipping plugs into obscure

publications like I just did [Editor: Hey!], how can I make sure that all sponsors' investments are worthwhile—increasing the odds that they will still be receptive if I need to hit them up for more goodies next year?

The first (and easiest) solution is to maintain enough of a public project narrative that everything is automatically included. Feature-length magazine articles are terribly restrictive in length and slant, and a recitation of vendors is boring to readers. But we have a blog, busy websites, and over 5,000 subscribers to an occasional emailed *Nomadness* posting, and as far as I know every sponsor has been mentioned at least once along with a paragraph or two about the product and a link. These are archived on our site, and are thus Googlable; there is also a sponsor page listing them all [U2], and for many years, entry to the Microship website triggered a Perl script that served up a random logo. The BEHEMOTH bicycle even had a sponsor-logo slideshow that ran on the console Macintosh during media events.

Still, that's not much exposure, although it is appreciated.

Ideally, I try to do a series of magazine articles in a variety of markets, highlighting each part of the system. This generates a bit of cash flow, while providing lots of opportunities to mention companies within the context of a trade journal targeted to their industry. It's harder than it sounds, however, as each one has to be pitched and sold to the editors, not to mention the problem of including enough of a general introduction to bring readers up to speed. In practice, this is not an effective way of making sure that every sponsor gets useful return for their donations, as only about half seem to lend themselves to this approach: cutting-edge newsworthy technologies, or highly visible components that show up clearly in photographs. Let's see what *else* we can do to make sponsors feel good about helping out with the project.

Enhancing Corporate Culture

Sometimes the decision to make "in-kind product donations" is so casual that companies don't really expect a lot of flashy media exposure in return... just coverage in their own publications. I have done dozens of interviews or provided photos for various sponsors' *house organs* (internal newsletters or magazines). A perfectly reasonable motive for supporting something exciting like this is to get the employees pumped about cool uses for their technology, and this actually can take a variety of forms:

I have camped for months in corporate facilities, building systems and deriving direct support from the employee population. (In such situations, I became a sort of high-tech court jester, adding a buzz of excitement to the routine 40-hour-a-week *blahs*.) I have conducted brown-bag luncheons for companies that donated equipment, giving energetic speeches that cover the entire project while emphasizing how their products were indispensable components and relating amusing anecdotes about integration or comments from the public. I've shown up at company picnics, sent regular updates to mailing lists or forums that repost to internal mailing lists, and become friends with the engineers who created the technology that I use. In every case, the net gain back to the sponsor is clear: their employee population gets to see their products appreciated outside the normal markets, part of something exciting and fun. Team spirit is powerful stuff, and has soft-dollar barter value.

Marketing Participation

Another angle is to participate directly in product marketing, not by independently getting media coverage but by offering your image for use in advertising. Personally, I tend not to do this too much as I don't want to become strongly associated with any single company, but those fears are probably unfounded—the few times I have done it worked out well.

One amusing variation on this that works particularly well with my fancy gadgetry is to appear as a guest in the sponsor's booth at a trade show. I've done this a lot, and it's good for all concerned: the company gets a kick-ass booth draw, the attendees get something way out of the ordinary with a bit of celebrity panache, and I get to hawk books and make contacts with other potential sponsors.

At a late-80s COMDEX in Las Vegas, I was exhibiting with Chips & Technologies. It was Day 4, and I was in serious burnout—the problem with these things is that person *n* arrives and asks a level-1 question, which you politely answer. Then person *n+1* arrives a few minutes later and jumps into the conversation that's just getting interesting... "Excuse me, what are you doing here? What is this thing?" After a few days of getting reset to zero every few minutes, you want to start punching people in the nose. I was in this approximate mood when a fellow arrived in the booth, looked at my bike, and exclaimed, "Wow, this is the coolest thing here!"

"Thanks," I said, handing him a flyer and sizing him up for a book sale.

"Yeah, but not for the reasons you think."

Oh crap. Another nutcase. I was steeling myself for the brain dump, religious testimony, or sales pitch when he bent down, dragged his finger along the frame, held it up to the light, wrinkled his nose, and said something that has stuck with me for decades:

"Look at this, it's filthy! It's the only thing at this show that I know is *real*."

Since then, I've never underestimated the value of putting a bizarre contraption in a sponsor's trade-show booth... it can mean a lot more than just another desperate theatrical attempt to get jaded attendees to stop and get their badges scanned for the mailing list. As long as the project genuinely uses the company's products and is not just hired to be a gratuitous booth draw in the same class as bikinis and magicians, it can be hugely effective.

FieldTesting

But hey, wait a minute. We're *geeks*. What's with all this marketing garf? Can't we return value to sponsors by doing something that's actually clever?

Well, I hope so. In the case of the Microship, we already have a whole infrastructure for data collection and the archiving of an arbitrary number of channels; this can be useful to some of the companies that have contributed products. Certainly it's not much trouble to record temperature cycling, humidity, and other environmental conditions—even PSD plots of the worst-case shock and vibration events. The big boys already put their stuff through its paces by subjecting samples to such abuses under much more controlled conditions, of course, but I've seen a few product specs that were compiled optimistically from the published data sheets of their components.

While wandering around outdoors for a few years hardly qualifies as a proper "accelerated life-test" program, it can certainly yield useful data... especially if the extremes happen to correlate with performance anomalies. We report failure modes, experience with adhesives or mounting problems, and anything else that may help forestall future problems.

Sharing the Hacks

There's another way geeks can do something for sponsors without posting logo GIFs or grimacing into cameras (though I actually kind of enjoy that sort of thing, in moderation). Why not return all the hacks, interfaces, and driver code that you had to conjure in order to put a particular widget to use—in the process, rewarding your sponsor with a sort of off-site skunkworks that is *unconstrained by internal politics*?

Sometimes this is irrelevant, but there have been times in these projects when we've been able to give our sponsors significant intellectual property—and since I'm not trying to market this stuff myself, it's not like there's any harm in sharing the results. Quite a few of the devices integrated into the ships have been removed from original packaging, given more efficient power supplies, associated with software objects and display widgets, and hacked to add interface hooks or enrich external monitoring. All this is fed freely back to the manufacturers, along with related code. Doing this has the satisfying flavor of closing the loop, and significantly enhances a sense of cooperation with the company—the result can be a very effective partnership in which you not only get ongoing gadget upgrades, but the expertise to make them dance.

Obviously, it's a bit tricky to generalize from computerized bicycles and micro-trimarans to all possible gonzo engineering projects, and some of this is irrelevant to, say, a software undertaking or epic data-collection exercise. But even tools built entirely of bits can be fair game for sponsor relationships, and hopefully we have shown how this can turn into a win-win. The only commentary I would add at this point is this: companies are made of *humans*, somebody there is going to remember that they made a donation, and employees want to look good to management. I can't emphasize enough the importance of keeping track of your supporting companies and contacts, making sure that you give them *something*, at least an occasional update, in return. It's not only likely to help you further down the road when the original version gets superseded by a new model, but it will make things easier for the next person who approaches them for help. More than once, I have been told: "Sorry, we tried sponsoring a couple of projects a while back, but it never paid off. Our policy now is to just offer a discount."

Next up: The Media Dance.

About the Author

Steven Roberts was the original "technomad," covering 17,000 miles around the US on a computerized recumbent bicycle from 1983-1991 while publishing tales via CompuServe and GEnie, then extending the same design objectives to water with an amphibian pedal/solar/sail micro-trimaran that consumed all available resources until 2002. As is typical of homebuilt boat projects, however, by the time it was finished he didn't really want to do that anymore... so he has since made the transition to a full-time life aboard a 44-foot steel pilothouse sailboat, and is now based in the San Juan Islands north of Seattle.

The ship is extensively networked with embedded data collection and control systems, streaming telemetry, and a user-interface layer reminiscent of the Enterprise... with a wrap-around console that includes communications, R&D lab, audio production studio, and a piano. Roberts has

published 6 books ranging from travel and adventure to microprocessor design, and prior to becoming a technomad spent a decade developing custom industrial control systems, early home computers, and other paleo-geekery. More on his technomadic projects can be found at microship.com [U3] (with the new boat at nomadness.com [U4]). He is publishing the ongoing technical narrative of the new project as a monthly PDF "Nomadness Report," as well as a series of Boat Hacking design packages detailing the subsystems.

Send the author your feedback [U5] or discuss the article in the magazine forum [U6].

External resources referenced in this article:

[U1] http://nomadness.com/blog

[U2] http://microship.com/sponsor

[U3] http://microship.com

[U4] http://nomadness.com

[U5] mailto:wordy@microship.com?subject=PragPub article

[U6] http://forums.pragprog.com/forums/134

Finding the Geek Who Fits

Five Tips for Hiring as an Agile Team

by Johanna Rothman

The team will have to work with the new hire. Shouldn't the team do the hiring?

Imagine this scenario. You've transitioned to agile. You are past the first few months, and your team has settled into an even cadence. You manage your work in progress, whether you work in flow or iterations. Your features actually get to *done* and your product owner or customer is happy. You are managing your pre-agile technical debt.

Now, comes a shock to your happy team. David, one of your developers, is leaving. He's not unhappy. But his wife has a unique opportunity with her job. They are transferring to the other coast. What do you do now? You don't want to hire just anyone.

Tip #1: Start with Identifying Your Culture

Every team—agile or not—has a unique culture. Your first job is to identify your unique culture. To do that, first understand what culture is. Culture is made up of these three components:

- What people can discuss
- How people treat each other
- What is rewarded

There is nothing right or wrong about culture. It just is. And while agile teams might be similar, every single agile team has a unique culture. Because every single team has their own stamp on how they make decisions, even in the context of the larger organization.

How do you discuss design and architectural decisions? That's part of your culture. How do you discuss the issues of context switching or whether you should do more features or address the legacy technical debt? That's part of your culture. Is your product owner/customer stronger or weaker than others in the organization? Do you have an agile project manager or Scrum Master? That's part of your culture.

Do you have team rewards or individual rewards? I know, I'm spitting into the wind here, talking about team rewards. So few of you have team rewards. But at some point, more of you will have them.

Once you have identified your culture, you can say, "Here is our culture. This is what we want someone to fit into. We don't need a perfect fit. We need someone who's not too far off."

