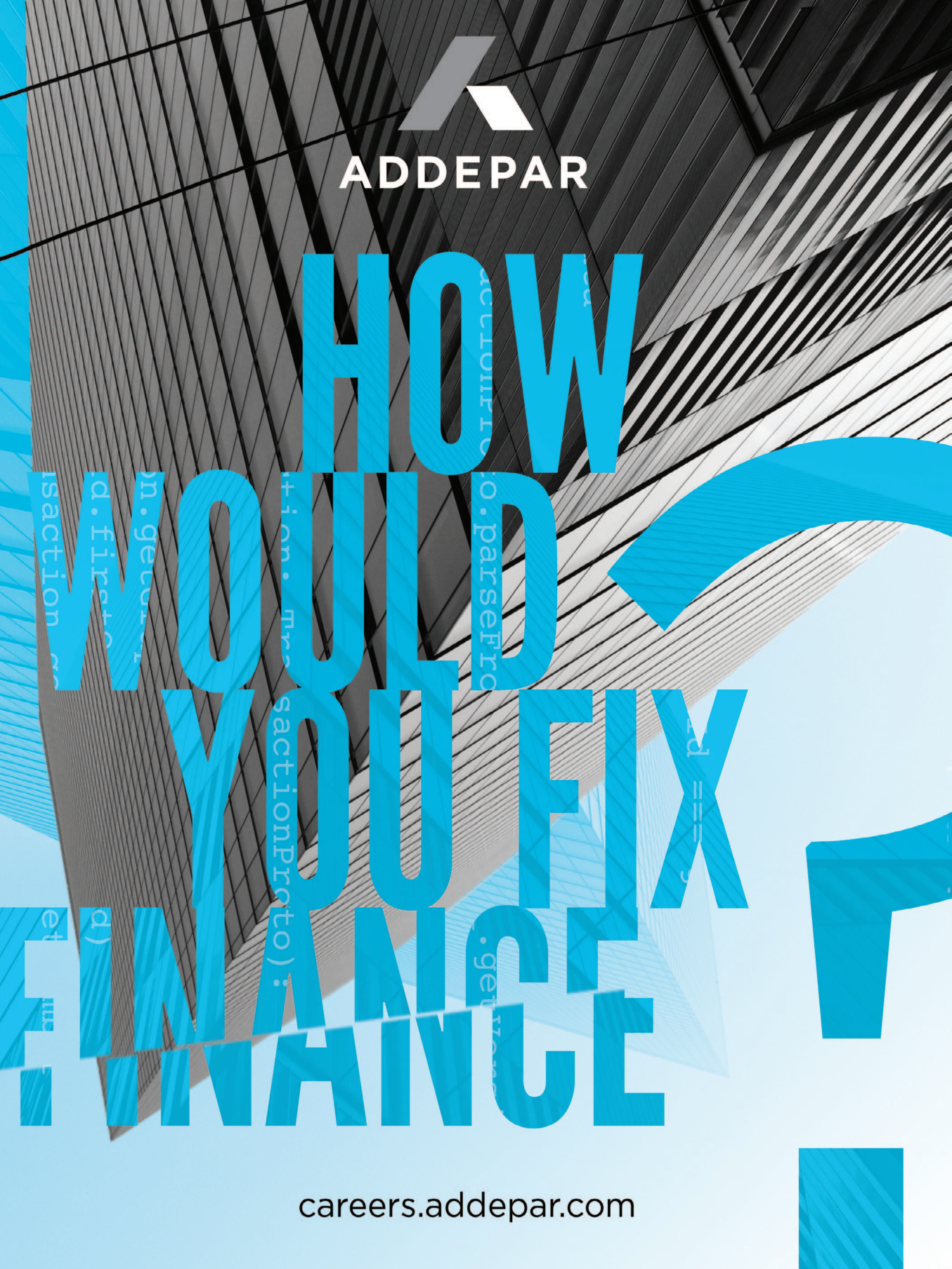
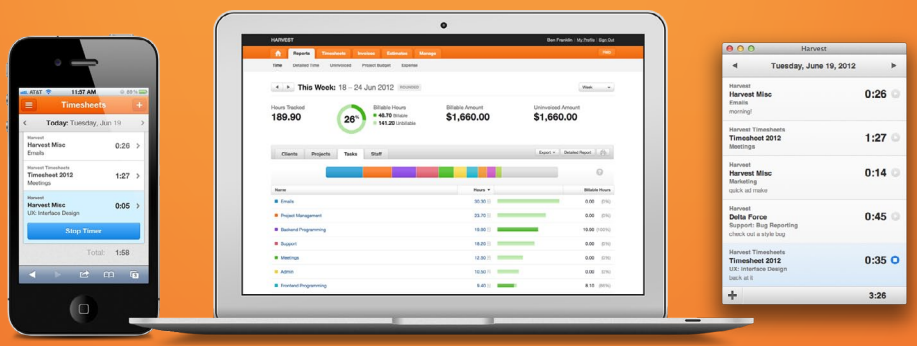
The Slow Web

Jack Cheng

HACKERMONTHLY

Issue 28
September 2012


ADDEPAR
HOW
WOULD
YOU FIX
FINANCE?
careers.addepar.com



HARVEST

Still using that rusty old Perl time tracking script you wrote when Reagan was in office? Try Harvest for two weeks and let us show you a better way to track time and get paid.

getHarvest.com/hackers

Curator
Lim Cheng Soon

Contributors
Jack Cheng
Kent Nerburn
Avery Pennarun
Adam Wiggins
Elaine Wherry
Sondra Eklund
Teddy Worcester
Emma Coats
Steve Hanov
Fernando Meyer
Andy Boothe
Joe Zim

Proofreaders
Emily Griffin
Sigmarie Soto

Printer
MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format.
For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.

Cover Photo by: Jenny Downing

Contents

Photo by: Ben Fredericson

For links to Hacker News dicussions, visit *hackermonthly.com/issue-27*

The Slow Web

By JACK CHENG

Photo by: Jenny Downing [*flickr.com/photos/jenny-pics/6908438828]*

One of the better spots to enjoy a bowl of ramen noodles here in New York is Minca, in the East Village. Minca is the kind of place just enough out of the way that as you're about to get there, you start wondering if you've already passed it. A bowl of noodles at Minca isn't quite as neatly put together as those of other ramen establishments in the city, but it is without a doubt among the tastiest. There's a home-cooked quality to a bowl of noodles at Minca, and there's a homey vibe to the restaurant. Minca is a good place to meet a friend and sit and talk and eat and drink, and eat and talk and sit and drink some more.

The last time I was at Minca, I had an especially enjoyable conversation with Walter Chen. Walter is the CEO of a company called iDoneThis, a quiet little service that helps you catalog the things you've accomplished each day. iDoneThis sends you a daily email at your specified time, and you simply reply with a list of things you did that day. It's useful for teams who want to

keep track of what everyone is working on, and for individuals who just want to keep track.

I first reached out to Walter because I was mesmerized by this koan at the bottom of the daily emails:

> *iDoneThis is a part of the slow web movement. After you email us, your calendar is not updated instantaneously. But rest up, and you'll find an updated calendar when you wake.*

iDoneThis is a part of the slow web movement. The Slow Web Movement. I had never heard that phrase before. I immediately started digging around — and by that I mean I googled "Slow Web Movement" — and the lone relevant search result was a blog post from two years ago. If you run the search again today, you'll find Walter's writeup on his company blog, which reflects a lot of what he told me over dinner.

As we talked further, I said to Walter that as soon as I saw "the slow web movement," I assigned my own meaning to it. Because it's a great name, and great names are like knots — they're woven from the same stringy material as other words, but in their particular arrangement, they catch, become junctions to which new threads arrive and from which other threads depart. For me, "The Slow Web" neatly tied together a slew of dangling thoughts.

Slow Web and Slow Food

The Slow Web Movement is a lot like the Slow Food Movement, in that they're both blanket terms that mean a lot of different things. Slow Food began in part as a reaction to the opening of a McDonald's in Piazza di Spagna in Rome, so from its very origin, it was defined by what it's not. It's not Fast Food, and we all know what Fast Food is… right?

Yet, if you ask a bunch of people to describe to you the qualities of Fast Food, you're likely to get a bunch of different answers: it's made from low-grade ingredients, it's high in sugar, salt and fat, it's sold by multinational corporations, it's devoured quickly and in overlarge portions, it's McDonalds/TacoBell/Subway, even though Subway has spent a lot of money marketing fresh bread and ingredients — it's still Fast Food albeit "healthy" Fast Food.

Fast Food has an "I'll know it when I see it" quality, and it has this quality because it's describing something greater than all of its individual traits. Fast Food, and consequently, Slow Food, describes a feeling that we get from food.

Slow Web works the same way. Slow Web describes a feeling we get when we consume certain web-enabled things, be it products or content. It is the sum of its parts, but let's start by describing what it's not: the Fast Web.

The Fast Web

What is the Fast Web? It's the out-of-control web. The oh-my-god-there's-so-much-stuff-and-I-can't-possibly-keep-up web. It's the spend-two-dozen-times-a-day-checking web. The in-one-end-out-the-other web. The web designed to appeal to the basest of our intellectual palettes — the salt-sugar-and-fat-of-online-content web. It's the scale-hard-and-fast web. The create-a-destination-for-billions-of-people web. The you-have-two-hundred-twenty-six-new -updates web. Keep up or be lost. Click me. Like me. Tweet me. Share me. The Fast Web demands that you do things and do them now. The Fast Web is a cruel wonderland of shiny, shiny things.

Timely vs. Real-Time

One of the centerpieces of the Fast Web is this notion of real-time. Your friend listens to a song, and you find out about it. The smaller the gap between these two, the closer it is to real-time.

Real-time interactions happen as they happen. Timely ones, on the other hand, happen as you need them to happen. Some real-time interactions, like breaking news about an earthquake, can be timely. But not all timely interactions are real-time. I'd argue that most are not. And where the Fast Web is built around real-time-ness, the Slow Web is built around timeliness.

A great example of a Slow Web product is Instapaper. Instapaper takes the process of discovering a long article and reading it on the spot (real-time). It breaks it apart, deferring the act of reading until later when we have an extended moment to read (timely). I may be stretching my analogy a bit here, but it's kind of like boxing up a meal and putting it away in the fridge for when you're hungry, except in this case, it doesn't lose as much of its taste.

Likewise, iDoneThis takes a pretty standard interaction of creating an item in a database and then reading it back — one that might normally take less than a few seconds to execute — and blows it apart.

A typical app might work like this: there's a text field for you to type in what you did. You type it in and hit submit. The database gets updated and almost instantly you see the submitted text displayed back to you. iDoneThis takes those last two steps — the update and the review — and stretches them out from a few milliseconds to half a day. The database gets updated sometime overnight and the display-back happens the next morning in your inbox.

Another name for this is turn-based, as in turn-based gaming. A traditional game of Scrabble or Pictionary is relatively demanding in real-time: it requires two or more people in the same place with the both desire and freedom to play these games. Deconstructing the real-time experience gives you the Words With Friends and Draw Somethings of the world. An activity that would otherwise be impractical can now carry on in a manner more timely for each participant. Instapaper is turn-based reading. iDoneThis is turn-based data tracking.

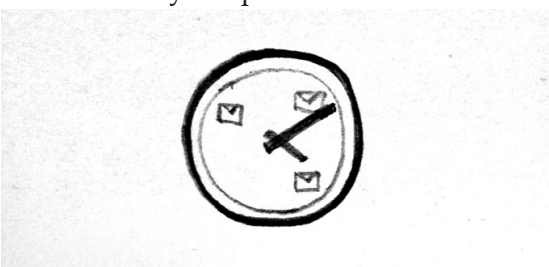
But timeliness alone doesn't make something Slow Web. Email, after all, is turn-based communication, and our email inboxes are probably one of the biggest sources of Fast Web distress. Those turn-based games can also quickly get overwhelming if we have too many of them going at once. What's missing in these cases is an inherent sense of rhythm.