Remember, we don't often fire people because they don't perform well. We fire people because they don't fit our team norms.

Tip #2: Analyze the Job as a Team

It would be easy to use a generic job description that you have, or the job description from three years ago. But, I guarantee you that any job description from your pre-agile days is no longer useful. And any job description you haven't updated in the past few months is suspect. So analyze the job. I have templates [U1] that can help you start.

When you analyze the job, you start with the person's interactions and roles. In effect, you start with a user story for this person. The value this person brings is in his or her activities and deliverables. So, you want to consider the span of responsibility.

Do you need someone who can coach more junior members of the team? Or someone who is coachable? Do you need someone who can help the team think more strategically about where the product is going? Or someone who is happy to keep implementing by feature and refactoring to patterns? How about someone who wants to keep the testers happy? It really depends on your context, doesn't it?

One of the big pieces of analyzing the job is to be able to differentiate the essential from the desirable in the analysis. If you don't do that, you will end up looking for a person you cannot afford.

Tip #3: Review Résumés as a Team

If you are large enough to have an HR department, you probably have an Applicant Tracking System, an ATS. Your HR department would love it if you used the ATS to review résumés and make decisions in the ATS, right? Don't do it.

Use the ATS as a way to gather résumés. The ATS is an abomination upon the earth.

The ATS has made people exaggerate experience, load their résumés with useless keywords, and made it impossible to review résumés. Ask the hiring manager perform an initial screen of the resumes, throwing out the impossible resumes. Instead of using the ATS to review, print the résumés that remain. Instead of using the ATS, print the résumés. Yes. On paper. One copy for everyone in the room.

While you are recruiting, at the same time every morning, maybe right after your standups, have everyone read the résumés you received the day before. Have everyone sort the résumés into three piles for phone-screens: Yes, Maybe, and No.

When everyone is done reviewing the résumés, discuss who said Yes to which résumés, who said No, and who said Maybe, and why. The *why* is most important. After a couple of weeks of discussing why, you will all understand what you are looking for in a candidate. This is why it's so important to work together and iterate on the analysis after you've phone-screened and interviewed some candidates.

Tip #4: Prepare to Interview as a Team

Just as software is a team activity, interviewing is a team activity. High performing software development teams are, for lack of a better word, intimate teams. You don't want just anyone to join you. You need to know you can stand working with these people for hours every day, week in, week out.

Eliminate the Dirt-Bags

I don't like working with rude people. So I organize the dirt-bag elimination phone-screen. That's the phone screen where the HR person calls the candidate to set up the real phone screen. Is the candidate nice enough to the HR person? Or is the candidate rude? If the candidate can't be bothered to be nice enough to the HR person to set up a real phone screen, that candidate gets crossed off the list.

That's part of my culture, and might not be part of yours.

Assign a Primary Phone-Screen Interviewer

I do find it useful to use a primary phone screen interviewer, so only one person has to coordinate all the phone screens. Since I recommend you use a phone screen script and write down answers in a phone screen, I also recommend you use a manager for this.

Often, the manager will have the responsibility for new-hire paperwork, for extending the offer, for taking the lead on the hiring. The coordination may be easier for you if you do it this way.

Use an Interview Matrix

Know who is asking which questions. Everyone should take two areas to ask questions about. In a future article, I'll provide you tips about questions to ask and questions not to ask. I like to use a matrix that includes the time of the interview and which areas each interviewer will ask questions about. I have a template [U2] for that, too.

Include an audition that fits your context. If you pair, by all means, pair with the candidate. But, if you don't pair, please, don't pair. That sets up an inaccurate expectation. Auditions are a huge topic. I've written more about them in Hiring Geeks That Fit, and have much more on my blog [U3].

After a great interview, you want to make a decision as a team.

Tip #5: Decide as a Team

Deciding as a team, in a follow-up meeting, is a form of limited consensus. Since you will all have to work with this candidate, it makes sense for you to decide as a group whether you want this candidate. Not all managers understand how to use limited consensus, so here's a way to make it work.

In a perfect world, no one would be brash and opinionated. Have you met me in person yet? What I lack in size, I make up for in volume. I can argue with anyone and often win—about just about anything. But we don't want the loudest person winning when it comes to candidates. We want the person who will fit our team the best. That's why we went through the culture activity first, so we could use it in the analysis and apply it to our résumé review and question development, and use it as a check on our decision-making.

In the follow-up meeting, we have several ways to see what people think. My first approach is to use thumbs: Everyone keeps their mouths shut, and uses thumb up to say, "Hire!" or thumb down to say, "No way!" or thumb sideways to say, "I'll go along with the group."

The key is for people to thumb vote while not saying anything. Once people have voted, now we can discuss and learn the reasons behind the thumbs.

Want More Tips?

These tips just skimmed the surface of hiring. If you want to read more, please visit my Hiring Technical People blog [U4]. And of course I'd love it if you'd buy my book Hiring Geeks That Fit [U5]. Stay tuned for my next article about questions to ask and not to ask. And yes, I'll tackle the reasons why you should not ask why or how to move Mt Fuji. Seriously.

About the Author

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 [U6]. Please do go there, wander around and subscribe!

Send the author your feedback [U7] or discuss the article in the magazine forum [U8].

External resources referenced in this article:

[U1] http://media.pragprog.com/refer/pragpub45/book/jrgeeks/templates.pdf

[U2] http://www.jrothman.com/wp-content/uploads/2011/11/HiringGeeksThatFitTemplates.pdf

[U3] http://www.jrothman.com/blog/htp/tag/audition

[U4] http://www.jrothman.com/blog/htp

[U5] http://pragprog.com/refer/pragpub45/book/jrgeeks/hiring-geeks-that-fit

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

[U7] mailto:michael@pragprog.com?subject=hiring

[U8] http://forums.pragprog.com/forums/134

Being the Geek Who Fits

Don't Forget That You're Interviewing Them, Too

by Andy Lester

In a counterpoint to Johanna Rothman's article on finding the geek who fits your agile team, Andy looks at the situation from the prospective hire's point of view.

You've been through a couple of job interviews on this job that looks cool, and they just extended an offer. The money's good, so you say "yes," right? Maybe not.

Say you go on a blind date, and things seem to go well. The two of you have dinner and she asks you all sorts of things to find out what kind of person you are, and your plans for the future, stuff like that. Then you go on another date, but this time she's brought her brother and her parents. The four of them bombard you with more questions about your life. Then, after a couple of hours, she says "I've decided that you're the fella for me, so let's get married." Would you say yes, right then and there? You'd be crazy to, right?

Now, think about how often people accept job offers based on a few hours of having been grilled and then being told that they were deemed adequate.

Accepting a job with an employer is the start of a long relationship. This is a relationship where you'll spend more hours at the office than you will spend hours awake with a spouse. You need to make sure the job is right for you.

Doing your work is only half of your life on the job. Just as your boss judges you on your work and how well you work with others, you should judge the company by your work and how well the company works with you. You need to assess the company culture.

Company culture isn't about the people, but people are part of it. For example, if someone you meet in the interview process is a jerk, that may make you not want to work there. That might not be a good criterion, however, because people come and go over time, and that jerk could go get another job next week. Culture is more consistent. In this case, the problem is not the jerk, but that the company culture tolerates jerks.

Much of your understanding of company culture will come from your conversations in the job interview. Remember that the interview is not an interrogation where the candidate is asked questions and remains silent otherwise. It's a conversation between potential colleagues discussing the business needs of the organization and how the candidate can fill those needs, as well as how the candidate will fit with the organization. Both employee and organization must fit with each other or the relationship is likely to be a failure.

Don't set out to assess culture at the beginning of the interviewing process. You should save your detailed questions until after you've received a job offer. Most of your time and energy in the interviewing process should be spent working toward getting an offer. If you fail to get an offer, then it doesn't matter what the company's culture is like.

Once you have an offer, though, the balance of power tips in your favor, and it's time to investigate culture in detail. Ask questions, meet with people, and observe as much as you can.

Ask Questions

Here are some questions that you may want to find the answers to as you get to know a company better. Note that the best way to find these answers might not be a direct question. An interviewer won't determine if you're a hard worker by asking "Are you a hard worker," because the answer will of course be "Yes." He'll for stories that illustrate your work ethic. Similarly, if you want to know about how the company handles crisis, ask about the last time a project went bad. You may want to ask multiple people that question, not just your hiring manager.

In these questions, I use the word *group* to mean an individual workgroup in a department, but these questions may apply to the department or company level as well. Also note that none of the questions are meant to imply that one way of working is better than another. For example, a tightly-knit team is not better or worse than a group of independent workers except as *you* judge it to be a culture that you want to be part of.

Work Life Questions

The first things to consider regarding culture are questions about how things work, about what the company values, about what day-to-day life is like.

- How tightly knit is the group? How much of the work is done as a team, and how much is on your own?
- What are the working hours? How often is it necessary to work outside those normal working hours? Is that extra time seen as just part of the job, or is it seen as effort above and beyond?
- What does the group value in others? What does your new boss value in the group, or the department? What does the company value in its employees?
- Are failed projects part of life? Or are they a firing offense?
- How does the team/department/company handle crisis?
- What is intra-company communication like? Are there strict funnels of information, or are employees encouraged to talk directly with other departments?
- Is the group open to change and new technologies? Or do they prefer the stability of what's known to work?

Relationships Questions

The other part is even more nebulous, thinking about how people get along and what the relationships are like. (Make no mistake, you have a relationship with every single person you interact with, even if you think you're just a heads-down kind of coder. Even direct avoidance of relationships with others is itself a relationship.)

- What are the relationships like in the group? Do people seem to like each other, or are relationships strictly business? Are people friends outside of the group, or are work and home relationships separate?
- Is there a group dynamic that extends beyond the work to be done? Does the group eat lunch together? Is there a regular outing for beer on Friday nights? Weekend trips to the rock climbing wall? Will you be the odd man out if you don't go climbing with them?
- What is turnover like in the department? How often does the team change? Will you be coming in to an established team that has been together for years?
- Are there leaders in the group, either defined or *de facto*? Or is everyone an equal?
- Do team members critique each other's work? Or does everyone do things their own way?

Company Life

It's also important to understand the culture of the company as a whole, and how the group you'll be joining fits into it.

- How is your performance assessed? Is your performance assessment based entirely on you, or does it include an assessment of the department as a whole, or the company as a whole?
- Are employees rewarded beyond a paycheck? How? Pizza party? Cadillac Eldorado? A set of steak knives? If there are bonuses, what are they dependent on? Your team's performance? The entire company's?
- How is the team perceived by the rest of the department? How is the department perceived by the rest of the company?

Again, these questions aren't a checklist to take with you on an interview. They're a starting point of questions for you to consider as you investigate the company you're looking to work for.

Other Ways to Observe Culture

There are other ways to get a sense of company culture that are less direct but can sometimes be even more informative. One great way is to look at the company's employee manual. Ask the hiring manager to see it. Take the time to read through it. Is it a thick binder filled with regulations as specific as saying where in your cubicle you're allowed to hang pictures? Or, on the other extreme, do they not even have an attendance policy and say "so long as you get your work done, it's all good?"

Is there a culture of recognition? I have an email folder called "Head Pats" where I accumulate everything nice said about me or my team from someone in the company. Many of these are from upper management thanking my group for a successful project. Does the hiring manager have such an archive? Ask if you can see some recent examples.

Ask for a tour of the offices in the middle of a workday. What do you see? Are people happy? Are they heads-down in work, or discussing things as a team?

Is it a cube farm, or are there offices? Are there public spaces for collaboration? Is it sterile and buttoned-down, or are people shooting Nerf guns?

Don't just look at the group you'd be working with. Take a look at, say, the accounting department, or customer service. Do they look happy to be there?

Finally, look around the Net for indications. Start with the company's own website and blog. For example, even a cursory read of GitHub's blog makes it clear that much of the culture revolves around social drinking. Maybe you can find blogs from current or past employees that can give clues. Search LinkedIn for those who have worked at the company. Maybe from surfing LinkedIn profiles you'll find that people who work there leave within a year.

One final note: No matter what you've learned about the culture and how awesome the job looks, never accept a job offer on the spot. Wait at least 24 hours to make sure that it's what's best for you. You're psyched, you're ready to go, and that's not the best frame of mind to make the final evaluation. Better to get away from the situation and look at it from a distance. If you have a significant other in your life, it's all the more crucial.

When you get the offer, say "Thanks very much, I'm glad we're at this stage. Of course, I'll need some time to examine the offer. When would be a good time to talk to you tomorrow?" Don't worry that they'll back out on you. Any reasonable company is going to understand. If they pressure you into a decision right then and there, that's a warning flag.

About the Author

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 [U1] 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 [U2], and can be reached by email at andy@petdance.com.

Send the author your feedback [U3] or discuss the article in the magazine forum [U4].

External resources referenced in this article:

[U1] http://www.pragprog.com/refer/pragpub45/book/algh/land-the-tech-job-you-love

[U2] http://petdance.com

[U3] mailto:michael@pragprog.com?subject=interviewing

[U4] http://forums.pragprog.com/forums/134

Uncle Bob and Functional Programming

A Reply to "Uncle Bob" Martin's "FP Basics Episode 3"

by Paul Callaghan

Paul has been debating functional programming and test-driven development with "Uncle Bob" Martin. Here he shares the result of that conversation.

In January, we published an engaging essay by "Uncle Bob" Martin on functional programming (or FP). Bob has since continued his exploration of FP on his blog [U1], and we recommend it to anyone interested in the functional paradigm. One person who has been following Bob's series is Paul Callaghan, who has been writing a series on FP for us. This led to Bob and Paul having a spirited and productive back-and-forth about FP, and that led to Paul's current article, in which he explores some of the points raised in a recent post in Bob's series. –ms

Taking the Uncle Bob Challenge

Episode 3 of Bob Martin's series [U2] on Functional Programming Basics is a very readable investigation into use of TDD ideas in FP and how TDD-grown code compares with more "idiomatic" functional code. It raises several interesting points, and what you are reading now attempts to address some of these and carry them forward. It also contains some Haskell.

The starting point is a small programming exercise: formatting a string into block of lines according to certain rules. Bob's description of the problem took the form of an informal overview, a motivating example and some test cases. Here is the example, using a maximum line width of 10.

```
# input string
inp = "Four score and seven years ago our fathers brought " +
      "forth upon this continent a new nation conceived in " +
      "liberty and dedicated to the proposition that all men " +
      "are created equal"
# resulting output
out = "Four score\nand seven\nyears ago\nour\nfathers\n" +
       "brought\nforth upon\nthis\ncontinent\na new\n" +
       "nation\nconceived\nin liberty\nand\ndedicated\n" +
       "to the\npropositio\nn that all\n" +
      "men are\ncreated\nequal"
# start of the printed output
Four score
and seven
years ago
our
fathers
...
```

The test cases from Bob's article are below, expressed as a list of assertions on some input function `wrap` to be tested (I'll want to test several versions).

```
tests wrap
 = [ wrap 1 ""    == ""
   , wrap 1 "x"   == "x"
   , wrap 1 "xx"  == "x\nx"
   , wrap 1 "xxx" == "x\nx\nx"
   , wrap 1 "x x" == "x\nx"
   , wrap 2 "x x" == "x\nx"
   ]
```

We'll explore various ideas about coding techniques, comprehension, testing, verification, and validation. A key notion for me throughout is *confidence*. Whatever we do while developing code to solve some problem is about improving confidence that it's the "right" code (whatever that means). Testing provides one kind of confidence. Code reviews give another kind. Various proof-related techniques yet another. It's always debatable how much confidence any particular technique can give, though we can consider what kind or level of confidence we want in some situation, and select appropriate methods.

Missing a Trick

Before we launch into functional programming, here's a solution that is arguably better than the others we're considering. Why? Because it uses a domain-specific language (DSL) specifically designed for this kind of problem, and it's enviably short! What more could we want?

```
# this is perl!
# $n is the max line length
s/(\w{$n})(?=\w)/$1\n/g;
s/([\w ]{1,$n}) /$1\n/g;
```

The idea is to use regular expressions, matching relevant patterns and replacing them with certain strings.

Regexps allow us to say *exactly* which patterns we want to match, and the replacement facility provides an easy way to modify the string.

The first substitution expression splits all long words down to size, adding a newline after each N-character chunk—as long as there's a word character following the potential break. (The `(?=\w)` is Perl syntax for a lookahead assertion, which tests for an expression but doesn't consume it—that is, the match still ends where the lookahead starts.) The `g` suffix means it is applied throughout the string, with the next match attempt starting after the end of the previous match (not including the lookahead), hence the overall effect is to split all applicable words.

The second expression handles the line-packing functionality, matching between 1 and N word or space characters *followed by a space*, and then changing the final space into a newline. Again, it's applied globally so is applied as many times as possible, starting from after the previous match. Note that regexp matching (in most implementations) is "greedy," so in the second case it will always try to match the longest sequence (up to N characters) before splitting a line.

This is great until we reach the last line, where greediness means we break too early (because there's no space at the end of the string so it backtracks to the last space seen). One sneaky way around this is to add a space to the string,

do the matching, and then remove the final character. Maybe it's a bit too sneaky though.

```
$_ .= " ";
# now do the substitutions as above
chop;
```

Here's another idea for you to try: we can code this up as a state machine, consuming one character at a time and deciding to emit zero or more character(s) on each step. This could start the wrapping process while you were still feeding in text, even if the text was huge or potentially infinite. This version should be achievable with one of the Unix lex variants, or a simple state machine library in your favorite language. (Incidentally, this "convert as we go" functionality comes for free in Haskell because of the laziness. Check out the interact function too.)

We can run the tests and observe a set of passes. But questions to leave hanging: are we sure it's right, and how are we sure it's right? And how sure are we?

Bob's TDD-Grown Solution

Below is the code Bob derived using a TDD process, translated into Haskell. I'm including it for interest, and for comparison with the other Haskell versions.

I'll stick my neck out and suggest that the Haskell version, with its more "modern" syntax, is easier to read and to follow. Take a look and decide for yourself. But it's not the key point of this article, so let's save that discussion for another time. Plus, the ideas covered in this article apply in most other FP languages too, not just Haskell, so this isn't purely an ad for one language.

Notice that I'm also relying on Haskell's version of strings as a list of characters, rather than a Java-ish array of characters. This allows me a bit more flexibility by using basic list functions like take and drop instead of substr. Does it change the nature of the exercise though? I think not, because both lists and arrays are sequences and the basic operations are fairly similar. However, it does maybe encourage me to split a long string into words sooner, that is, to work with more convenient forms of the data—and to make some further simplifications.

```
wrap s n
 | length s <= n = s
 | otherwise     = let (start,end) = find_start_and_end s n
                       head = take end s
                       tail = drop start s
                   in head ++ "\n" ++ wrap tail n
find_start_and_end s n
  = (line_start, line_end)
    where
      space_before_end = last_index_in ' ' n
      line_end         | space_before_end < 0 = n
                       | otherwise            = space_before_end
      trailing_space   = ' ' == s !! line_end
      line_start       | trailing_space = 1 + line_end
                       | otherwise      = line_end
```

This code passes all of the tests, and that was the original goal. You can see how it works—basically looking for the last space within N characters and determining the end point for the current line and the start point for the

following line. Next, one would normally refactor, but it's not too clear what refactorings should apply. From a "smells" viewpoint, there are a few points that would be good to address. I'll let you decide for yourselves what these are.

My initial reaction though, as a somewhat seasoned functional programmer with a fetish for data structures, was "it shouldn't be this complex!" The rest of the article will explore this reaction.

Last detail here: Bob's Clojure code calls a method from Java's String class to find the last index of a character in a string (optionally up to a certain maximum position). Here's one way to do it in Haskell. find_last reverses the input string and then looks for the first occurrence of the character. Since we are dealing with plain lists, though, it doesn't have to be a character—this operation is OK with a list of anything as long as we can test for equality.

```
find_last :: Eq a => c -> [c] -> Int
find_last c s
 = case elemIndex c (reverse s) of
     Nothing -> -1
     Just i  -> length s - 1 - i
```

The maximum position aspect is handled by trimming the string argument before the call to find_last, remembering to add 1 because we're converting from a position to a length. (And yes, that's a maximum *position* not a maximum length—it caught me out too. So much easier for lists to talk offsets and lengths, rather than start and end points! That is, easier to split a string into the actual pieces you want, rather than work out the indices to target the corresponding slice.)

```
last_index_in :: Eq a => c -> Int -> [c] -> Int
last_index_in c n s
  = find_last c (take (n+1) s)
```

Some TDD in Haskell

Let's try TDD-ing directly in Haskell, just to see where we get. There are several XUnit adaptations available for Haskell, plus more advanced tools like QuickCheck [U3] (or see RushCheck [U4] for a Ruby port), but we don't need anything sophisticated, and rolling my own version below allows coverage of a few more programming ideas.