Rhythm vs. Random

Let's say I told you there was a new HBO drama that aired for one hour from 9-10pm every Wednesday night. Once you decide it's a show you're interested in and can make room for, the act of watching takes over. It becomes about the show. Now, let's say I told you there's a new HBO drama that's sometimes times an hour, sometimes half an hour, sometimes two hours, that may or may not air every Tuesday, Wednesday, and Thursday night, between 6 and 11pm. Suddenly it's no longer just about the show. It's about whether or not the show will be on. What next? becomes When next?

In the Fast Web, we're faced with this proposition numerous times a day. The randomness and frequency of the updates in our inboxes and on our dashboards stimulate the reward mechanisms in our brain. While this can give us a boost when we come across something unexpectedly great, dependency leads to withdrawal, resulting in a roller coaster of positive and negative emotions. The danger of unreliable rhythms is too much reward juice.

Reliable rhythms lead to predictable outcomes, and rhythm is an expression of moderation. Apps like iDoneThis have this moderation: you receive your email prompt at the same time each day, and each interaction is similarly demanding. Unlike your inbox, where there can be a large range of demands: there are newsletters you can scan and trash, personal emails that require lengthy responses, and everything in between. The lack of moderation means sometimes you spend a few minutes going through your inbox, and other times you spend a few hours.



That's why most email productivity systems are concerned with a form of moderation: standardization. They encourage you to standardize the size and demands of the interaction (archive or delete messages and move on, transfer email requiring lengthy follow-ups to a to-do list, limit responses to three sentences [three.sentenc.es]) and standardizing the frequency (limit checking email to x times a day, at specified times).

A great example of rhythm and moderation in practice is the rollout of Wander [onwander.com]. For the weeks leading up to their beta launch, Keenan and crew took what could have been a first-run experience on another site and stretched it out over the course of four weeks into something akin to an advent calendar. Every week there is a similarly demanding interaction: give a place, pick a photo, type a reason.

Another service that does this well is Budge [bud.ge] from Buster and the team at Habit Labs. Budge is built around notifications reminding you to do the daily things that improve your life in small but beneficial ways, like flossing, meditating, or tracking your weight. Once you've signed up, you can interact with Budge solely through their notifications. In the past I've gone for weeks without visiting their site or app while still happily using the service just by replying to the timed texts I get on my phone.

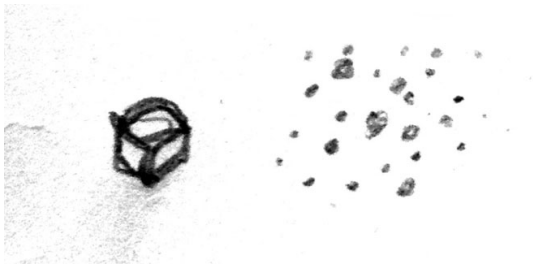
This is a tremendously important distinction between Slow Web and Fast Web. Fast Web is destination-based. Slow Web is interaction-based. Fast Web is built around homepages, inboxes, and dashboards. Slow Web is built around timely notifications. Fast Web companies often try to rack up page views, since page views mean ad impressions. Slow Web companies tend to put effectiveness first. Here's the crazy thing about Budge: the better it works, the less I use it. Once I get in the habit of flossing, my brain takes over, and I no longer need the notifications. Walter describes this credo well in the aforementioned blog post:

> *Behavior change, not growth. Behavior change is about improving the lives of others, scale is about ego. Getting scale after nailing behavior change is easier than nailing behavior change (and thus having a shot at durability) after hitting scale.*

It doesn't mean Slow Web companies can't grow. It simply means that they put effectiveness before growth. And effectiveness leads to a sense of gratitude — I may be done flossing with Budge, but there are other things I could improve, and having been through it once, I trust the company even more.

Knowledge vs. Information

Timeliness. Rhythm. Moderation. These things dovetail into what I consider the biggest difference between Slow Web and Fast Web. Fast Web is about information. Slow Web is about knowledge. Information passes through you; knowledge dissolves into you. And timeliness, rhythm, and moderation are all essential for memory and learning.



Again, iDoneThis serves as a fitting example. After you use it for a few days, you start seeing at the bottom of your daily emails the things you've done in the past, a day or a week before. It's kind of a contained version of Timehop [timehop.com], Benny and Jon's product that, once you've connected it to your various social accounts, sends you a daily — get ready for this word — digest with everything you did a year ago on that day.

Timehop and iDoneThis both help us remember and reflect, and this gives us perspective. It grounds us in the flow of time, or perhaps lifts us up above the treetops. iDoneThis is the only task management tool I've come across with the potential to help you realize you're working on the wrong

thing. Fast Web derives value from the just-happened or the soon-to-happen. Slow Web unlocks value from deeper in the past.

The Slow Web

Timely not real-time. Rhythm not random. Moderation not excess. Knowledge not information. These are a few of the many characteristics of the Slow Web. It's not so much a checklist as a feeling, one of being at greater ease with the web-enabled products and services in our lives.

Like Slow Food, Slow Web is concerned as much with production as it is with consumption. We as individuals can always set our own guidelines and curb the effect of the Fast Web, but as I hope I've illustrated, there are a number of considerations the creators of web-connected products can make to help us along. And maybe the Slow Web isn't quite a movement yet. Maybe it's still simmering. However, I do think there is something distinctly different about the feeling that some of these products impart on their users, and that feeling manifests from the intent of their makers.

Fast Web companies want to be our lovers, they want to be by our sides at all times, want us to spend every moment of our waking lives with them. Sometimes that's not what we really need. Sometimes what we really need are friends we can meet once every few months for a bowl of ramen noodles at a restaurant in the East Village. Friends with whom we can sit and talk and eat and drink and maybe learn a little about ourselves in the process. And at the end of the night get up and go our separate ways until next time.

Until next time. ■

Jack Cheng is writer, designer and entrepreneur living in Brooklyn. He co-founded Disrupto, a digital product development studio, and Memberly, a platform for subscription services.

Reprinted with permission of the original author.
First appeared in *hn.my/slow* (jackcheng.com)

Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com

The Cab Ride I'll Never Forget

By KENT NERBURN

There was a time in my life twenty years ago when I was driving a cab for a living. It was a cowboy's life, a gambler's life, a life for someone who wanted no boss, constant movement and the thrill of a dice roll every time a new passenger got into the cab.

What I didn't count on when I took the job was that it was also a ministry. Because I drove the night shift, my cab became a rolling confessional. Passengers would climb in, sit behind me in total anonymity and tell me of their lives.

We were like strangers on a train, the passengers and I, hurtling through the night, revealing intimacies we would never have dreamed of sharing during the brighter light of day. I encountered people whose lives amazed me, ennobled me, made me laugh and made me weep. And none of those lives touched me more than that of a woman I picked up late on a warm August night.

I was responding to a call from a small brick fourplex in a quiet part of town. I assumed I was being sent to pick up some partiers, or someone who had just had a fight with a lover, or someone going off to an early shift at some factory for the industrial part of town.

When I arrived at the address, the building was dark except for a single light in a ground-floor window. Under these circumstances, many drivers would just honk once or twice, wait a

Photo by: Ben Fredericson [flickr.com/photos/xjrlokix/4379281690]

short minute and then drive away. Too many bad possibilities awaited a driver who went up to a darkened building at 2:30 in the morning.

But I had seen too many people trapped in a life of poverty who depended on the cab as their only means of transportation. Unless a situation had a real whiff of danger, I always went to the door to find the passenger. It might, I reasoned, be someone who needs my assistance. Would I not want a driver to do the same if my mother or father had called for a cab?

So I walked to the door and knocked.

"Just a minute," answered a frail and elderly voice. I could hear the sound of something being dragged across the floor. After a long pause, the door opened. A small woman somewhere in her 80s stood before me. She was wearing a print dress and a pillbox hat with a veil pinned on it, like you might see in a costume shop, in a Goodwill store or in a 1940s movie. By her side was a small nylon suitcase. The sound had been her dragging it across the floor.

The apartment looked as if no one had lived in it for years. All the furniture was covered with sheets. There were no clocks on the walls, no knick-knacks or utensils on the counters. In the corner was a cardboard box filled with photos and glassware.

"Would you carry my bag out to the car?" she said. "I'd like a few moments alone. Then, if you could come back and help me? I'm not very strong."

I took the suitcase to the cab, and then returned to assist the woman. She took my arm, and we walked slowly toward the curb. She kept thanking me for my kindness.

"It's nothing," I told her. "I just try to treat my passengers the way I would want my mother treated."

"Oh, you're such a good boy," she said. Her praise and appreciation were almost embarrassing.

When we got in the cab, she gave me an address and then asked, "Could you drive through downtown?"

"It's not the shortest way," I answered.

"Oh, I don't mind," she said. "I'm in no hurry. I'm on my way to a hospice."

I looked in the rearview mirror. Her eyes were glistening. "I don't have any family left," she continued. "The doctor says I should go there. He says I don't have very long."

I quietly reached over and shut off the meter. "What route would you like me to go?" I asked.

For the next two hours we drove through the city. She showed me the building where she had once worked as an elevator operator. We drove through the neighborhood where she and her husband had lived when they had first been married. She had me pull up in front of a furniture warehouse that had once been a ballroom where she had gone dancing as a girl. Sometimes she would have me slow in front of a particular building or corner and would sit staring into the darkness, saying nothing.

As the first hint of sun was creasing the horizon, she suddenly said, "I'm tired. Let's go now."

We drove in silence to the address she had given me. It was a low building, like a small convalescent home, with a driveway that passed under a portico. Two orderlies came out to the cab as soon as we pulled up. Without waiting for me, they opened the door and began assisting the woman. They were solicitous and intent, watching her every move. They must have been expecting her; perhaps she had phoned them right before we left.

I opened the trunk and took the small suitcase up to the door. The woman was already seated in a wheelchair.

"How much do I owe you?" she asked, reaching into her purse.

"Nothing," I said.

"You have to make a living," she answered.

"There are other passengers," I responded.

Almost without thinking, I bent and gave her a hug. She held on to me tightly. "You gave an old woman a little moment of joy," she said. "Thank you."

There was nothing more to say. I squeezed her hand once and then walked out into the dim morning light. Behind me, I could hear the door shut. It was the sound of the closing of a life.

> **We are so conditioned to think that our lives revolve around great moments. But great moments often catch us unawares.**

I did not pick up any more passengers that shift. I drove aimlessly, lost in thought. For the remainder of that day, I could hardly talk. What if that woman had gotten an angry driver or one who was impatient to end his shift? What if I had refused to take the run or had honked once and then driven away? What if I had been in a foul mood and had refused to engage the woman in conversation? How many other moments like that had I missed or failed to grasp?

We are so conditioned to think that our lives revolve around great moments. But great moments often catch us unawares. When that woman hugged me and said that I had brought her a moment of joy, it was possible to believe that I had been placed on earth for the sole purpose of providing her with that last ride.

I do not think that I have ever done anything in my life that was any more important. ■

Kent Nerburn is the highly acclaimed author of over a dozen books on Native American issues and spirituality. The Cab Driver story is excerpted from his book, Make Me an Instrument of Your Peace: Living in the Spirit of the Prayer of St. Francis. It is published here in its original form with permission of the author. Visit his website *atkentnerburn.com* to learn more about Kent Nerburn's life and work and to purchase books.