We'll write the tests as a list of boolean-valued assertions, where each case indicates what is expected from each input string. The test list below takes a function to test as a parameter because we'll want to play with different versions. So, assuming the code above is defined as wrap_bm, then evaluating tests (\n s -> wrap_bm s n) will perform the assembled tests on the translation of Bob's TDD-grown code. (The lambda handles the difference in argument order. I could also use Haskell's flip function for this.) The outcome will be a list of booleans, of course.

```
tests foo
  = [ foo 1 ""   == ""
    , foo 1 "x"  == "x"
    , foo 1 "xx" == "x\nx"
    , foo 1 "xxx" == "x\nx\nx"
    , foo 1 "x x" == "x\nx"
    , foo 2 "x x" == "x\nx"          -- Bob's six tests end here
    , foo 3 "x xxxx" == "x\nxxx\nx"  -- A new test case
    ]
```

It helps to dress this up a bit, so why not number the tests and show "ok" or "not" depending on each test result. The following is something I will rerun at each step of TDD-ing the Haskell code directly, passing in the latest version of the function to test. Literally, it generates a list of raw booleans by applying the tests to the given function, converts each to "ok" or "not" depending on the test result, then zips them with a list of numbers (so pairing 1 with the first, 2 with the second, ... until either side runs out of values) and creates an appropriate string from each result. This is a list of strings, but we want them printed out one to a line and the combination putStr (unlines STUFF) does this in Haskell. As a style thing, note the use of . to compose functions and the general lack of parentheses, which leaves the overall operation looking like a pipeline of simple steps. Which it is...

```
check = putStrLn
      . unlines
      . zipWith (\n s -> show n ++ " " ++ s) [1..]
      . map (\t -> if t then "ok" else "not")
      . tests
```

To cut a long story short, here's some code that passes all seven tests. You can see the progression of versions on this github gist [U5]. Seven tests? Well, I found another test case (foo 3 "x xxxx" == "x\nxxx\nx") that failed on the code I'd reached after the first six tests. Here's my code after seven passes. There's at least one other case where this version fails. See if you can find one!

```
wrap_7 n "" = ""
wrap_7 n (' ':x) = wrap_7 n x
wrap_7 n x
| null rest
= line
| otherwise
= case break (== ' ') $ reverse line of
    (no_space, []) -> line ++ "\n" ++ wrap_7 n rest
    (tl, ' ':hd)   -> reverse hd ++ "\n"
                          ++ wrap_7 n (reverse tl ++ rest)
  where (line,rest) = splitAt n x
```

It was an interesting—and for me a slightly discomforting—experience. I did do it blind, several days after seeing Bob's article and not looking at his answer. It was tempting to do too much on each step, like inadvertently using parts of my preferred solution.

The key idea above is to take slices of N at a time and decide how to handle them based on whether there's a space near the end of the slice or not. Haskell's splitAt function has type Int -> [a] -> ([a], [a]), so splits a list of anything into two pieces with the first piece being length N or less, and the second piece being the remainder of the string.

The function `break :: (a -> Bool) -> [a] -> ([a], [a])` is similar, where the first part of the result is the prefix (beginning) of the list that failed the test and the second part is the rest of the list— starting with this first hit. So, `break (== ' ') "foo bar"` gives `("foo", " bar")`. So, `break` can be used to find a space (or not). I'm interested in the *last space* so I reverse the current line chunk before hunting for it.

I'm using a bit of pattern matching to determine which case I have, namely (a) no space found—which makes the post-break list empty, so the result should be the current slice then a newline then the result of wrapping the rest; else (b) there was a space, and the result is the before-space chunk then a newline and then the result of wrapping the remaining text. Because the after-space text is unconsumed, we add it to the front of the text to process.

One big difference between this code and Bob's code is how it's using data structures a lot more, splitting a string into pieces and manipulating them rather than juggling with string indexes. As mentioned before, I think this is a key aspect of modern functional programming—using the rich data structures and their tools to help simplify our programs. It does help to avoid certain classes of error.

Do you have to memorize the language's libraries to do this, though? I hope not. My claim is that if you start to *think in terms of data structures*, and think about the transformations, then you'll start to realize what you might need and have a clearer idea of what to look for. To be more concrete, if you think about splitting a long string into pieces, then you can start asking, does it feel like an obvious library function, or does it feel like something that should be decomposed into smaller pieces? Even if you can't find something, the smaller pieces tend to be simple enough for you to write yourself. For example, `splitAt n xs` can be defined as `(take n xs, drop n xs)`.

To consider that point another way: I don't think it's about memory, but more about being able to think through a problem and having some confidence in your thoughts. So, if your ideas make sense then have faith to chip away at the hard bits to get to the easy bits. (This point reminds me of many former students who took "Computer said no" too personally and didn't try to work out why the computer rejected their code, often because they didn't trust their instincts enough, so instead tried to add more complexity…)

Anyway, is this good code? Depends what "good" means. It passes all of the tests, which says something, but are we sure that those tests are enough? It's debatable. I made a mistake in the sixth iteration—missed a call to `reverse`—which the original six tests didn't catch. My new test was on `wrap 3 "x xxxx"`, though perhaps using a mixture of letters in the original tests might have worked too, by explicitly excluding any permutation of the input. There's at least one more test case that fails. I am still not sure whether further test cases are needed.

Anyway, there are several functional smells in this code, like why the strange case for `wrap_7 n (' ':x)`, and the `null rest` case feeling like a premature end case. Can these be refactored out, and some better structure recovered? It's not obvious how. But most significantly, it's also not too clear *how* the code works, particularly in the sense of having confidence that it does what we intended

(other than by tests). Can we do it in a transformational style, as a pipeline of simpler stages? That's the next thing to discuss.

Thinking About It

The old me thinks he sees a pattern. It looks like a splitting and packing problem, and that kind of thing can be done in a pipeline. (Actually, a lot of computation can be done as a pipeline! And if it can't, then why not? Or why not as a pipeline plus some monadic wrapping?) Intuitively, we can split the input into words, then split the long words into a list of chunks no longer than N and concatenate the results into a single list of word fragments. Next, we assemble this list of fragments into a list of lines, where we pack as many words into a line as possible without exceeding the length limit. Then, we just join the lines together.

Why not try to visualize this? For example, draw a picture for how the input data gradually gets transformed, perhaps adding some concrete examples to illustrate finer detail or help understand the abstract view. This technique really helps understanding, so I do recommend it. (I used to have great fun in lectures, drawing such diagrams out on large blackboards 30 feet wide, then in a cloud of chalk dust gradually converting sections of the diagram to straightforward code. Whiteboards just aren't the same!)

Let's write this down as code. For "obvious" things, we'll write down the usual library functions. For non-obvious things, we'll put in a placeholder to come back to later.

```
wrap n i = intercalate "\n"
         $ makeUpTo n
         $ concat
         $ map (chunksOf n)
         $ words i
```

This clearly makes sense, and we can square it with our visualization, that is, it says what we were thinking. Next, fill in the gaps, working on one piece at a time. The first task is to split long words into the smaller chunks. Considering the input and output, we can expect `chunksOf` to have type `Int -> [a] -> [[a]]`, so given a size limit it converts a list of things into a list of lists of such things, for example a long word into pieces of word. There is much that the type doesn't say—but we'll return to that later.

Hang on! Is this defined in a library or package (that is, a standard library or an optional gem-like entity)? It seems like something that is pretty useful, so let's search a bit. Turns out yes: a package called `split` provides a module called `Data.List.Split` that contains function chunksOf [U6] One way to find such functions is just to use a google search, for example haskell library split list [U7] and look through the various library pages from or use Haskell's specialist search tool Hoogle [U8]—although the current version doesn't seem to index the package we found.

You don't have to use this package (and as with ruby gems you should check that the code is OK before you import it—and that it's from a secure server(!)), or you could just copy the definition into your own code—with a reference to its source, of course. The package definition is however slightly optimized so let's see a simpler definition. First, `iterate (drop n)` repeatedly removes N

elements from a list, returning a list of the diminishing results, for example, iterate (drop 3) [1..10] will give [[1..10], [4..10], [7..10], [10..10], [], [], ...]—and empty lists until infinity. The takeWhile only keeps passing results from iterate while the result is not empty—hence stopping when it sees the first empty list and so also avoiding the infinite part. Finally, it takes the first N items from each of the iterate results to end up with the required result.

```
chunksOf n
 = map (take n)
 . takeWhile (not . null)
 . iterate (drop n)
```

This pattern is sometimes called an "unfold," since it is kind of a fold in reverse— useful for converting some value into a list of results. Hence chunksOf n = unfold (take n) (not . null) (drop n). As an exercise for you, see if you can use unfold to convert an integer to its string representation by repeatedly dividing by 10.

```
unfold :: (a -> b) -> (a -> Bool) -> (a -> a) -> a -> [b]
unfold h p t = map h . takeWhile p . iterate t
```

The tricky bit is packing words into a line. It's not immediately obvious, but one good technique is to use an "accumulator" to build up the result before emitting it when some condition is met. This is a standard pattern in FP, and often the first thing to try when the obvious things don't seem to be relevant. (As another exercise, try to write map in an accumulator style and then compare to the standard definition. You agree which is simpler?)