Reprinted with permission of the original author.
First appeared in *hn.my/cab* (zenmoments.org)

A Profitable, Growing, Useful, Legal, Well-Loved...Failure

By AVERY PENNARUN

Since before graduating from college and up until taking my current job, I've initiated several things that could be called startups. That is, we incorporated companies, we had a small number of people that got paid wages, we collected Canada SR&ED tax credits, etc. Every one of these startups turned a profit and more than one had outside financing. One of them we sold to IBM.

I'm telling you this not to show off, but as a setup for the rest of this story. What I want to explain is that I fail strangely. Or at least, it feels like I do. Maybe it's not so strange; maybe you should just go read Paul Graham's How Not to Die article [hn.my/die], where he advises us that "Startups rarely die in mid keystroke. So keep typing!"

Because that's really the moral of this story; or maybe it isn't. Maybe this story is about how that advice hasn't actually worked for me because inside each of those successes is a story of failure. It's interesting that by leaving out some details I can honestly make any one of the companies I've started sound like resounding successes or resounding messes. If I include all the details, then, well they're just confusing. So you'll usually hear just one side or the other, depending what point I'm trying to make.

Today I'll tell you both sides though for just one of those companies. I'm not going to name the company here but it's still alive, it's still making money, and my co-founder is still working his butt off to keep it from falling over. Given the details I'm about

to share, it's trivially easy to find the company name with a little Googling, and I encourage you to do so. I just don't want to name it here because I really don't want this article to be the first one that comes up when you Google it.

So anyway, here's what happened. We started the company back in 2008. We wanted to do something in the world of databases because we figured databases were ripe for disruption, what with SQL being SO VERY SUCKY in so many ways. We wanted to create a new variant of SQL based on the analogy that our "new thing" is to SQL as C is to assembly language. That is, C is little more than a portable assembly language, so we need a portable version of SQL. (If you've used more than one SQL variant, you know the analogy is apt.) Oh, and maybe we'll throw in functions and variable assignment and loop control structures while we're there. Yeah, I know, crazy. But if you've written stored procedures in MS SQL, those are the things you know you need.

Why did we want the C of database query languages, instead of something modern, like the Python of database query languages? We thought this was the clever part of the analogy: it's because people already *tried* the high-level query languages. They're called ORMs (object relational mappings), and sure enough, they're just like high-level languages were in 1975: slow, bloated, wasteful, unreliable, non-portable, and nobody can agree which one is best. C changed all that. Sure, there were non-portable features in C (there still are), but dammit, + was just always +, and for loops were for loops, and the world made one big step forward. People still use C today. High-level languages are much better now, but they're almost all still built on top of C. How much better could the world be if we could do that for SQL?

Anyway, that seemed really hard, and we were just two guys who wanted to get a minimal product launched in, say, 4 months. So we decided to trim down the idea. What's the minimal idea that would get us in that direction, but with a product in 4 months? Well, first of all, to invent C you don't need multiple assembly language variants; you just need one to start with. Let's pick one. Why not the simplest one we can find? A bit of searching around revealed the obvious candidate: Microsoft Access. It's even dumber than MySQL.

Okay then, what will we build on top of Access? Well, we want to make a portable, slightly-higher-level query language. What will be its initial use case? Forgetting about other databases for now, what do Access developers need most? … Ah, to publish their data on the web, of course. Access totally sucks for web development. Even now it does. They keep claiming to have

finally added web support, but it's nearly useless every single time. Still is.

So we would write code to let you easily query Access tables using web tools, like AJAX or JSON or whatever. Excellent, that justifies writing our query parser, but it doesn't have to be feature-complete on day 1. We can add more database engine plug-ins later. We can get a few customers, launch, and iterate. Perfect!

Just one little problem. You have to actually get that data to the web server. Access sucks for web apps because Access databases are a single .mdb file on your desktop machine. Multi-user access means multiple clients accessing the .mdb file using a samba file share. But how do you get the data onto the web?

Well, the .mdb file format is undocumented. Reverse-engineering it will take forever, so we figured we'd write a plug-in for Access that reads through your data, exports it to text, and uploads it to our server. That turned out to be a fair bit of work, of course, but whatever. I do love replicating data, and we figured the ability to replicate SQL databases could be a big deal, so it's certainly not a waste of time.

Once we were well under way writing the replication system, we thought about it some more and realized that the minimal product for our 4-month launch target didn't have to include a query language at all; just replicating the databases was surely enough to please some user somewhere, as long as it would sync in two directions. Ta-da, Internet-enabled Access replication! We stopped after writing only the barest minimum query parser.

We got the basic Access web replication engine working, which was a huge amount of work, don't get me wrong, and the code is singularly awesome, but I'm going to skip over it here. We gave it a convincing-sounding version number with the word BETA in it, put it up on a web site I designed with my super lame web design skills, and waited for the world to beat a path to our door.

Okay, you know how this goes, right? You can't just do that. Nobody will come.

Well, this time you're wrong. People came. We had stumbled onto a huge unsolved problem and unaddressed market. There are lots, and lots, and lots, and lots of legacy Access databases in places you don't even want to think about. If you find our web site and go to Testimonials and scroll to the bottom, you'll see what I mean. The actual CIO of a huge pharmaceutical company called us out of the blue and asked us to solve their problem because they had thousands of Access databases they wanted to share across their tens of thousands of seats.

But I'm jumping ahead of myself. Not all those people called us on day 1.

On day 1, our website sucked because it was talking about Access Replication.

And what the bloody hell is replication? Most Access users with Access problems didn't have a clue. They certainly weren't searching for it.

That didn't stop some of them from finding us and calling anyway. See, we also had a couple of pages talking about our query engine, and they contained phrases like "Access on the Web." Turned out a lot of people were searching for that. They still are. Microsoft caught on with Access 2010 and marketed the heck out of that search phrase, so if you search for it now, you'll find them and not us, which is funny because Access 2010 is still basically useless for the web. But it shows what marketing dollars can do.

Now, I'm badmouthing Access 2010 a lot here, but here's how I know it's useless: because people keep on clicking and searching, and I don't even know what keywords they search anymore, but they keep finding us. They use Access 2010. They're not dumb; they're real programmers and they know what features Access 2010 has. Even if they were dumb, God knows Microsoft has marketed them to death. And these people still want to pay us to put Access on the web.

Anyway, I've gotten ahead of myself again. The important part of the story is, we had a web site all about Access replication and nobody had any clue what we were talking about, but they called and emailed and the message was clear: We want Access on the web. How much money can we pay you to provide it?

Um, well, look, the on-the-web part is kind of sucky and...

...and the customer is always right. So, back to the drawing board. One day, a customer called me and explained his very specific and immediate problem. He had just billed a customer many thousands of dollars over many months to build a custom Access application. Right at the end, the customer said they were happy. Now...he should just publish it on the web and they'll be done.

Oh. Crap. The guy was really in trouble. Serious trouble. They hadn't specified the requirement up front; he was an Access-only developer, so he couldn't rewrite it. Even if he knew how, it would be months more work. So he had a serious problem, and let me tell you, our 5%-finished JSON query language was not going to solve it. Neither was "replication," but that day on the phone, we came up with an idea.

What if we could run Access on our servers and display it over VNC in a web browser? What if we ran Access under Wine on Linux so we could squeeze more instances onto a single box? What if changes to the database in these VNC sessions could be replicated back down to your desktop copy of Access using our plug-in?

> **“The customers need it because nobody else has ever created something like this. I don’t think anybody ever will.”**

What if, indeed. Turns out there’s a cool program called Flashlight-VNC that’s an implementation of VNC in Flash, which runs in virtually any web browser (this was before there was an iPad and before Apple dropped Flash out of Safari). Turns out recent versions of Wine can actually run some versions of Access. Turns out…well, let’s just say it worked. And that, my friends, is the product we have today, more or less. Sure, since then we’ve added performance optimizations and reliability improvements. We store the database contents in git and use a custom merge algorithm for resolving changes made while in disconnected mode. (It’s neat; git can store the whole revision history in less space than the original .mdb.) But fundamentally, that’s the product.

And people want it. No, I take that back; the product is a magnificent heap upon heaps of insane hackery. I mean, we are running Access in Wine in X11 on Linux in an isolated user account on our server slice that revision controls your Access database in git, and we’re displaying it using VNC in your web browser in Flash. People can’t possibly want that, but they need it, which is better.

That’s the other neat thing. They need it because nobody else has ever created something like this. I don’t think anybody ever will. I mean, how many people know Linux, Flash, C++ (for the plug-in), Python (for the server), and Microsoft Access, of all things, and are willing to combine them all with a healthy knowledge of streaming network protocols and database replication? And even if you could find a whacko like that, would that person be willing to enter the market, starting from scratch, knowing someone else got there first?

Every month, we have more revenue. And our costs are tiny, so that means more profit.

> **“Eventually I realized that there is no windfall big enough to rationalize spending 3-5 years of my life writing compatibility layers for Microsoft Access.”**

Customers need this so badly that they're willing to pay a lot for it. Like $35/user/month/database for the basic plan. In case you're counting, in a year, that's much more than a copy of Access. And just to be safe, because we want to avoid lawyers, we tell customers to make sure all their users already have an Access license on their desktop (in addition to the legally required ones we have for our servers). This isn't so bad; turns out big companies — the kind with lots of Access databases — pretty much all buy Microsoft Office Professional for everybody anyway. So no, in case you were wondering, our business model is not about cheating on Access licensing. If anything, people are buying more licenses than they strictly need, and I don't feel like getting on Microsoft's bad side, and neither do they, so everybody wins.

No, it's not about cheating. It's just about providing something people want and are willing to pay for. What do they want? They want to not rewrite legacy apps. Please, please, let us just keep running the app we spent the last 10 years building, but let us run it outside our office because we all have laptops now.

How much money will people pay to keep their app going? About as much as the cost of rewriting it in a web language. More, even, since it lowers their risk. You do the math. As a bonus, it's a small monthly expense, not a big capital expenditure.

And yes, every month, our profit is more than the last one.

But all that was the good news. I've already given you a hint about the bad news. Remember when I asked what whacko, with all those skills, would want to do this? I now know one of the answers, and it's OH GOD NOT ME. Eventually I realized that there is no windfall big enough to rationalize spending 3-5 years of my life, working full time, writing compatibility layers for Microsoft Access. In the ideal world, if we were successful, my days would involve on-site visits to huge bureaucratic companies of the sort that...well, let's be honest, the sort that would run mission-critical Access databases.

Really, on a rational level, I know that's unfair. I know these are good people. I think Access developers are great, actually. I love the fact that they know a good thing when they see it. Access *is* the easiest, most rapid of rapid development environments I've ever seen. I think almost all database developers have terrible tast, because they can use Access and compare it to, say, MS SQL, and not see what makes Access great and MS SQL suck, even knowing perfectly well the development in MS SQL + C# or Java will take something like 10x as many man-hours. For some apps, it's worth it for the higher quality; for a random internal business process app, it's not, but people spend it anyway because they "heard Access isn't industrial strength."

So don't get me wrong. I like Access users. Access developers, in particular, are the anti-IT department, the rebels, the people who aren't willing to wait for the system administrators to provision them a server, and they don't have to because they can just share an Access file on the fileserver. IT departments hate them, which is how I know they're on to something. These are the kind of people I want to help. This is the sort of thing that's the reason I do the work that I do. No kidding.

But, Lord, no, don't make me actually code Access plug-ins. Don't make me work with Windows anymore. Just don't.

It's so lame when I write it down. Actually, it's been lame for months, every time I even think it. I can't believe I have that kind of lack of follow-through. I don't want to think that about myself. It's a travesty. A terrible embarrassment. Something that makes me question my self-worth. If I can't take something that's so obviously working, and milk it for all it's worth, then what kind of human am I, anyway? I think I suck at capitalism. Maybe that's it.

You know the truth? I don't know. I just don't know. I am a completely irrational human being, and I hate it, but deep inside me there's a voice that just says, "No. Get the hell out. If you continue doing this, you will die."

So I got the hell out. I "stopped typing," as Paul Graham might say. Nowadays I have a pretty great "real job" where I can spend all night hacking the Linux kernel, programming embedded systems, and working on highly parallel build systems. And even though the potential upside is much less, I like it. For now, at least. I'm happy.

And that's my failure. Every day, my co-founder keeps working away, keeping the systems running with as little effort as he can spare. He's got a day job now for various reasons; among them, he's an extravert and he needs co-workers. I still own half the shares, but I told him to keep the operating profits; the least I could offer, literally, I guess. That huge pharma deal is still in the pipeline and needs another callback, but there's nobody willing to do it. We don't optimize the web site for Google anymore; we haven't updated the news page since 2010 and even I can't find our site in Google using any generic keywords. I guess I'm not looking hard enough because new customers still find it, sign up, and subscribe. Virtually nobody ever cancels once they've started. There is no competition and nothing to switch to. There never will be. Where would they go if they stopped?

I know I've let my co-founder down. If the company would just die — if it would only be so simple, and nobody would want the product, or the users got angry at us and quit, or it were impossible to run it at a profit and we finally ran out of cash — then stopping would be easy. But no. They love it instead. They need it. There's an opportunity cost in continuing, but there's a sentimental cost in shutting it down — to say nothing of the users who have no other options.

In short, I learned that I don't have what it takes. Someone probably does, now that the actual insane part has already been invented, but I don't know who.

What would you do? ■

Avery founded his first startup, Nitix, making Linux-based server appliances while at the University of Waterloo. The company was acquired by IBM and is now called Lotus Foundations. He wrote wvdial, netselect, git-subtree, sshuttle, bup, and redo, and now lives in New York.

Reprinted with permission of the original author.
First appeared in *hn.my/fail* (apenwarr.ca)

How To Scale a Development Team

By ADAM WIGGINS

As hackers, we're familiar with the need to scale web servers, databases, and other software systems. An equally important challenge in a growing business is scaling your development team.

Most technology companies hit a wall with development team scalability somewhere around ten developers. Having navigated this process fairly successfully over the last few years at Heroku, this post will present what I see as the stages of life in a development team and the problems and potential solutions at each stage.

Stage 1: Homebrewing

In the beginning, your company is two to four guys/gals working in someone's living room, a cafe, or a coworking space. Communication and coordination is easy: with just a few people sitting right next to each other, everyone knows what everyone else is working on. Founders and early employees tend to be very self-directed, so the need for management is nearly non-existent. Everyone is a generalist and works on a little bit of everything. You have a single group chat channel and a single all@yourcompany.com mailing list. There's no real need to track any tasks or even bugs. A full copy of the state of the entire company and your product is easily contained within everyone's brain.

At this stage, you're trying to create and vet your minimum viable product, which is a fancy way of saying that you're trying to figure out what you're even doing here. Any kind of structure or process at this point will be extremely detrimental. Everyone has to be a generalist and able to work on any kind of problem — specialists will be (at best) somewhat bored and (at worst) highly distracting because they want to steer product development into whatever realm they specialize in.

Stage 2: The first hires

Once you've gotten a little funding and been able to hire a few more developers, for a total of five to nine, you may find that the ad-hoc method of coordination (expecting to overhear everything of importance by sitting near teammates) starts to break down. You have both too much communication (keeping tabs on six other people's work is time-consuming) and too little communication (you end up colliding on trying to fix the same bug, answer the same support email, or respond to the same Nagios page).

At this point, you want to add just a sprinkle of structure: maybe an iteration planning on Monday, daily standups, and tracking big to-do items and bugs on a whiteboard or in a simple tool like Lighthouse. Perhaps you switch to a support system like Zendesk where incoming support requests can be assigned and you add a simple on-call rotation for pages via Pagerduty. Your single internal chat and email channels continue to work fine.

Resist the urge to introduce too much structure and process at this point. Some startups, on reaching this stage, declare "we've got to grow up and act like a real company now" and immediately try to switch to heavy-handed tactics. For example: full-fledged SCRUM, heavyweight tools like Jira, or hiring a project manager or engineering manager. Don't do that stuff. You've got a team that works well together in an ad-hoc way. You probably have some natural leaders on the team who direct a lot of the work while still being hands-on themselves. And while your product is launched and in the hands of users, in many ways you're still trying to figure out what your company is really all about. Introducing bureaucracy into this environment is almost guaranteed to block you from doing what you're really supposed to be doing, which is pivoting in search of your scalable business model.

Focus at this stage is key. Everyone is still a generalist, but the whole development team should be aligned behind a single goal (aka milestone) at a time. If you try to attack multiple battlefronts at once, you'll do everything badly. Great companies are more likely to die of indigestion from too much opportunity than starvation from too little. Pick your battles carefully and stay focused.

Crisis on the brink of Stage 3

Grow to 10-15 developers, and you're on the verge of a major team structure change. I've been told that many promising startups have been killed by failing to weather the transition between these stages.

With this many developers, iteration planning, standups, or any other kind of development-team meeting has become so big that the attendees spend most of their time bored. Any individual developer will find it difficult

to find a sense of purpose or shared direction in the midst of trudging through laundry lists of details on other people's work.

In programming, when a class or sourcefile gets to big, the solution is to break it down into smaller pieces. The same principle holds for scaling a development organization. You need to break into targeted teams.

Stage 3: Breaking into teams

Dividing your single team of generalists is harder than it sounds. Draw the fences in the wrong place, and you'll create coordination problems that make things even worse. Find the right places to divide, and you'll see a massive increase in focus, happiness, and productivity.

The key to a good team is a well-defined sphere of authority, with clear interfaces to other teams. The team should own the vision and direction for the part of your product that it works on. It should be able to operate with maximum autonomy on everything it owns without having to ask for permission or information from other teams, except for the infrequent case of a feature or bug that crosses team boundaries.

A close mapping between your software architecture and your team architecture will be a big help here. By this time you have probably already converted your monolithic application into a distributed system of multiple components communicating over REST, AMQP, or other RPC mechanism. (And if not, you should strongly consider doing so, coincident with your dev team split.) There should be an obvious mapping between software components — each of which has their own source repository and deployment location/procedure — and your nascent teams.

Deciding what person goes on what team will be somewhat arbitrary at first. My approach was to sit down with each developer and dig in to understand what parts of the system they were most passionate about working on. From there I divided up the teams as best I could. Some people found perfect homes on their first team assignment; others were dissatisfied and needed to transfer to another team fairly quickly. Over time, the team territories became very well-defined, so it became much easier to slot new hires in the right place. Let developers follow their own passions and they will gravitate toward the team where they will do the best work.

Separately, you should have found your product/market fit by this point. If you've grown to this size and are still figuring out your company's meaning for existence, you've got big problems. If that's the case, stop growing and scale back down until you nail the product/market fit.

Specialization

Another reason to break into teams is specialization. Types of engineering specialists include ops engineers/sysadmins, infrastructure developers, front-end web developers, back-end web developers, business engineers / data analysts, and developers who focus on a particular language. Language specialists are becoming more common, because many internet-scale companies write high-concurrency components in functional programming like Erlang, Scala, or Clojure, generally handled by a different set of developers than the authors of the Ruby, Python, or PHP web components.

Early on, specialists are rarely desirable. There are too many different layers to work on in delivering a software product relative to the number of people available to contribute, so everyone pitches in on everything. This may put a developer doing such far-ranging work from projects like kernel updates on the OS to front-end projects like writing JQuery effects for the UI.

Once you reach the point where you've got a dozen developers, your product has reached a level of usage and maturity where the problems are getting much harder. Scaling the database is something that is not only a full-time job, but requires a deep level of specialized knowledge that can't be acquired if that person is also simultaneously learning to be a JQuery expert and an iOS expert and an Erlang expert.

You need people who can and are willing to focus on just a few closely related areas so that they can build very deep knowledge in those areas. Some of these will be your existing generalists deciding to specialize, and some will be new hires. You can now hire for the kind of specialist that would not have been appropriate when your company was smaller. Generalists are always useful to have around, and some of them may move into management — filling business owner roles for a team, rather than hands-on development.

Heroku's first teams

Heroku's initial team breakdown looked like this:

- API: Owns our user-facing web app and the matching Heroku client gem.
- Data: Builds and runs our PostgreSQL-as-a-service database product.
- Ops: Shepherds and protects availability of the production system.
- Routing: Manages everything necessary to get HTTP requests routed to user web processes.
- Runtime: Handles packaging code for deploy and starting/stopping/managing user processes.

Each of these teams owns between one and five components. For example, the API team owns the Rails app, which runs at api.heroku.com, and the Heroku client gem. The Data team owns the provisioning and monitoring tool for our database service, as well as all of the individual running databases.

Team size and roles

For us, the ideal team layout has been two developers and one business owner. One developer is not enough over the long term (they need a second pair of eyes on the code, and besides, one is a lonely number). Three developers works fine as well. Get to four or five and things start to become a bit crowded; there may not be enough surface area for them to all work without stepping on each others' toes constantly. Almost all of Heroku's teams have two developers.

"Business owner" is a somewhat clumsy term, but it's the best we've come to describe the person doing some combination of product management, project management, and general management for the team. The business owner fills the important role of knowing the business value of the team's work to the company and how it fits in with the larger product. They can broker cross-team communication, help prioritize projects and tasks by business value, and may provide status reports on the team's progress or presentations to the senior executives and/or the entire company to justify the team's ongoing existence.

I'm a fan of hacker-entrepreneurs in the business owner role: a strong technical background means they have an in-depth understanding of the work being done and are able to command huge respect from those whose work they are directing. This sort of person is not necessarily available for all teams, but find them when you can. In many cases it involves quite a bit of convincing to get a hacker to give up coding as their primary function.

Avoid having developers belong to more than one team. They are makers and need to be able to focus their full attention on their team's current projects without distractions or attempts at multitasking. Business owners, however, can sometimes belong to multiple teams. It's not always a full-time job, and there are benefits to cross-team communication by having one person as the business owner for two or more related teams.

Cohesion

In the earlier stages, you should avoid attacking on multiple battlefronts, and instead keep all developers focused on a single goal for the company. With creation of fiefdoms for each team, this has changed. Now you can and should attack on multiple battlefronts. Each team should be executing independently against its own goals and not worrying too much about what other teams are doing.

It's awesome to be able to pursue three, four, five big goals simultaneously. A few months after breaking into teams at Heroku, we had a day where three different teams were all releasing major new features. It's an incredible feeling.