The real work of packing is done with a local function mlu hidden inside the main function. Line 1 says, when there are no more words to process, return what's in the accumulator. When the accumulator is empty, line 2 puts the next word into it. Line 3 tests whether the next word can be added to the accumulator to still give a string less than the target length, and if so, it goes on to process the remaining words with a new accumulator. The local definition on line 5 just names the expression pre ++ " " ++ w so we can use it twice without repeating ourselves. Line 4 handles the case when the accumulator can't accept the next word, and causes the accumulator to be emitted and then the recursive call starts the next line with an empty accumulator.

```
makeUpTo :: Int -> [String] -> [String]
makeUpTo n = mlu []
 where
  mlu pre []     = [pre]                                    -- 1
  mlu []  (w:ws) = mlu w ws                                 -- 2
  mlu pre (w:ws) | length new_pre <= n = mlu new_pre ws     -- 3
                 | otherwise           = pre : mlu [] (w:ws) -- 4
                   where new_pre = pre ++ " " ++ w          -- 5
```

We've written this as a function for packing strings, but inserting a space is the only detail that ties it to strings. It can be used for packing lists of anything if we pass in the separator element as a parameter, which will give us a useful function that can be used in other contexts (subject to certain assumptions or restrictions).

Let's recap the main points. The key thing in my view is to think about the data being manipulated and to split complex operations into simpler pieces. It helps if you know the libraries, but I contend that being more aware of the

shape of the data at each stage makes it easier to identify missing pieces and to have some idea what to look for— or whether you might need to fill in those pieces with new code. Put another way, it's about the confidence to think about the higher level and leave the lower level details for later.

What about testing? What do we think we need? At this point, I feel fairly confident that the code is OK. (Testing in the context of *maintenance* is a different issue, and will be discussed later on.) My intuitive understanding of the spec makes sense and there are no awkward corners that stand out for special treatment. The code seems to follow fairly naturally from this intuitive understanding, and the bit that seemed complex (line packing) is safely isolated in a short function which itself makes pretty good sense. Plus, this short and independent function could be further analyzed or tested quite easily if I wanted more confidence.

Now, is this naive? Or over-confident? Another smug functional programmer? Possibly. I think a little bit of smugness is allowable, though (but staying mindful of the dangers of too much confidence). Generally, functional programmers do understand how to use the language to sidestep obstacles and to break big problems into simpler ones to a point where most of the code becomes more obvious, and from experience we know how to spot complexity and put in relevant effort. For example, the tricky part of the code above is the line packing, so it helps to isolate the tricky bit and keep the rest as simple as we can make it. Then, we come up with line packing code that we're happy with or else do our best and start testing or proving it in various ways. We don't test all the things, just the ones where the effort is worth it, and we manage the complexity to limit how much needs that effort.

Though to put it in real terms, would I bet my motorbike on it? Almost yes, though a small sliver of doubt remains. The code leaves certain details implicit, like the assumptions on the input to line packing (for example fragments should not be larger than N) or that the long words are split to one or more fragments of size N followed by zero or one fragments of size less than N. If these assumptions somehow become incorrect, it's not certain that the code will still work. So, it would be nice to be more sure, as long as it didn't get too tedious. We'll see later how far dependent types can help to strike a workable balance.

Bob's Pipeline Code

Bob then compared his TDD-grown code to a more conventional FP style of coding. The top level function should look familiar, but difficulties arose for him in the intermediate steps—leading to some interesting discussion that will be addressed in the next section. Bob doesn't claim to be an expert on FP, and this section is certainly not a criticism of his code. Instead, let's explore how to go from this code towards a more polished version, as a way of explaining the programming style and to help recognize and avoid certain patterns.

Quite a few of you might go through this stage of learning, so hopefully it will help you get to the following stages a little bit more easily.

Again, I'm using Haskell: first because I can and I like it, but more seriously because I think it's easier to spot certain patterns and features when the code is expressed in the sparser syntax.

```
wrap s n
  = intercalate "\n"
  $ make_lines_up_to n
  $ break_long_words n
  $ words s
```

That's the top-level function above. It should be understandable by now. Breaking long words happens like this:

```
break_long_words n [] = []
break_long_words n (word:words)
 | n >= length word
 = word        : break_long_words n words
 | otherwise
 = take n word : break_long_words n (drop n word : words)
```

The code is quite straightforward, looping through the words, letting words through if they are short else taking N off the front of the word and looping again with the rest of the word at the head of the other words. No if-expressions in this Haskell version, you notice—it's doing everything with pattern matching and guards.

But thinking about the data structures involved, and the kind of transformation required, you might realize that it shouldn't be this complex. Looking more closely, you might start to see that this code is doing several orthogonal things, and that you can tease these apart. You might say, the above code is not exactly following FP's "single responsibility principle"...

The most important point is that splitting one word *is independent* of splitting any other, and this independence often means we can use `map`. Recall how mapping works— applying a function to each element in a list.

```
map f []     = []
map f (x:xs) = f x : map f xs
```

We can see a shadow of this pattern in `break_long_words`, though it's not immediately obvious how to refactor. The obstacle is the detail of actual word splitting. Rather than burn brain cells wondering about the refactoring, let's just try to isolate the code for single word splitting. It looks like this.

```
break_long_words_aux n word
 | length word <= n
 = [word]
 | otherwise
 = take n word : break_long_words_aux n (drop n word)
```

This is simpler, and clearly does something that could be used for other things. (It's not tied to words, so can split up lists of anything—its type is `Int -> [a] -> [[a]]`, that is, list of something to a list of lists of something.)

In case you're wondering, it is a form of loop but not a map or a fold. If anything, it's kind of a fold in reverse, that is, the same `unfold` introduced a few pages ago. Notice that the loop is not completely arbitrary or random—there is a structure to it and sometimes it helps to identify which

structure. For one thing, it saves having to read a longer definition and then getting annoyed that it is actually not worth the extra code...

One detail remains, which is to ensure we generate a list of fragments after splitting a list of words, given that splitting one word produces a list of fragments and we're applying this to a list of words to give us a list of lists of fragments. The answer is to use `concat :: [[a]] -> [a]`, which flattens a list of lists by appending the inner lists together— for example `concat [[1,2,3],[],[4,5,6]]` is equal to `[1,2,3] ++ [] ++ [4,5,6]` and hence `[1,2,3,4,5,6]`. Notice how the empty list disappears, much like for 0 in `2 + 0 + 3`.

The combination `concat` with `map`, typically as `concat (map f xs)`, occurs quite frequently when some function `f` is used to generate zero or more results (as a list) and mapped over a list of things, and you want to simplify the result. It's worth getting to know this pattern well. Putting it all together—and using `$` to avoid one set of parentheses—we arrive at this:

```
break_long_words n ws = concat $ map (break_long_words_aux n) ws
```

If you yet don't trust this rewriting, you can expand the definitions of `concat` and `map` and end up with code very close to the original version. This is one way to have confidence that your new code does the same thing as the old, and something that is often easier to do in a functional language. Finally, notice also how the new code says something useful about how the code works. It's not just a collection of symbols that happens to pass the tests.

Next up, the line packing. The auxiliary function does the real work, using two extra arguments to hold the current line being packed *and* the lines already completed. One easy simplification is to remove the width parameter `n` from the `aux` function: it's bound as a parameter of the main function and doesn't change throughout the `aux` function calls, so can be left out.

```
make_lines_up_to n words
 = aux_fn n words [] []
   where
     aux_fn n []            line lines
       = lines ++ [unwords line]
     aux_fn n (word:words)  line lines
      | n >= length (unwords new_line)
      = aux_fn n words new_line lines
      | otherwise
      = aux_fn n (word:words) [] (lines ++ [unwords line])
        where
          new_line = line ++ [word]
```

As before, thinking about the data structures and visualizing the transformation as a pipeline suggests various simplifications. The input is a list of fragments and the output is a list of (packed) lines. When packing, we put as many fragments as possible into the current line before spitting it out and going onto the next line, just like a hay baling machine on a farm. This suggests we can simplify the (already processed) "lines" accumulator. Once this detail is factored out, the code starts to look like my simpler version.

So, you might have followed the discussion and agreed with the code changes, but the important question is, can you "internalize" these ideas and start applying them successfully yourself? Here's what I suggest:

- always think about the data structures and look for a straightforward way to do the transformations
- know and understand how to use the key operations like mapping, folding, and filtering, and to a lesser degree other patterns like unfolding and accumulators, and always look for opportunities to use them in an appropriate way (but don't over-use them)
- always review your code and ask whether it is doing orthogonal things
- think about whether you can split complex operations into (pipelines of) simpler stages, possibly by introducing new data structures for an intermediate phase

Talking About It

After presenting his second version, Bob mentions the difficulties encountered when developing the code in an unfamiliar style, and says "it's hard for me to believe that my [TDD] solution to the word wrap is inferior to the [second version]." In the context, it is a fair point, though it does depend on a few assumptions.

Rather than argue about these assumptions (which could take some time), let's consider what we've seen here. I've shown an alternative way to approach the problem that has a stronger functional flavor, namely being more up front about the data structures being manipulated and using the language features to simplify some of the code. Tests have not played a significant role in the code's development, though they have been useful as a check on functionality.

What do you think? Now the approach and the technique have been explained, does my alternative seem more understandable, or less? Do you have more confidence in its correctness, or less? Which version would you prefer to maintain? And how would you like to program this kind of problem, imagining if you could use any language—past, present, or future? We'd welcome your views on the PragPub forum [U9]!

Combining TDD and Functional Thinking

As Bob demonstrated, TDD is indeed possible in a functional setting. I've argued for being able to think about the wider problem and use that insight in the coding process. Can we have our cake and eat it, that is, combine the strengths of these approaches? Here are two ideas to ponder.

First, on how to get more mileage out of existing tests when we decompose some operation into a pipeline. For example, if we decide that `wrap` should start off with a word-splitting stage (as done above), what can we do about testing the rest of the stages? Do we lose our tests? Or do we have to make up new ones? Happily, no. We can instead apply the first parts of the pipeline to the test input to get the input to be tested in the later stages. For example:

```
-- suppose that 'wrap' can be decomposed into three parts
wrap n s = stage3 $ stage2 n $ stage1 n s
-- and we decide that stage1 is this
stage1 n s = concat $ map (chunksOf n) $ words s
-- and given this test case
(Test7) wrap 3 "x xxxx" == "x\nxxx\nx"
```

Then we can use what we know about stage1 to derive a new test case automatically that indicates how stage3 plus stage2 should behave, and we do this by running stage1 on the original input to get the resulting input that stage2 will get. That is, stage1 3 "x xxxx" will give ["x", "xxx", "x"] hence we get a new test case of

```
-- replacing 'stage1 3 s' with its output
(Test7a) stage3 $ stage2 3 ["x", "xxx", "x"] == "x\nxxx\nx"
```

We can do a similar trick on the other end of the pipeline in certain cases—specifically when the final stage(s) (here stage3) are functions that can be run in reverse, so we can work out what the input was given their output. (In technical terms, when stage3, etc., is a *bijection*. Bijections got a bit of fame on Twitter recently...) So in the wrap example, if stage3 is intercalate "\n" then—assuming the input didn't contain any newlines—we can do the inverse operation by splitting on newlines. Hence with our example, we can work out the resulting test for just stage2.

```
-- using 'stage3' in reverse
(Test7b) stage2 3 ["x", "xxx", "x"] == ["x", "xxx", "x"]
```

So a simple way to get more mileage out of existing tests, but still requiring a bit of manual work.