But now you have a new problem: lack of cohesion. Your decentralized teams are setting their own roadmaps and deciding on features independently. To avoid fragmentation in your product, someone needs to decide an overall direction and set of product values. More succinctly: you need a strategy. ■

Adam is a hacker, technology entrepreneur, and occasional rapscallion. He's co-founder and CTO at Heroku, and author of The Twelve-Factor App [12factor.net]. Follow his work via *adam.heroku.com* or @hirodusk

Reprinted with permission of the original author.
First appeared in *hn.my/devteam* (adam.heroku.com)

The Recruiter Honeypot

By ELAINE WHERRY

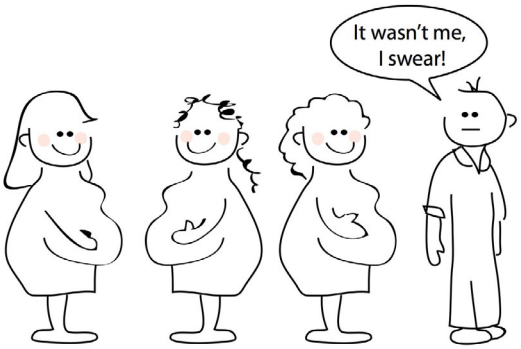
In late 2009, I created an online persona named Pete London, a self-described JavaScript ninja, to help attract and hire the best JavaScript recruiters. While I never hired a recruiter from the experiment, I learned a ton about how to compete in today's Silicon Valley talent war. Based upon two years of non-scientific research, here's what you should know…

The recruiting crisis

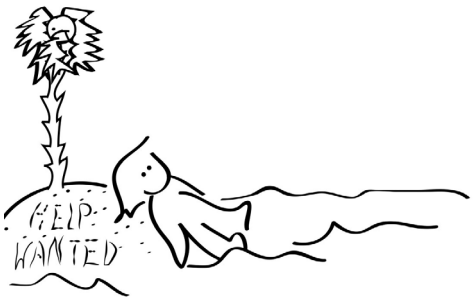
In late 2009, my desk was piled with JavaScript resumes. Our homegrown JavaScript framework edged us over competitors but maintaining our technical advantage meant carefully crafting a lean, delta-force Web team. Though I averaged two interviews a day, we had only grown the team by three or four engineers each year.

However, in 2010, that had to change. It was our first year with a real revenue target and also the first time we planned to pivot from our original IM product. We charted our end-of-year goals and quarterly milestones, and we eventually backtracked to our team and hiring priorities. To meet our 2010 goals, I needed to double the JavaScript team in just one quarter. If I didn't, innovation would stall and without revenue, our business would be in serious jeopardy.

I had very little more to give. Over the previous four years, I had already spent my personal networks, seeded every nook of the Web with job descriptions, and experimented with guerilla recruiting tactics like hosting JavaScript meetups across the country, planting hand-written congratulatory notes on the seats of CS Stanford students who'd just finished their finals, coding a spidering engine to find online JavaScript resumes, and even buying Google AdWords for relevant terms like xmlhttp, opendatabase, and localstorage.



But then my recruiting problem went from serious to heart-stopping dire. In the final months of 2009, every female on Meebo's recruiting team became pregnant within a month of each other. Our expectant mothers were searching for contract replacements, but as winter crept closer, finding someone who could temporarily step up to our extraordinary JavaScript challenges during our most critical hiring quarter looked unlikely. I was truly on my own.



Pete London is born

I desperately needed amazing recruiters. After the third expectant mother relayed her good news, I sunk into to my chair overwhelmed with urgency and stared blankly at my monitor thinking over and over, Oh my god, what do I do now? My first impulse was to look at the recruiters in my Inbox — specifically those who had pinged me for a JavaScript role and presumably had prior JavaScript recruiting experience. However, I also needed a recruiter who was smart enough not to poach a founder.

The honeypot idea emerged slowly, If only I weren't a founder! Which recruiters would have contacted me as an engineer? I stewed on the idea of posting my resume online with a fictitious name for days. Then one sleepless night, without telling anyone, I woke up and posted a small three-page website [petelondon.com] with an about page, resume, and blog for a supposed Pete London whose interests and engineering persona mirrored my own except he wasn't a founder. I swapped out my post-graduate experience with my husband so it wouldn't be too easy to trace back to me. I returned to bed with a small glimmer of hope — I had been hunting for recruiters for months, but now the recruiters would come to me!

Last resort — Linkedin

My hopes sank pretty quickly. PeteLondon.com sat alone in Internet ether for weeks with absolutely nada activity. I was about to pull down the entire site when I thought, I'll just post the resume on LinkedIn as a last resort.

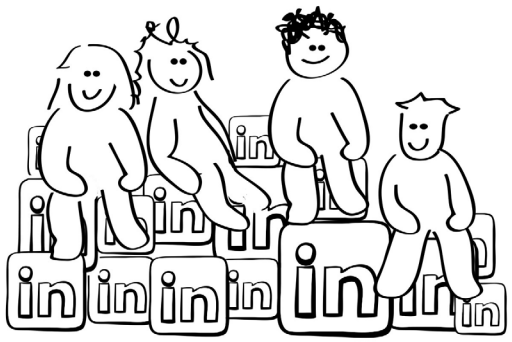
Bam. It was as if I'd finally stumbled upon the door to the party.

On December 10th, 2009, the first LinkedIn message arrived from Google. Mozilla followed on December 15th. Ning and Facebook followed in January. Since then, Pete averaged a recruiter ping every 40 hours and saw 530 emails from 382 recruiters across 172 organizations.

What I learned

After two and a half years, I learned less about recruiting recruiters and more about recruiting engineers. Here are my eight biggest take-aways to finding the best talent online…

Lesson 1: Recruiters rely exclusively on Linkedin



You might be thinking, Really? This is obvious! But understand the context. I was interviewing tech recruiters who said they had "moved beyond LinkedIn." LinkedIn was a "crutch for everyone else" but them. When I asked what techniques they used to fulfill JavaScript roles, they'd describe complex Boolean queries, highway 101 billboards, and obscure search engines. I ate it up! But at the same time, I wondered, Wait, if this is all true, why hasn't anyone found Pete London yet?

To further my confusion, LinkedIn wasn't how Meebo found its initial superstar JavaScript team. From 2005-2011, only one JavaScript team member was hired via LinkedIn — the rest came from personal networking, meetups, blog scouting, and other guerilla recruiting approaches.

I also assumed that a professional who made their living from recruiting would want to optimize their response rate and would seek out ways to contact

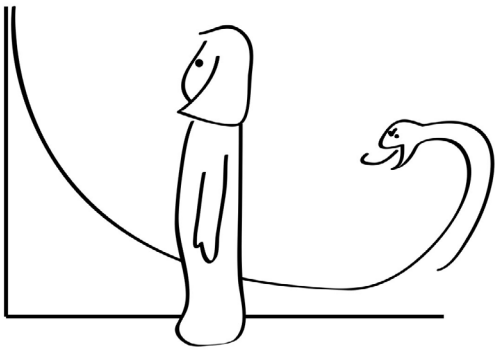
Pete London beyond LinkedIn. Though Pete London's website and personal email address were just one click from his LinkedIn profile page, the majority of emails still arrived via LinkedIn — especially from larger companies.

Surprisingly, very few recruiters tried more than one communication channel.

TIP #1: *If you're a start-up who always feels like you're scraping the bottom of the LinkedIn barrel, you're probably right — LinkedIn is incredibly competitive. Recruit latent talent off the grid.*

TIP #2: *Recruiters usually flock to LinkedIn first, if not always. To increase your personal opportunities, join LinkedIn.*

Lesson 2: Fear the Silicon Valley long tail



When I wrote to potential engineers, I always imagined my email landing next to recruiting giants like Google or Facebook. As a result, I was careful to emphasize Meebo's unique start-up learning opportunities, amazing culture, and the opportunity to make impact.

However, my strategy was misguided. The Silicon Valley companies that drew TechCrunch headlines from 2010-2012 (i.e. Adobe, Amazon, AOL, Apple, Facebook, Google, LinkedIn, Netflix, Microsoft, Mozilla, Skype, Twitter, Yahoo, Zynga) only represented 15% of the landscape.

But I should have been more scared than I was — the emails from start-ups and mid-sized companies sounded nearly identical (my own included): "We're a fast-growing start-up disrupting a lucrative space where your talents will shine and your efforts will be amply rewarded." By emphasizing the classic start-up experience, everyone sounded exactly the same:

Start-up in Mountain View: "We've assembled a world class team. Our monthly uniques have already exceeded [###] million and continue to trend higher at a rapid pace. We've reached an inflection point where we're looking to scale, and with your background I wanted to speak with you about our engineering hiring."

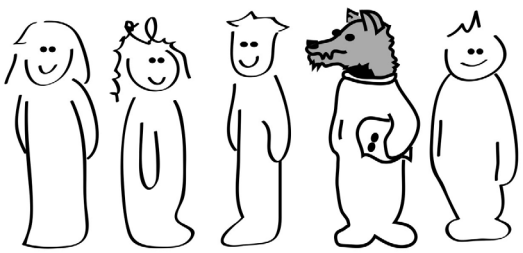
Start-up in San Francisco: "There are a variety of interesting technical challenges in front of us including scaling for millions of users, developing applications, building a sophisticated data platform, securing user data and, most importantly, ensuring an incredible experience for our users. Aside from our plethora of awesome technical projects, this is also a great place to work. Everyone on the team benefits from free meals and tremendous organizational transparency (weekly all hands, daily stand ups, etc.)"

Larger companies employed an entirely different strategy and anecdotally, I saw terser, canned emails from larger companies than start-ups. To quantitatively compare strategies, I went through all emails and noted whether the recruiter included role details, company information, or if the email was personalized specifically to Pete. I was incredibly lenient and gave points whenever I could. By almost every metric, the larger companies performed weakest: smallest word count (114 vs. 148 words per email), least likely to describe the company mission or personalize email, and least likely to use a personal email address. However, large companies hired triple the number of recruiters and made up for their shortcomings in volume. Pete heard from an average of 1.4 recruiters at each start-up and 4.6 recruiters at each large company.

You might assume that with more internal recruiters, big companies would do better than start-ups that depend more on external recruiters. After all, big companies have had more time, resources, and infrastructure to make this a key strategic asset. But it turns out you don't want to emulate the big guys and you also don't want to assume they are your stiffest competition.

TIP #3: *Your real recruiting nemesis is the start-up down the street. Pitch your job opportunities with more specificity than "fast-paced, innovative startup."*

Lesson 3: The recruiting landscape isn't just filled with recruiters



Only 97% of the recruiting emails can be attributed to traditional recruiting. So who represents the remaining 3%?

Surprise! VCs — specifically early-stage angel investors.

Though they are a small lot, they are a super lethal bunch with an eye on your jugular artery — your revered first engineers who built your system from scratch. The charming VCs know that your prized engineers could fulfill a similar role at their future portfolio companies and set their hooks early. In most cases they don't have a specific company or role in mind but are just proactively networking and hoping to be top-of-mind in the future. Given how interconnected and fast-moving the start-up world is, this might be inevitable but woah! good to know.

> *"I'm with [a VC firm] and my charter is to build out their talent services capabilities. What that means is we are looking for high caliber individuals that would be interested in potentially exploring opportunities with our portfolio companies.*
>
> *Your experience is exceptional and you have the type of background that should be a part of the network. If you are interested in learning more I would love the opportunity to speak with you in more detail. What we are looking to establish is a "go to" network of top notch individuals that would be a value add to our portfolio of companies. I hope to hear from you soon."*

TIP #4: *Keep your engineers happy (i.e. free food, great people, & amazing challenges). When the VCs come knocking, make sure your MVPs are glued in.*

Lesson 4: Can a start-up rely on external recruiting?

As a start-up, you are inevitably resource-starved. When you have the good fortune to gain traction, you have the setback of suffering infrastructure growing pains while realizing the only way to get ahead is to find time to recruit, interview, and close candidates. In the early days, external recruiters appeared on Meebo's doorstep and promised to screen and pass along

qualified candidates so I could turn my attention back to Friday's release — it seemed like a dream come true!

However, the first people you hire set your engineering and cultural DNA for the lifetime of the organization and while you desperately need to hire well, can you depend on external recruiters to step up to the task? Once the scaling challenges strike, does it make more sense to proactively hire a superstar in-house recruiter or to rely on external recruiters to scale the engineering team?

The answer is surprising — external and internal recruiters perform similarly in start-up environments. Internal recruiters are 14% more likely to describe the position but 14% less likely to personalize the email.

However, larger companies don't have a viable external recruiting option. External recruiters at the top companies were much weaker overall — 340% less likely to include a description of the role, 140% less likely to personalize their email, and 88% less likely to include detailed company information. Though larger company recruiters were relatively weak overall, in-house recruiters are their only viable option.

Given this significant performance difference, it's no surprise that larger companies also employ far more internal recruiters than start-ups.

TIP #5: *As a start-up, you can sleep easier knowing that external recruiters are a fantastic resource. Find your superstar engineers first and your superstar in-house recruiters second.*

TIP #6: *Contingency recruiting firms are financially incentivized to hire for less selective companies. For difficult roles, a dedicated contract recruiter may be your only realistic option.*

However, before you get too excited about external recruiters, read further…

Lesson 5: Be careful whom you invite into your house

Unfortunately, it's not all about the numbers. Though external recruiters perform well for start-ups, there's another side to this story. It pains me to write this, but I think it's important to share…

Meebo employed lots of external recruiters when we were getting off the ground. We had standard 18-month no-poach restrictions with all of our contractors specifying that those

recruiters were not allowed to contact Meebo employees within 18 months of our contract expiring. Most of those contracts expired in 2008-2009.

However, every recruiter and firm we'd worked with who was still in the recruiting business tried to poach Pete London.

Every single one!

It's impossible to know whether our former recruiters were pinging employees during the no-poach period prior to 2009 but I wouldn't be surprised. However, I doubt they were being malicious — it's more likely they were just disorganized and didn't communicate an off-limits list to their staff.

In addition to pings from too-familiar recruiters, there were two cases that left me especially uneasy. In the first case, a former recruiting agency tried to poach Pete London and then 15 minutes later, wrote to me offering recruiting services! I was being pulled on both ends! When I didn't respond, they repeated the stunt again six weeks later. I got wind that they'd sent recruiting emails to everyone on our Engineering teams and I called them on it (without referencing Pete London). I never heard from them again.

May 13th, 2:20pm

"Hi Peter,

I am a recruiter who works with high-growth, top-tier start ups and industry leaders. I came across your information and was impressed with your background. I'm guessing you may not be actively looking for a new job right now, but I'm sure you plan on continuing to advance your career in the long term, and would be open to hear about opportunities that may accelerate that advancement.

I'd like to get a better idea of your interests and goals, so that I can identify and present to you a few of the most attractive opportunities in the market both now and in the future. You may be pleasantly surprised at what is out there for you. Let me know a good time and number to call you…"

May 13th, 2:35pm (15 mins later)

"Hi Elaine,

I'm a recruiter… We specialize in the placement of technology professionals. I've been working with many excellent candidates from the space and researching companies for them. meebo came up in my search as a good company to consider, so I'd like to present some of these candidates to you for interviews.

Please call me or email me a good time and # to reach you…
Thanks and I look forward to working with you!"

The second case that made me uneasy involved a contractor recruiter who worked from Meebo's office for nearly a year. During this time, the recruiter went to lunch with the team, participated in hackdays, and became close with many folks. Two years later, that recruiter poached Pete London and a few hours later, showed up at Meebo's informal Friday happy hour! I was definitely in a queasy gray zone where there wasn't a strong divide between our personal and professional relationship. Technically, it was hard to nail down any real grievances, but I was certainly aware that our teams were constantly under former recruiter attack.

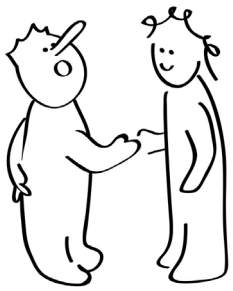
External recruiters are an inevitable necessity for start-ups. But after seeing all of the emails that those external recruiters generated in subsequent years, I wish Meebo had switched to in-house recruiting sooner.

The external recruiters you work with today are good, but they will learn your strengths and your team, and you'll probably be uncomfortably top-of-mind later on.

TIP #7: *External recruiters are a mixed blessing — be selective and switch to internal recruiters as soon as you can.*

TIP #8: *Push for at least 18-month no-poach policies with external recruiters.*

Lesson #6: The most common little white lie is…



With very few exceptions, recruiter emails were well-written, smarmy-free, and didn't smell of phishing. I expected far worse. However, if a little white lie is going to sneak into an email, it's going to look like this…

"I was referred to you as a possible source for a position I am working on here" – Large company

"I previously worked with [Bob] & [Andrew] and have heard great things about you and feel you'd be a great fit…" – Startup

"I understand that you may not be actively looking at this point, but we have heard that you are very good and wanted to see if you might consider looking into a position with [us]" – Startup

"I'm reaching out to you because I've been an admirer of your work at Meebo and believe you could be the perfect founding engineer to lead front-end engineering for our product." – Startup

Little white lies appeared across all recruiting groups and generally took the form, "I was referred to you" or "I've heard very good things." While even unfounded flattery feels good, I learned to be suspicious of vague recruiter compliments.

TIP #9: *Flattery will get you everywhere! Take recruiter praises with a healthy pinch of salt.*

Lesson #7: It's time to buy more hoodies

If you are a JavaScript engineer, you know that the talent market is increasingly competitive and you are inevitably feeling the pull of San Francisco. The demand for engineers has intensified over the last two years and recruiting activity has exploded in the foggy north.

It's impossible to ignore the momentum that is growing in San Francisco. If I were a start-up getting off the ground today, I would start in San Francisco. In 2011, Meebo saw more of its JavaScript engineers hailing from SF than from Mountain View for the first time. While it's exciting that there are more geographic options to start a tech company, it's also time to recognize that companies need strategies for geographically dispersed teams and for recruiting from different areas of the Peninsula.

TIP #10: *As the city of Palo Alto or Mountain View, I would make sure that resident tech companies are happy and that public transportation is a top priority.*

TIP #11: *When writing to candidates, specify where your office is located — it's no longer assumed that an opportunity is south of San Mateo unless otherwise specified.*

TIP #12: *The entrepreneurial epicenter is no longer Palo Alto. If you're south of San Mateo, figure out your SF strategy now.*

Lesson #8: Who's the best in the valley?

You are.

There were 19 emails from managers, execs, founders, and board members who presumably had no professional background in recruiting. However, those non-recruiters collectively outperformed every other professional recruiting segment — scoring just as high or higher by every metric: email

quality, outreach technique, and word count. No matter how many recruiters you hire, there is no substitute for a heart-felt note from a future manager.

However, managers have responsibilities beyond recruiting and it's not realistic to spend eight hours a day reading resumes and penning candidate emails — professional recruiters are a necessity. However, most managers probably hope to hire a recruiter who does the job better than themselves. Of all of the emails Pete received, only 40% of the recruiter emails scored better than the average manager who actively sought out Pete London. And within this top 40%, there were proportionately more start-up recruiters than any other segment.

TIP #13: *Look for recruiters with start-up backgrounds rather than large companies.*

TIP #14: *Hire the best recruiters and treat them like gold. If a product is only as good as its team, then the product is only as good as its recruiting team.*

Summary

Of the 382 recruiters, there was only one recruiter who actually figured it out. To do so, he did one thing that no other recruiter did — picked up the phone and called someone who should have been connected to Pete to ask for an introduction. And that's where the ruse unraveled. If there were one recruiter I would have partnered with during my toughest hiring crunch ever, it would have been him.

However, that recruiter had also recruited for Meebo the prior year and he shouldn't have been poaching Pete London from our team. He apologized. In the end, the honeypot ended up identifying the one amazing recruiter I already knew about but couldn't justify working with again.

Ultimately, our recruiting challenge was solved by hiring more JavaScript managers who could help recruit too. ■

Elaine Wherry co-founded Meebo in 2005 and served as Meebo's Chief Experience Officer and Vice President of Product. Prior to Meebo, Elaine Wherry served as Manager of Usability & Design at Synaptics. At Stanford, she majored in Symbolic Systems with a concentration in Human-Computer Interaction. Her unmarketable interests include seeing the world via rented bicycles, playing the violin, and perfecting homemade ice cream recipes.

Reprinted with permission of the original author.
First appeared in *hn.my/honeypot* (ewherry.com)

My Prime Factorization Sweater

By SONDRA EKLUND

I WORE MY PRIME Factorization Sweater to KidlitCon09, and it shows up in all my pictures, so I think it's time for me to explain it.

This is the sweater that proves that I am a Certified Math Nut.

Okay, here's how it works. You have to start in the bottom left-hand corner, because the mathematician in me couldn't bear to start anywhere except where the origin would be on Cartesian coordinates. Naturally, the numbers go from left to right and from low to high.

I'll post a picture of the front of the sweater:

Okay, look at the bottom row. It looks like there is a blank space on the left. That represents 1, because 1 is the background color, because 1 is a factor of every number.

Next is a blue square, which represents 2.

Next is a red square, for 3.

Then comes 4. 4 = 2 x 2. So 4 is represented by two blue rectangles.

Then comes 5. 5 is prime, so 5 gets a new color, yellow.

Next is 6. 6 = 2 x 3. So 6 is represented by a blue rectangle and a red rectangle.

7 gets a new color, purple.

8 comes next. 8 = 2 x 2 x 2. So 8 is in a square with three blue rectangles.

Then comes 9. 9 = 3 x 3. Two red rectangles.

Last on the bottom row is 10. 10 = 2 x 5, so we have blue and yellow.The second row starts with 11, which is given the color pink.

12 has three factors, since 12 = 2 x 2 x 3, so two blues and a red.

Get the idea? This sweater presents a chart giving the color-coded prime factorization of every number from 2 to 100.

The patterns are wonderful and fascinating. You'll quickly notice that the yellows and the blues line up, because 5 and 2 are factors of 10. You also might notice that all perfect squares are symmetrical. Multiples of 11 go in a lovely pink diagonal across the sweater. There are hundreds more patterns. It would be a lovely visual aid for teaching number theory. Fun to quietly wear to Math competitions, too!

What's more, you can use this as a quick conversion table to convert to Octal (Base 8), because on the back I did the same thing with rows of 8:

The fun thing about rows of 8 is that the patterns are all different! Notice how the last column is full of blue squares because every number there is a multiple of 8 and has at least three factors of 2. And now 9 (two reds) acts like 11, going diagonally up the sweater, as does 7 (purple) in the opposite direction.

On the sleeves, I did rows of 2 and rows of 3. The rows of 3 is the only one where the blues do not line up, because 2 and 3 are relatively prime.

Isn't it just the coolest thing in the world?!!!

Okay, I warned you: this is the item that proves I am a Certified Math Nut. I can get hugely excited and animated talking about this sweater.

I have already done a library program called "Puzzles and Patterns" showing kids how they can make simple codes using the ideas from this sweater. There's definitely a children's book in there, but I haven't gotten around to writing it yet. I definitely plan to some day!

One of the cool things about this sweater is that it works in any language and on any planet!!! You see, even if an alien race had only four fingers on each hand, they could look at the back of the sweater and all their numbers would work. For that matter, a number system with a base of 7 or some other strange base would still work, even though it might not be in neat rows for that base. The chart is entirely independent of the symbols used to represent a number, and based only on color.

So we had a family joke that if an alien ever came to our door, we'd run and get the sweater to prove that we are intelligent life.

I only hope the aliens are not color blind!

Of course I also like to tell the story that when I was knitting this sweater, I brought it along to visit my family and friends one Christmas. Most of my family are Math Geeks, too, so they were impressed. But one friend had a young son who listened to my explanation and responded, "That's just weird!"

What can I say? He does have a point. Call me weird, but I still think it's one of the coolest things in the world! ■

Sondra taught college math for 10 years, then switched careers and am currently a children's librarian who loves getting kids excited about reading — and math. She knits mathematical objects for fun.