Second, some of the newer programming environments actually automate some of the manipulations above, and allow new possibilities. Specifically, the *proof assistant* style of interface pioneered in dependent types (especially the more advanced interfaces in Agda and Epigram) allow programs to be developed interactively and run or tested *before* they are complete, even showing the partial results on various test cases using the code that has been filled in. For example, if we defined wrap to have three stages but only gave actual code for the first stage (leaving the rest as "undefined"), and tried to run a test case, it would do as much as it could using the code for stage one and then show what it was left with—which would be the expected input for stage two, etc. These are powerful and exciting tools, opening up new and interesting ways to develop code. I aim to demonstrate more of this next month.

Time for some Types

There's not too much opportunity to use complex types in this code, but note that types were always there in the background, supporting what we were doing.

They help to manage the transformation from a single string to a list of words, to a list of word fragments, then the packaging of these into a list of strings. We're using the String type for fragments and for lines, but there's nothing to stop us adding synonyms or abbreviations (for example type Word = String) if it helps remind us what is going on where.

A further step is to wrap up the various strings inside new types that allow firm distinctions between (for example) fragments and lines. Then, the line packing code would have a type Int -> [Fragment] -> [Line], and the type checker would be able to protect against another potential class of errors. Is this overkill? Maybe. It partly depends on the importance placed on a code component being correct, and the confidence that we have in the techniques or design

we are using. (Consider the risk of confusing different units, like cm vs inches, and the various real consequences that pop up in the news... It's the same kind of situation.)

Note that adding such new types in Haskell takes little effort, usually much less effort than setting up a new class in OO languages, and the cost of such techniques may be an acceptable given the potential added confidence.

There's a lot that Haskell-style types still can't say, and that tests don't really help with either. Like, are we sure that the line packing code (any version) always generates a line that fits inside the given width? For how much are we still assuming things or crossing our fingers? Fortunately, we have the technology to do better, and it doesn't involve pen & paper.

Dependent Types

Dependent types were introduced in an earlier article in this series as a richer language to explain what a program was doing, encoding both structural information *and* proofs about properties.

In closing, here are some hints on how the additional capabilities of dependent types might be used to improve confidence in the code:

- does long word splitting always produce the right size and order of fragments?
- will line packing never overflow, assuming that the input is valid?
- guarantee that the output doesn't add or delete or permute non-space characters

Next month I'll discuss how to write this word-splitting program in Idris [U10]. Idris is being developed by Edwin Brady (of WhiteSpace [U11] fame) as a practically-oriented dependently-typed programming language. It now also compiles to Javascript, thus opening up many more interesting possibilities for browser programming! I'm excited about it, anyway.

About the Author

Dr Paul Callaghan has been programming with Haskell for a while. Bits of his bio can be seen on earlier articles. Paul also flies big traction kites and can often be seen being dragged around inelegantly on the beaches of North-east England, much to the amusement of his kids. He blogs at free-variable.org [U12] and tweets as @paulcc_two [U13].

Send the author your feedback [U14] or discuss the article in the magazine forum [U15].

Paul Callaghan thanks Bob Martin for the interesting discussions.

External resources referenced in this article:

[U1] http://blog.8thlight.com
[U2] http://blog.8thlight.com/uncle-bob/2013/01/07/FPBE3-Do-the-rules-change.html
[U3] http://hackage.haskell.org/package/QuickCheck-2.5.1.1
[U4] http://rushcheck.rubyforge.org/
[U5] https://gist.github.com/4577526
[U6] http://hackage.haskell.org/packages/archive/split/0.2.1.1/doc/html/Data-List-Split.html#v:chunksOf
[U7] https://www.google.co.uk/search?q=haskell+library+split+list
[U8] http://www.haskell.org/hoogle
[U9] http://forums.pragprog.com/forums/134
[U10] http://idris-lang.org/

[U11] http://en.wikipedia.org/wiki/Whitespace_(programming_language)
[U12] http://free-variable.org
[U13] http://twitter.com/paulcc_two
[U14] mailto:michael@pragprog.com?subject=functional programming
[U15] http://forums.pragprog.com/forums/134

Becoming a Self-Certified Pomodoro Master

Step Four: Estimating

by Matthias Günther

Using the Pomodoro Technique, developed by Francesco Cirillo, you work in focused sprints throughout the day. Matthias Günther delivers Pomodoro wisdom in short sprints on this series.

If you have followed my article series [U1] to this point you should know the rudiments of how to use the Pomodoro Technique to get control of your time. You know what Pomodoros are, how to run them, and how to defend yourself from interruptions. In this installment you will learn how you can better estimate your time for doing Pomodoros.

Is Estimation Evil?

I have to do it every sprint at work and nearly every time it is difficult for everyone in the team: Estimation.

If you are the Product Manager, you need estimates from the developer about features in order to prioritize them. You will use this information to determine how much effort it takes to build the feature and how much business value will be generated for the customer. This is part of the cost-benefit analysis. You will use this information to create a release plan of your backlog. For example, you want your marketing people to advertise the new feature to the users just in time and communicate it when it's out and not the other way round. We all know that just-in-time is expected but difficult to achieve. The only constant thing in software development is change, and so we know that the requirements will change, and with them your plans.

Knowing all the requirements of a feature is almost impossible. There will always be uncertain variables:

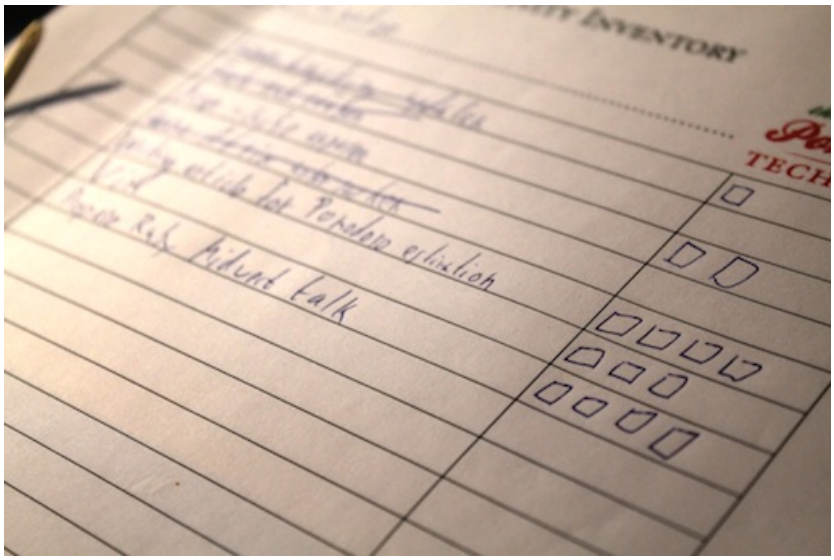
- some of your teammates get the flu,
- the Product Manager is stuck for the whole week in meetings and isn't available for questions,
- developers have problems with the infrastructure because the system team is doing some important migrating job,
- or you are fighting very nasty bad bugs, or you detect during the implementation that the design needs a whole refactoring, or...
- I'm sure can think more of these variables on your own.

All developers know about and expect these kinds of problems. But we tend to underestimate the time we need to solve the problems and we almost forget the unknown obstacle which will appear in some way. As a developer you have to chose a strategy to solve the problem, build the thing, and show it to your customer, and the sooner the better. Getting early feedback gives you a chance to adapt your feature to changes in the requirements instead of building in the wrong direction. This will save you time and money.

You need a plan for doing things but you know that your plan will not work in the way you want. You'll have to adapt it in small steps. You can learn more about this topic in Ron Jeffries [U2]' article [U3], "Why Estimation Is Evil." Let's see what the Pomodoro Technique has to offer.

Make Estimations in Pomodoros

Remember the *Activity Inventory Sheet*? Right, it's the piece of paper that is the gathering of all the things you want to do today. There is a special column in the sheet where you can draw boxes. Each box stands for a Pomodoro. There you have your estimation guessing area for each task. When one Pomodoro is over you make a cross in the next box. Each crossed box stands for one Pomodoro you've spend on the task.



Your first estimation.

The first image shows the task "Ruby Midwest talk." It has four boxes. That means that I think that I need four Pomodoros to prepare the talk. Since I haven't started any Pomodoros for this task, the boxes are empty.

You may ask what the perfect size for a task should be? In my eyes something between 1–4 Pomodoros is a perfect fit. If you think a task takes more then 4 Pomodoros it is actually too big and you should break it down. Splitting up big problems is like climbing mountains—taking small steps saves you from falling down and you will have enough time to think over your route, and you will have more time to take some pictures during your trip. The smaller the problem is the better you are able to estimate.

Determine Your Available Pomodoros for a Day

Before writing down your estimations for the each task you first need to determine how many Pomodoros you have today. A bad day may only consists of four Pomodoros for concentrated working.

On nice days, where you have the feeling that you are actually working, you have between 8–13 Pomodoros. These are times where you are feeling the team spirit, leaving your comfort zone, reaching for the stars, and making the impossible deadline possible. You are happy to do your beloved programming job. Yes, you're feeling smart.

Have your "available-Pomodoro-number-per-day" in mind before you create a list with tasks for the day. Think of these tasks as a commitment (or "promise" to say it in with the words of Uncle Bob [U4]) for day. The nice side effect of this strategy is that you are limiting yourself about the things you can really finish that day. Kanban [U5] describes this way of working as "Stop Starting, Start Finishing."

Tracking Your Estimation Progress

At the end of the day.