Reprinted with permission of the original author. First appeared in *hn.my/prime* (sonderbooks.com)

Check out Sondra's CafePress Store [cafepress.com/sonderbooks], where you can order t-shirts using this idea.

Hacking the iPod

How I Earned $65K in High School

By TEDDY WORCESTER

After a long day at school, the house phone rang and my mother answered. "It's Apple and they want to have a word with you," she said. At the time, I was 16 and I had been hustling iPod parts to all parts of the world.

> *"I'm not telling you this as an authority but as say, an uncle figure: you need to stop what you're doing." – An undisclosed Apple attorney*

When I was 15, my third-generation iPod had broken. This was a tragedy as music has been a huge part of my life for as long as I can remember. With no funds to purchase a new iPod, I was determined to fix it. After scouring eBay, I purchased a logic board and read countless tutorials on how to crack open my iPod and surgically replace the logic board. The operation was successful and I felt triumphant — functioning iPod, new awesome skill as iPod-surgeon, and none of my music was `lost. I grew very curious as to how frequently out-of-warranty iPods malfunctioned and simply required a new part or two. Everyone knows that out-of-warranty Apple repairs are absurdly expensive, often costing as much as a new iPod. I found that second- and third-generation iPods broke a lot and people rarely bothered fixing them. Word spread amongst my friends that I could fix broken iPods and soon after, people flocked to me to fix their iPods. The supply for parts was scant and as a result, prices were very high. I started buying broken iPods by the bulk, salvaging the functioning parts and accumulating a surplus of parts to fix friends' iPods.

A hobby and good deed turned into an obsession and I started buying huge bulk orders of broken iPods and selling the parts on eBay. By the time I was

17, I had purchased over a hundred iPods, turning a spare room in my house into an iPod graveyard. From 2005 to 2008 (15-18), I had taken in more than $65,000 in revenue from my iPod and eBay ventures before I could even legally hold a Paypal account.

I saw the third-generation iPod evolve into the fourth-generation and then the fourth-generation color, and eventually the beautiful fifth- generation, arguably the biggest leap in technology of any of the iPod generations. Generation after generation, as the components shrank, repairs became harder and harder. I hated working on iPod Minis. Nanos? Forget about it. The parts became so integrated and hard to replace that the market for parts deteriorated. I had a good two-year run, but I wasn't making much money off of parts anymore. Instead of buying and selling parts, I started to buy broken iPods that were still under warranty, mailing them back to Apple and receiving brand new refurbished iPods for the cost of shipping. This was the most lucrative venture of all, but it was the primary reason why an Apple lawyer had called me that day. Understandably so, they did not like me taking advantage of their transferrable warranties. They knew that I was a kid and let me off the hook, but it hurt to have Apple crush your income stream, the income that had allowed me to avoid a high school job while my cohort was slaving away at part-time jobs.

I learned so much peddling iPod parts. From customer service, to accounting, to shipping logistics; it was my foray into how a business functioned. I made a lot of silly mistakes, but they were all part of the learning experience. My profit margin was not monstrous, but the hard work and the lessons learned were invaluable. I differentiated myself by offering international shipping, a service that few sellers bothered with at the time. Receiving orders from China, Eastern Europe, Australia, and numerous far flung regions was incredibly exciting and eye-opening. The power of e-commerce allowed a high school student to offer an affordable way for someone across the world to repair their iPod. It fascinated the hell out of me. Once you experience this power first-hand, it becomes addicting. The internet had won me over one iPod at a time. ■

Teddy Worcester is a 22-year old product manager living and working remotely in San Francisco. In his spare time, he rides bicycles and writes about travel. You can follow him on Twitter at @teddy

Reprinted with permission of the original author.
First appeared in *teddy.is/ipod*

The Rules of Story Telling

By EMMA COATS

01 You admire a character for trying more than for their successes.

02 You've got to keep in mind what's interesting to an audience, not what's fun to do as a writer. They can be very different.

03 Trying for theme is important, but you won't see what the story is actually about till you're at the end of it. Now rewrite.

04 Once upon a time there was ___. Every day, ___. One day ___. Because of that, ___. Because of that, ___. Until finally ___.

05 Simplify. Focus. Combine characters. Hop over detours. You'll feel like you're losing valuable stuff but it sets you free.

06 What is your character good at, comfortable with? Throw the polar opposite at them. Challenge them. How do they deal?

07 Come up with your ending before you figure out your middle. Seriously. Endings are hard, get yours working up front.

08 Finish your story, let go even if it's not perfect. In an ideal world you have both, but move on. Do better next time.

09 When you're stuck, make a list of what WOULDN'T happen next. Lots of times the material to get you unstuck will show up.

10 Pull apart the stories you like. What you like in them is a part of you; you've got to recognize it before you can use it.

11 Putting it on paper lets you start fixing it. If it stays in your head, a perfect idea, you'll never share it with anyone.

12 Discount the 1st thing that comes to mind. And the 2nd, 3rd, 4th, 5th — get the obvious out of the way. Surprise yourself.

13 Give your characters opinions. Passive/malleable might seem likable to you as you write, but it's poison to the audience.

14 Why must you tell THIS story? What's the belief burning within you that your story feeds off of? That's the heart of it.

15 If you were your character, in this situation, how would you feel? Honesty lends credibility to unbelievable situations.

16 What are the stakes? Give us reason to root for the character. What happens if they don't succeed? Stack the odds against.

17 No work is ever wasted. If it's not working, let go and move on. It'll be useful later.

18 You have to know yourself: the difference between doing your best and fussing. Story is testing, not refining.

19 Coincidences to get characters into trouble are great; coincidences to get them out of it are cheating.

20 Exercise: take the building blocks of a movie you dislike. How do you rearrange them into what you DO like?

21 You've got to identify with your situation/characters; you can't just write "cool." What would make YOU act that way?

22 What's the essence of your story? What's the most economical telling of it? If you know that, you can build out from there. ■

Emma Coats worked as a storyboard artist at Pixar for over five years, and has been writing & directing live-action short films almost as long. She recently left Pixar to pursue a career in the liveaction film industry. You can follow her on Twitter: @lawnrocket

20 Lines of Code That Will Beat A/B Testing Every Time

By STEVE HANOV

A/B testing is used far too often, for something that performs so badly. It is defective by design: segment users into two groups. Show the A group the old tried-and-true stuff. Show the B group the new whiz-bang design with the bigger buttons and slightly different copy. After a while, take a look at the stats and figure out which group presses the button more often. Sounds good, right? The problem is staring you in the face. It is the same dilemma faced by researchers administering drug studies. During drug trials, you can only give half the patients the life saving treatment. The others get sugar water. If the treatment works, group B lost out. This sacrifice is made to get good data. But it doesn't have to be this way.

In recent years, hundreds of the brightest minds of modern civilization have been hard at work not curing cancer. Instead, they have been refining techniques for getting you and me to click on banner ads. It has been working. Both Google and Microsoft are focusing on using more information about visitors to predict what to show them. Strangely, anything better than A/B testing is absent from mainstream tools, including Google Analytics and Google Website optimizer. I hope to change that by raising awareness about better techniques.

With a simple twenty-line change to how A/B testing works, that you can implement today, you can always do better than A/B testing— sometimes, two or three times better. This method has several good points:

- It can reasonably handle more than two options at once, e.g., A, B, C, D, E, F, G….
- New options can be added or removed at any time.

But the most enticing part is that you can set it and forget it. If your time is really worth $1000/hour, you really don't have time to go back and check how every change you made is doing and pick options. You don't have time to write rambling blog entries about how you got your site redesigned and changed this and that and it worked or didn't work. Let the algorithm do its job. These twenty lines of code automatically find the best choice quickly, and then uses it until it stops being the best choice.

The Multi-armed Bandit Problem

The multi-armed bandit problem takes its terminology from a casino. You are faced with a wall of slot machines, each with its own lever. You suspect that some slot machines pay out more frequently than others. How can you learn which machine is the best, and get the most coins in the fewest trials?

Like many techniques in machine learning, the simplest strategy is hard to beat. More complicated techniques are worth considering, but they may eke out only a few hundredths of a percentage point of performance. One strategy that has been shown to perform well time after time in practical problems is the epsilon-greedy method. We always keep track of the number of pulls of the lever and the amount of rewards we have received from that lever. We choose a lever at random10% of the time. The other 90% of the time, we choose the lever that has the highest expectation of rewards.

```
def choose():
    if math.random() < 0.1:
        # exploration!
        # choose a random lever 10% of the time.
    else:
        # exploitation!
        # for each lever,
            # calculate the expectation of
            # reward. This is the number of
            # trials of the lever divided by the
            # total reward given by that lever.
        # choose the lever with the greatest
        # expectation of reward.
    # increment the number of times the chosen
    # lever has been played.
    # store test data in redis, choice in
    # session key, etc..

def reward(choice, amount):
    # add the reward to the total for the given
    # lever.
```

Why Does This Work?

Let's say we are choosing a color for the "Buy now!" button. The choices are orange, green, or white. We initialize all three choices to one win out of one try. It doesn't really matter what we initialize them to, because the algorithm will adapt. So when we start out, the internal test data looks like this.

Orange	Green	White
1/1 = 100%	1/1 = 100%	1/1 = 100%

Then a website visitor comes along and we have to show them a button. We choose the first one with the highest expectation of winning. The algorithm thinks they all work 100% of the time, so it chooses the first one: orange. But, alas, the visitor doesn't click on the button.

Orange	Green	White
1/2 = 50%	1/1 = 100%	1/1 = 100%

Another visitor comes along. We definitely won't show them orange, since we think it only has a 50% chance of working. So we choose Green. They don't click. The same thing happens for several more visitors, and we end up cycling through the choices. In the process, we refine our estimate of the click through rate for each option downwards.

Orange	Green	White
1/4 = 25%	1/4 = 25%	1/4 = 25%

But suddenly, someone clicks on the orange button! Quickly, the browser makes an Ajax call to our reward `function $.ajax(url:"/reward?testname=buy-button");` and our code updates the results:

Orange	Green	White
2/5 = 40%	1/4 = 25%	1/4 = 25%

When our intrepid web developer sees this, he scratches his head. What the F*? The orange button is the worst choice. Its font is tiny! The green button is obviously the better one. All is lost! The greedy algorithm will always choose it forever now!

But wait, let's see what happens if Orange is really the suboptimal choice. Since the algorithm now believes it is the best, it will always be shown. That is, until it stops working well. Then the other choices start to look better.

Orange	Green	White
2/9 = 22%	1/4 = 25%	1/4 = 25%

After many more visits, the best choice, if there is one, will have been found, and will be shown 90% of the time. Here are some results based on an actual web site that I have been working on. We also have an estimate of the click through rate for each choice. ■

Orange	Green	White
114/4071 = 2.8%	205/6385 = 3.2%	59/2264 = 2.6%

Steve Hanov has worked on everything from embedded wireless protocol stacks to natural language processing and web apps. In his spare time, Steve works on *websequencediagrams.com* and *rhymebrain.com* and blogs about computer science topics from his home base in Waterloo, Ontario.

Reprinted with permission of the original author.
First appeared in *hn.my/bandits* (stevehanov.ca)

```
#Newbie programmer
def factorial(x):
    if x == 0:
      return 1
    else:
      return x * factorial(x - 1)
print factorial(6)
```

```
#First year programmer, studied
Pascal
def factorial(x):
    result = 1
    i = 2
    while i <= x:
        result = result * i
        i = i + 1
    return result
print factorial(6)
```

```
#First year programmer, studied C
def fact(x): #{
    result = i = 1;
    while (i <= x): #{
        result *= i;
        i += 1;
    #}
    return result;
#}
print(fact(6))
```

Evolution of a Python Programmer.py

By FERNANDO MEYER

```
#First year programmer, SICP
@tailcall
def fact(x, acc=1):
  if (x > 1): return (fact((x - 1), (acc * x)))
  else:       return acc
print(fact(6))


#First year programmer, Python
def Factorial(x):
    res = 1
    for i in xrange(2, x + 1):
        res *= i
    return res
print Factorial(6)


#Lazy Python programmer
def fact(x):
    return x > 1 and x * fact(x - 1) or 1
print fact(6)

#Lazier Python programmer
f = lambda x: x and x * f(x - 1) or 1
print f(6)


#Python expert programmer
import operator as op
import functional as f
fact = lambda x: f.foldl(op.mul, 1, xrange(2, x + 1))
print fact(6)
```

```
#Python hacker
import sys
@tailcall
def fact(x, acc=1):
  if x: return fact(x.__sub__(1), acc.__mul__(x))
  return acc
sys.stdout.write(str(fact(6)) + '\n')


#EXPERT PROGRAMMER
import c_math
fact = c_math.fact
print fact(6)


#ENGLISH EXPERT PROGRAMMER
import c_maths
fact = c_maths.fact
print fact(6)


#Web designer
def factorial(x):
  #-----------------------------------------------
  #-- Code snippet from The Math Vault          --
  #-- Calculate factorial (C) Arthur Smith 1999 --
  #-----------------------------------------------
    result = str(1)
    i = 1 #Thanks Adam
    while i <= x:
        #result = result * i  #It's faster
        #result = str(result * result + i)
        result str(int(result) * i)
        #result = int(str(result) * i)
        i = i + 1
    return result
print factorial(6)
```

```
#Unix programmer
import os
def fact(x):
    os.system('factorial ' + str(x))
fact(6)


#Windows programmer
NULL = None
def CalculateAndPrintFactorialEx(dwNumber,
                                 hOutputDevice,
                                 lpLparam,
                                 lpWparam,
                                 lpsscSecurity,
                                 *dwReserved):
    if lpsscSecurity != NULL:
        return NULL #Not implemented
    dwResult = dwCounter = 1
    while dwCounter <= dwNumber:
        dwResult *= dwCounter
        dwCounter += 1
    hOutputDevice.write(str(dwResult))
    hOutputDevice.write('\n')
    return 1
import sys
CalculateAndPrintFactorialEx(6, sys.stdout, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL)


#Enterprise programmer
def new(cls, *args, **kwargs):
    return cls(*args, **kwargs)

class Number(object):
    pass

class IntegralNumber(int, Number):
```

```
    def toInt(self):
        return new (int, self)

class InternalBase(object):
    def __init__(self, base):
        self.base = base.toInt()

    def getBase(self):
        return new (IntegralNumber, self.base)

class MathematicsSystem(object):
    def __init__(self, ibase):
        Abstract

    @classmethod
    def getInstance(cls, ibase):
        try:
            cls.__instance
        except AttributeError:
            cls.__instance = new (cls, ibase)
        return cls.__instance

class StandardMathematicsSystem(MathematicsSystem):
    def __init__(self, ibase):
      if ibase.getBase() != new (IntegralNumber, 2):
          raise NotImplementedError
      self.base = ibase.getBase()

    def calculateFactorial(self, target):
        result = new (IntegralNumber, 1)
        i = new (IntegralNumber, 2)
        while i <= target:
            result = result * i
            i = i + new (IntegralNumber, 1)
        return result

print StandardMathematicsSystem.getInstance(new (InternalBase, new
(IntegralNumber, 2))).calculateFactorial(new (IntegralNumber, 6))
```

Fernando is a brazilian software engineer who nowadays lives in Sao Paulo and enjoys his wife, books, gadgets and life. He blogs at *fmeyer.org*

Reprinted with permission of the original author. First appeared in *hn.my/evo* (github.com)

Complication is What Happens When You Try to Solve a Problem You Don't Understand

By ANDY BOOTHE

Code should be simple. Code should be butt simple. Code should be so simple that there's no way it can be misunderstood. Good code has no nooks. Good code has no crannies. Good code is a round room with no corners for bugs to hide in.

We all know this. So why does most code suck?

Because it's written by people who don't understand the problem they're trying to solve.

What is a program?

To make a gross oversimplification, a program is nothing but a model of things (for the sake of discussion, call them "objects") and rules for how those objects interact with each other.

A factorial program is nothing but a group of objects (those "integer" things), and a rule that turns one integer into another (the factorial function). A word processor is nothing but a group of objects (the "alphabet"), and a bunch of rules that describe how those letters can be combined and displayed on a page. And a social network is nothing but a group of objects ("people," usually "idiots") and a bunch of rules about how those people can do stuff to annoy you.

I may be an introvert.

Anyway, these objects and their associated rules should be very simple. In fact, as Einstein pointed out, these objects and their associated rules should be made as simple as possible, but no simpler. As the model needs to be able to do more and harder things, the objects and the rules will start to:

- Increase in number
- Remember more data
- Gain more and more corner cases

This gradual accretion of nuance and behavior is called "complexity."

Complexity? But this is about complication, you moron.

Oh. Right.

So, remember a minute ago when I said "This gradual accretion of nuance and behavior is called 'complexity'"? Well… I lied.

But just a little.

In reality, this gradual accretion of nuance and behavior from none at all up to and including the minimum possible simplicity is called "complexity." Any incremental nuance and behavior above and beyond that minimum is "complication."

Complexity is a necessary evil when building systems that do anything useful. If you're doing anything more complex than putting `Hello, world!` on the screen, you're going to need some complexity. Complication, on the other hand, is the bane of programmers' existence.

When you pick up a new code base and it's a Gordian mess of 1,000-line functions, 10-deep if/else ladders, and — shudder — gotos, you're bearing horrified witness to a monument of complication. And when you start adding to your own code things like haphazard conditions, or duplicated, slightly different exceptional cases in 6 different layers of your model, or generally making any change to your program just hoping that it will work this time for the love of God without understanding the changes you're making, you're worshipping at complication's altar.

So what's a dev to do?

Ultimately, a programmer's job is less to actually write code, and more to manage complexity. Obviously you need to build features and meet deadlines, but the code itself is incidental. Hypothetically, if you could build features without writing code — such as by making a configuration change — then you should. When you do have to write code, though, it's your job to write the simplest possible code as much as it is to build the feature at hand.

So, since a programmer's real job to manage complexity, there's only one thing a developer can do in the face of complication — simplify, simplify, simplify.

A good developer has a natural, almost visceral aversion to complexity. A good developer smells complexity a mile away, and constantly shifts the code to keep his eyes to the front and his back upwind just so complexity can't sneak up on him. It's only by diligently trying to avoid all complexity that one can in fact avoid unnecessary complexity.

The best way to manage complication is to avoid creating it in the first place. If you find yourself in a `mindless change → pray → run` loop, you don't understand your code well enough to be editing it. Stop what you're doing, actually get up and walk away from the keyboard, think about what you're trying to do, and don't come back to the keyboard until you understand exactly what you're doing and how to do it. Obviously there's some slack here for debugging, but it's not controversial to say that you shouldn't change code you don't understand, even (especially?) when it's your own.

Unfortunately, despite our best efforts, complication always finds its way in. The best way to deal with complication that has already found its way into your codebase is to attack it whenever you find it. As you're sitting down for a coding session and reading your code to get it back into your head, if it takes you longer than about 10 minutes to really get going, your code's too complicated. Take the opportunity to make it simpler. (If you're unfamiliar with refactoring, Martin Fowler's Refactoring: Improving the Design of Existing Code is the bible. Read it, live it, love it, thank me later.) Do that every time you sit down, and before too long your code will be less complicated, and you'll hate yourself just a little less. ■

Andy Boothe has been a developer for more than 10 years, during which time he's written code for everything from calculators to enterprise application servers. He spends his time now as an analyst and data scientist specializing in social media analysis for the Fortune 500. You can find Andy on his website, *sigpwned.com*, or on Twitter as @sigpwned

Reprinted with permission of the original author. First appeared in *hn.my/complicate* (sigpwned.com)

The Lazy Man's URL Parsing in JavaScript

By JOE ZIM

Have you ever needed to parse a URL using regular expressions? It's not easy to write regular expressions (for a lot of people, including myself), and it's even tougher to test if that regular expression is reliable across every situation. You could, of course, just copy and paste a regular expression (or function or library) that someone else developed and use that, but I propose that there is a simpler and more concise way of parsing URLs that doesn't require any regular expressions.

This method — originally posted on Github by John Long [gist.github.com/2428561], though probably not originally discovered by him — uses native parsing abilities built into the DOM to give you simple access to the parts of a URL simply by querying properties of an anchor element. Check it out:

```
var parser = document.
createElement('a');
parser.href = "http://
example.com:3000/
pathname/?search=test#hash";

parser.protocol; // => "http:"
parser.hostname; // => "example.com"
parser.port;     // => "3000"
parser.pathname; // => "/pathname/"
parser.search;   // =>
"?search=test"
parser.hash;     // => "#hash"
parser.host;     // => "example.
com:3000"
```

This code is pulled directly from the Gist that John Long posted at the above link. I haven't seen any statements about which browsers this works with, but I assume that, at a minimum, it works with all modern browsers. If you don't trust it,

you can either test it yourself, or use a library such as URI.js [hn.my/URI.js].

One of the coolest things about this method is that you can enter a partial/relative URL into the href property and the browser will make it a full URL, just like it translates partial URLs on real HTML links into full URLs. For example, try this using your browser's console on this page:

```
var parser = document.
createElement('a');
parser.href = "/";

parser.href; // =>
"http://www.joezimjs.com/"
```

You could also just use an empty string for the href and it would give you your current URL (not including the hash, though), but this is a waste because window.location has the exact same properties, so you don't even need to create an anchor element for that.

In all of these examples, you still need to parse the query string, but at least you've got it pulled out of the URL.

This does not work in IE6 because the href property isn't parsed into a full URL unless it is parsed by the HTML parser. There is a simple workaround that forces the HTML parser to go over it though:

```
function canonicalize(url) {
  var div = document.createElement('div');
  div.innerHTML = "<a></a>";
  div.firstChild.href = url;
  // Ensures that the href is properly
  // escaped
  div.innerHTML = div.innerHTML;
  // Run the current innerHTML back
  // through the parser
  return div.firstChild.href;
} ■
```

Joe Zim has been doing web development for 12 years, which may make him sound old, but since he started in middle school, he's still pretty young. HTML and CSS were the coolest inventions ever. In college, Joe was introduced to real JavaScript, starting his full addiction. Now his addiction pushes him to continuously learn more and spread the knowledge to the internet.

Reprinted with permission of the original author.
First appeared in *hn.my/lazy* (joezimjs.com)

Accept payments online.

MEMSET® HOSTING

Rent your IT infrastructure from Memset and discover the incredible benefits of cloud computing.

MINISERVER™ CLOUD COMPUTE

From £0.015p/hour
to 4 x 2.9 GHz Xeon cores
31 GBytes RAM
2.5TB RAID(1) disk

MEMSTORE™ CLOUD STORAGE

£0.07p/GByte/month or less
99.999999% object durability
99.995% availability guarantee
RESTful API, FTP/SFTP and CDN Service

SCAN THE CODE FOR MORE INFORMATION

MEMSET® HOSTING

CarbonNeutral® hosting

ISO 9001: Quality
ISO 14001: Environmental
ISO 27001: Security

Find out more about us at www.memset.com

or chat to our sales team on 0800 634 9270.