At the end of the day you should have have a list like the one in the next image. As you can see there are tasks where not all boxes are crossed. These are tasks that took less time that you thought in the beginning (*overestimation*). If there are more crosses then boxes beside a task you have take more time that you estimated for it (*underestimation*).

At work I track the time I need to solve a problem, but instead of thinking in a classical time estimating mode, I'm using Pomodoros as my metric. At the beginning I may have had 4 or more at some task because I was working in hostile environment without any tests and very old legacy code. It was more than 5 year old code and all comments were wrong. If you are not experienced with this kind of work you can triplicate the time you think you need at the beginning.

Description of Activity	Estimate	Real	Diff.
Answer questions on thermodynamics in Ch 4	2	2	0
Repeat laws of thermodynamics out loud to Mark	3	2	-1
Summarize laws of thermodynamics in writing	3	4	1

Picture of the Record Sheet.

You can track your estimates on the *Record Sheet*. The difference between estimated and real results shows you how well you estimated your task. If the differences for tasks are between -2 to +2 you're pretty good at estimating.

I'm not using the *Record Sheet* yet. I'm only using the Activity Inventory Sheet to track the estimations of my Pomodoros. Out of historical data I can say that I'm a much better estimator than I was before I was using the Technique. The most important thing is to recognize when and why you need so long for a task. Was it due to lack of skill, due to some stupid decision, because you didn't talk to anyone about the way you want to solve the problem, or because you were solving the wrong problem? We all make mistakes, and mistakes are the best fertile ground to build up your developer skills.

Don't give up if you have a huge gap between the estimated and needed Pomodoros. This will work better if you keep practicing it.

Conclusion

Estimation is one of the things programmers hate. We all have trouble predicting the time we need for a story or single task. Only after running a marathon do you know how long it took. It is the same if you are working with your team on a long-term project. Only when you are finished with it do you know where the bottlenecks were. Apparently every large project in IT is different and difficult to estimate, and there will always be external dependencies or unforeseen problems that will destroy you plan. Edmund Burke [U6] summarizes this perfectly with "You can never plan the future by the past."

But by using the Pomodoro metric to track your commitment, you become better at estimating and at keeping your promises to your users, Product Owner, Team, and yourself.

About the Author

Matthias is an Open Sourcer by heart, loves giving presentations about Vim [U7], and writes a book about Padrino [U8]. When Günther is not working as a developer at MyHammer [U9], he spends his free time visiting hacking events, painting small figures, running for his health, organizing Ruby conferences like eurucamp 2013 [U10], and experimenting with making delicious cakes. He lives in Berlin and is blogging under wikimatze.de [U11] You can see Matthias speaking at Ruby Midwest 2013 [U12] about using the Pomodoro Technique for Open Source work.

Send the author your feedback [U13] or discuss the article in the magazine forum [U14].

External resources referenced in this article:

[U1] http://www.pragprog.com/magazines/
[U2] http://xprogramming.com/index.php
[U3] http://pragprog.com/magazines/2013-02/estimation-is-evil
[U4] https://twitter.com/unclebobmartin
[U5] http://en.wikipedia.org/wiki/Kanban
[U6] http://en.wikipedia.org/wiki/Edmund_Burke
[U7] http://www.vim.org/
[U8] https://github.com/matthias-guenther/padrino-book
[U9] http://www.my-hammer.de/
[U10] http://2013.eurucamp.org/
[U11] http://wikimatze.de/
[U12] http://www.rubymidwest.com/
[U13] mailto:michael@pragprog.com?subject=Pomodoro
[U14] http://forums.pragprog.com/forums/134

Solved!

Code in Code

by Michael Swaine

Last issue I presented a short program. It was an example designed to introduce a particular programming language, published on the website for the language, and written by the language's author. The only modification I made to it is to encrypt the letters with a simple substitution cipher. All numerals, punctuation, spacing, and capitalization remained unchanged. The challenge was to answer this simple question:

What is the language?

The answer is that the language is Rebol, which you can learn about at the company site [U1], and the author of the program and the language is Carl Sassenrath. Here's the decoded code:

```
REBOL [
    Title: "Digital Clock"
    Version: 1.3.3
    Author: "Carl Sassenrath"
    Purpose: {A simple digital clock.}
]

f: layout [
    origin 0
    b: banner 140x32 rate 1
        effect [gradient 0x1 0.0.150 0.0.50]
        feel [engage: func [f a e]
                        [set-face b now/time]]
]

resize: does [
    b/size: max 20x20 min 1000x200 f/size
    b/font/size: max 24 f/size/y - 40
    b/text: "Resize Me"
    b/size/x: 1024 ; for size-text
    b/size/x: 20 + first size-text b
    f/size: b/size
    show f
]

view/options/new f 'resize
resize
insert-event-func [
    if event/type = 'resize [resize]
    event
]
do-events
```

External resources referenced in this article:

[U1] http://www.rebol.com/

Calendar

Want to meet one of our authors face-to-face? Here's where they'll be in the coming months.

Here's what our authors will be up to in the coming months.

2013-03-01 **Keynote at a conference focused on developer joy**
Chad Fowler (author of The Passionate Programmer (2nd edition) [U1] and Rails Recipes [U2])
The Joy of Coding - Rotterdam, the Netherlands [U3]

2013-03-01 **Java is from Mars, Ruby is from Venus**
Paolo Perrotta (author of Metaprogramming Ruby [U4])
Joy of Coding, Rotterdam, Netherlands [U5]

2013-03-04 **Vim Masterclass**
Drew Neil (author of Practical Vim [U6])
Online [U7]

2013-03-04 **An opportunity to hang out with fellow speakers.**
Venkat Subramaniam (author of Practices of an Agile Developer [U8] , Programming Groovy [U9] , Programming Scala [U10] , Programming Concurrency on the JVM [U11] , and Programming Groovy (2nd edition) [U12])
SpeakerConf - Aruba [U13]

2013-03-07 **Game Dev Workshop**
Jonathan Penn
CocoaConf Chicago [U14]

2013-03-07 **iPad Productivity APIs (All-Day Tutorial)**
Chris Adamson (author of iOS SDK Development [U15])
CocoaConf Chicago [U16]

2013-03-07 **Mobile Movies with HTTP Live Streaming**
Chris Adamson
CocoaConf Chicago [U17]

2013-03-07 **Core Audio in iOS 6**
Chris Adamson
CocoaConf Chicago [U18]

2013-03-08 **Zen and the Art of iOS Gesture Recognizers**
Jonathan Penn
CocoaConf Chicago [U19]

2013-03-08 **Keynote: Agile Management**
Johanna Rothman (author of Behind Closed Doors [U20] , Manage It! [U21] , Manage Your Project Portfolio [U22] , and Hiring Geeks That Fit [U23])
AgileIndy Conference [U24]

2013-03-08 **Talks on HTML 5 and JVM languages.**
Venkat Subramaniam
NFJS - Minneapolis, MN [U25]

2013-03-09 **UI Automation - Automate ALL THE THINGS!**
Jonathan Penn
CocoaConf Chicago [U26]

2013-03-12 **Change -- How long does it take?**
Staffan Nöteberg (author of Pomodoro Technique Illustrated [U27] and Pomodoro Technique Illustrated (audio book) [U28])
Lean Kanban Nordic, Stockholm [U29]

2013-03-13 **Various JVM languages related topics.**
Venkat Subramaniam
33degree - Warsaw, Poland [U30]

2013-03-14 **Change -- How long does it take?**
Staffan Nöteberg
DevLin 2013, Linköping [U31]

2013-03-20 **Three days, two presenters, one objective—to get you up and running with Rails**
Dave Thomas (author of Programming Ruby (2nd edition) [U32] , Agile Web Development with Rails (3rd edition) [U33] , The Ruby Object Model and Metaprogramming [U34] , Agile Web Development with Rails (4th edition) [U35] , and Programming Ruby 1.9 & 2.0 (4th edition) [U36])
Pragmatic Studio, Reston, VA [U37]

2013-03-21 **iPad Productivity APIs (All-Day Tutorial)**
Chris Adamson
CocoaConf DC [U38]

2013-03-21 **Mobile Movies with HTTP Live Streaming**
Chris Adamson
CocoaConf DC [U39]

2013-03-21 **Core Audio in iOS 6**
Chris Adamson
CocoaConf DC [U40]

2013-03-25 **CoreData In Motion - An overview of how CoreData works, when to use it, and how to use it in RubyMotion**
Jonathan Penn
#inspect 2013 - Brussels, Belgium [U41]

2013-03-28 **Wrapping RubyMotion - Discussing techniques for making Objective-C APIs more pleasant in Ruby**
Clay Allsopp (author of RubyMotion [U42])
#inspect, Brussels Belgium [U43]

2013-04-04 **Clojure for the Unsuspecting**
Paolo Perrotta
Ancient City Ruby, St. Augustine, Florida, USA [U44]

2013-04-04 **A 1-day conference dedicated to Cucumber, Specification by Example and BDD.**
Aslak Hellesøy (author of The RSpec Book [U45] , The Cucumber Book [U46] , and Cucumber Recipes [U47])
CukeUp!, London [U48]

2013-04-04 **JavaScript related topic**
Venkat Subramaniam
DenverJS - Denver, CO [U49]

2013-04-05 **Talks on HTML 5 and JVM languages.**
Venkat Subramaniam
NFJS, NYC [U50]

2013-04-12 **Workshop "Gumption Traps Reloaded" with Ivan Moore and talk "Is eXtreme Programming still alive and kicking?"**
Rachel Davies (author of Agile Coaching [U51])
ACCU, Bristol, UK [U52]

2013-04-18 **Core Audio Workshop (All-Day Tutorial)**
Chris Adamson
CocoaConf San Jose [U53]

2013-04-18 **Mobile Movies with HTTP Live Streaming**
Chris Adamson
CocoaConf San Jose [U54]

2013-04-18 **Core Audio in iOS 6**
Chris Adamson
CocoaConf San Jose [U55]

2013-04-21 **5.5 days of leadership training for people who want to learn to be problem solving leaders.**
Johanna Rothman
Problem Solving Leadership Workshop, Albuquerque, NM [U56]

2013-04-21 **Talks on HTML 5 and JVM languages.**
Venkat Subramaniam
NFJS, Reston, VA [U57]

2013-04-22 **"How to Design Indexes, Really" and "Extensible Data Modeling with MySQL"**
Bill Karwin (author of SQL Antipatterns [U58])
Percona Live MySQL Conference and Expo 2013 [U59]

2013-04-22 **A talk on something other than Rails. This is a Rails conference about everything except for Rails.**
Chad Fowler
Railsberry - Krakow, Poland [U60]

2013-05-07 **Workshops and talks on various topics**
Venkat Subramaniam
Great Indian Developer Summit, Bangalore [U61]

2013-05-20 **Tutorial: Hiring for Your Team: Culture Trumps Skills**
Johanna Rothman
Let's Test Conference, Sweden [U62]

2013-05-21 **Keynote: Becoming a Kick-Ass Test Manager**
Johanna Rothman
Let's Test Conference, Sweden [U63]

2013-05-22 **Talks on various topics related to Groovy and the JVM.**
Venkat Subramaniam
GR8Conf, Copenhagen [U64]

2013-05-28 **Flow Control with Promises: Learn to control async tasks in JavaScript with Promise-based interfaces.**
Trevor Burnham (author of CoffeeScript [U65] and Async JavaScript [U66])
Fluent 2013 [U67]

2013-05-29 **Change -- How long does it take?**
Staffan Nöteberg
DevSum 2013, Stockholm [U68]

2013-06-06 **Exploding Management Myths: Johanna will explain the management myths and what to do instead. She'll tackle favorites such as 100% utilization, no time for training, promoting the best technical person and more.**
Johanna Rothman
Agile Development Conference/Better Software, Las Vegas [U69]

2013-06-12 **Talks on various topics**
Venkat Subramaniam
NDC Oslo [U70]

2013-07-16 **Workshops and talks on various topics.**
Venkat Subramaniam
ÜberConf [U71]

2013-09-11 **Get a flying start with BDD, the collaborative process that's changing the face of software development.**
Matt Wynne (author of The Cucumber Book [U72] and Cucumber Recipes [U73])
BDD Kickstart, Barcelona [U74]

2013-09-14 **Hexagonal rails**
Matt Wynne
Barcelona Ruby Conference [U75]

2013-09-14 **TBD**
David Chelimsky (author of The RSpec Book [U76])
Baruco [U77]

O'Reilly Events

Upcoming events from our friends at O'Reilly.

2013-03-24 **Tools of Change Bologna: "Happening in conjunction with the Bologna Children's Book Fair, TOC Bologna is where the children's publishing and technology industries come together to discuss, learn, and share ideas around the art, craft, and business of storytelling in the digital age."**
Bologna, Italy [U78]

2013-05-28 **Fluent Conference: "This year, Fluent will bring together an even broader range of individuals and organizations and drill even deeper into the essential technologies and tools that power the Web."**
Fluent [U79], San Francisco, CA

USENIX Events

What's coming from our USENIX friends.

2013-03-15 **Cascadia IT Conference 2013: "A gathering of professionals from the diverse IT (computer and network administration) community in the U.S. Pacific Northwest / British Columbia to learn, share ideas, and network."**
Seattle, WA [U80]

2013-04-02 **5th USENIX Workshop on the Theory and Practice of Provenance: "The workshop may cover any topic related to theoretical or practical aspects of provenance, including but not limited to: provenance in databases, workflows, programming languages, security, software engineering, or systems; provenance on the Web; real-world applications of or requirements for provenance."**
Lombard, IL [U81]

2013-04-03 **10th USENIX Symposium on Networked Systems Design and Implementation: "NSDI focuses on the design principles, implementation, and practical evaluation of large-scale networked and distributed systems."**
Lombard, IL [U82]

2013-04-15 **Eurosys 2013: "EuroSys has become a premier forum for discussing various issues of systems software research and development, including implications related to hardware and applications."**
Prague, Czech Republic [U83]

2013-05-03 **LOPSA-East Professional IT Community Conference: "LOPSA-East is an annual conference held in NJ with the goal of providing a forum where system administrators can come together, network with their industry peers, share war stories, and learn from some of the greatest minds in the profession."**
New Brunswick, NJ [U84]

2013-05-13 **14th Workshop on Hot Topics in Operating Systems: "Continuing the HotOS tradition, participants will present and discuss new ideas about computer systems research and how technological advances and new applications are shaping our computational infrastructure."**
Santa Ana Pueblo, NM [U85]

March Birthdays

March 1 **Yahoo**

March 3 **GNOME**

March 11 **J.C.R. Licklider**

March 13 **Microsoft as a public company**

March 16 **Richard Stallman and Andy Tanenbaum**

March 24 **Mac OS X**

March 25 **The Wiki**

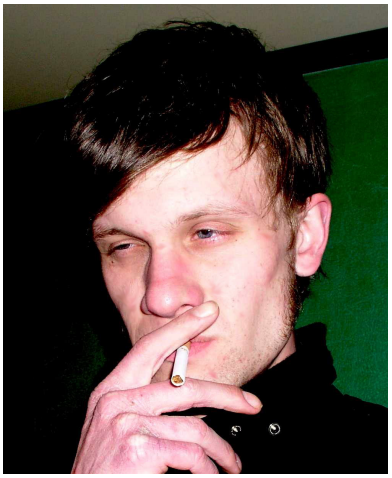
March 26 **Larry Page**

Shady Illuminations

Clouds and Caves

by John Shade

John is so distressed by the social nature of software today that he turns to poetry.



Seems like every tech news story these days is about collaboration and the wisdom of crowds. Everybody's pair programming, groupthinking the genome, distributedly searching for aliens, micromanaging their social media reputations, and you gotta be working on an open source project on Github. It's a moshpit out there, everybody bumping up against one another for the promise of sparks. It's all crowd this and cloud that in this wacky Wiki Wiki world.

Sort of discouraging to a misanthropic hermit like myself.

I mention all this as a possible explanation for the fact that the following news items caused me to break out in iambic pentameter.

Crafty Crowdsourcing

Pinterest is one of those whowoodathunk success stories in social software. Like all such ventures it is a species of grafitti or leafleting, with the virtual wall or lamppost supplied by a virtual company that in turn rents it by the minute from Amazon, this being what is known these days as a business model. As Pinterest's founder Ben Silbermann told MIT's Technology Review [U1], search algorithms don't know what you want, but strangers on the internet do.

And apparently he's right. Pinterest's users post collections of images on what Silbermann calls "pinboards" in order to share their interest in antique dolls or things containing chocolate. You would think that would be enough of a contribution to society for anyone, but Silbermann likes to think about Pinterest as a tool for finding things online. He calls it a human indexing machine, and says it works exceptionally well. "The whole reason Pinterest exists," he said, "is to help people discover the things that they love and [somehow cause someone to give us money]."

```
It's something that I never would have guessed:
That crowdsourced wisdom truly stands the test.
As search tools go, crowdsourcing is the best
If what you seek is crap on Pinterest.
```

Collegiate Collaboration

Oxford University recently blocked access to Google Docs all across the campus network. Students were outraged, according to Paul McNamara writing in Buzzblog [U2]. Their natural need to collaborate online was being thwarted.

Oxford took the action, University mouthpieces said, because Google Docs, particularly Google Forms, was being widely used for phishing purposes by unscrupulous ne'er-do-wells to take over email accounts of students and crank

out spam. So, taking their cue from the Old West, they shot those train robbers right in the horse.

After two and a half hours the University reinstated Google Docs and with it the *status spammus quo ante*.

```
Reacting to a rash of phishing, Ox-
Ford University flips out and blocks
All students' access to their Google Docs.
How will they write those papers for the jocks?
```

Smells Like Team Spirit

Ever heard of Braess' Paradox? It crops up in traffic planning and power transmission networks and materials science and team sports. An article in arXiv [U3] reveals that it can also infect social networks like Pinterest.

German mathematician Dietrich Braess showed that removing roads from a system of roads can lead, paradoxically, to less traffic congestion. Or compressing a material can cause it to expand. Or removing one player from a team can improve team performance. OK, that last one doesn't seem particularly paradoxical to me. Some people are just negative. My friends keep telling me.

When applied to social networks, Braess' Paradox identifies situations where reducing the amount of information you're getting from those helpful online recommendations can lead to better decisions. And not because you've filtered out the bozos. Although presumably that helps, too.

```
When traffic flow's improved by closing streets,
And teams improve when mates hang up their cleats,
Can sailboats catch the breeze with folded sheets
And armies win their wars in full retreat?
```

Groupthink Triumphant

Computer Science professor Peter Fröhlich of Johns Hopkins University developed a grading system in which he gave an A for the highest score on the final exam, and everybody else's grade was scaled accordingly. It worked well until last fall, he told Inside Higher Ed [U4].

At the end of the fall semester, students in three of Fröhlich's classes realized that there was a fatal flaw in his grading algorithm. If every student got the same score on the final, then that would be the highest score and would be assigned an A. So every student would get an A. This would be true for any score, even zero. So if all the students in class boycotted the exam, thus getting a zero, they would all get As.

The students did just that—arranging the boycott using social networks, of course.

Fröhlich gave them all As. What else could he do?

```
These Hopkins students, I just think they're great.
With them I think I could collaborate.
Refuse to play, yet force a check and mate:
A moveless move I much appreciate.
```

Clouds on the Horizon

"Mobility, social networking, cloud, and big data will be the four major trends for the next decade," according to Deccan Herald[U5], Bangalore. And "[c]loud drives the other three trends."

So there's no use fighting it then. I think I'll just go find a place to hide until this blows over.

```
I shouldn't criticize what folks take pride in;
Give all of them their fluffy clouds to ride in,
But while you're at it think about providing
A cave for misanthropes like me to hide in.
```

I didn't say *good* iambic pentameter.

Late-breaking bulletin: my editor informs me that "Oxford students don't do that sort of thing." I wouldn't know. I went to school in the United States. Several of them.

About the Author

John Shade was born under a cloud in Montreux, Switzerland, in 1962. Subsequent internment in a series of obscure institutions of ostensibly higher learning did nothing to brighten his outlook. He inclines to the Marxist view on not wanting to belong to any group that would have him as a member. Follow John on Twitter[U6], send him your feedback[U7], or discuss the article in the magazine forum[U8].