

A vertical rocket launch at night. A bright green plume of fire and smoke extends from the ground to the top of the frame. The background is a dark sky with some clouds. The word "Launched." is written in white text on the left side of the image.

Launched.

HACKERMONTHLY

Issue 1 June 2010

Curator

Lim Cheng Soon

Contributors

Brian Shul
Carlos Bueno
Jamie Zawinski
Eric Davis
Carter Cleveland
Bradford Cross
Hamilton Ulmer
Adam Kempa
Gary Haran
Walt Kania
Evan Miller
Tawheed Kader
Paul Graham
Jason Cohen
Steve Blank
Dave Rodenbaugh
William A. Wood

Proofreader

Ricky de Laveaga

Printer

MagCloud

Advertising

ads@hackermonthly.com

Rate Card

hackermonthly.com/ratecard

Contact

curator@hackermonthly.com

Published by

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

Curator's Note

I WOULD LIKE TO give huge thanks to the contributors, who so generously granted Hacker Monthly the permission to reprint their articles; to the advertisers, who believe in us despite it being just the first issue; to Paul Graham, who thankfully did not oppose this idea and gave me the go-ahead; and most of all, to the members of Hacker News, who provided both support and valuable feedback to materialize the idea.

Creating Hacker Monthly has been both interesting and educational. Prior to this, I did not have any experience working with magazines or print. In one month, I've learned everything I could about printing magazines and spent countless hours working my way through Adobe InDesign. I've also exchanged hundreds of emails asking for reprint permissions and looking for prospective advertisers. The only downside has been I don't have much time left to code, which I miss quite a bit.

I remember the day I was sitting at Starbucks, when I started imagining what the magazine version of Hacker News would be like. I can finally stop imagining now.

— *Lim Cheng Soon*

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com* — a social news website wildly popular among hackers 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*.

Contents

FEATURES

4 Flying the SR-71 Blackbird

By BRIAN SHUL

10 A Dismal Guide to Concurrency

By CARLOS BUENO



PROGRAMMING

14 iPhone Developer: "This is why I sell beer"

By JAMIE ZAWINSKI

16 2 Steps to Becoming a Great Developer

By ERIC DAVIS

18 Top Three Motivators for Developers

By DAVE RODENBAUGH

STARTUP

32 How I Took My Web-App to Market in 3 Days

By TAWHEED KADER

34 Organic Startup Ideas

By PAUL GRAHAM

36 Not Disruptive, and Proud of It

By JASON COHEN

38 Turning on Your Reality Distortion Field

By STEVE BLANK

CAREER

20 What Value do We Create Here?

By CARTER CLEVELAND

22 7 Tips for Successful Self-Learning

By BRAFORD CROSS and HAMILTON ULMER

SPECIAL

25 Adam?...is there a reason your laptop is in the fridge?

By ADAM KEMPA

26 The Scariest Pricing Idea Ever

By WALT KANIA

29 5 Actions that Made Me Happier

By GARY HARAN

30 How Not to Run an A/B Test

By EVAN MILLER

39 Best Writing Advice for Engineers

By WILLIAM A. WOOD

FEATURES

Flying the SR-71 Blackbird

By BRIAN SHUL

IN APRIL 1986, following an attack on American soldiers in a Berlin disco, President Reagan ordered the bombing of Muammar Qaddafi's terrorist camps in Libya. My duty was to fly over Libya and take photos recording the damage our F-111's had inflicted. Qaddafi had established a 'line of death,' a territorial marking across the Gulf of Sidra, swearing to shoot down any intruder that crossed the boundary. On the morning of April 15, I rocketed past the line at 2,125 mph.

I was piloting the SR-71 spy plane, the world's fastest jet, accompanied by Maj Walter Watson, the aircraft's reconnaissance systems officer (RSO). We had crossed into Libya and were approaching our final turn over the bleak desert landscape when Walter informed me that he was receiving missile launch signals. I quickly increased our speed, calculating the time it would take for the weapons—most likely SA-2 and SA-4 surface-to-air missiles capable of Mach 5 - to reach our altitude. I estimated that we could beat the rocket-powered missiles to the turn and stayed our course, betting our lives on the plane's performance.

After several agonizingly long seconds, we made the turn and blasted toward the Mediterranean 'You might want to pull it back,' Walter suggested. It was then that I noticed I still had the throttles full forward. The plane was flying a mile every 1.6 seconds, well above our Mach 3.2 limit. It was the fastest we would ever fly. I pulled the throttles to idle just south of Sicily, but we still overran the refueling tanker awaiting us over Gibraltar.

Scores of significant aircraft have been produced in the 100 years of flight, following the achievements of the Wright brothers, which we celebrate in December. Aircraft such as the Boeing 707, the F-86 Sabre Jet, and the P-51 Mustang are among the

important machines that have flown our skies. But the SR-71, also known as the Blackbird, stands alone as a significant contributor to Cold War victory and as the fastest plane ever—and only 93 Air Force pilots ever steered the 'sled,' as we called our aircraft.

As inconceivable as it may sound, I once discarded the plane. Literally. My first encounter with the SR-71 came when I was 10 years old in the form of molded black plastic in a Revell kit. Cementing together the long fuselage parts proved tricky, and my finished product looked less than menacing. Glue, oozing from the seams, discolored the black plastic. It seemed ungainly alongside the fighter planes in my collection, and I threw it away.

Twenty-nine years later, I stood awestruck in a Beale Air Force Base hangar, staring at the very real SR-71 before me. I had applied to fly the world's fastest jet and was receiving my first walk-around of our nation's most prestigious aircraft. In my previous 13 years as an Air Force fighter pilot, I had never seen an aircraft with such presence. At 107 feet long, it appeared big, but far from ungainly.

Ironically, the plane was dripping, much like the misshapen model had assembled in my youth. Fuel was seeping through the joints, raining down on the hangar floor. At Mach 3, the plane would expand several inches because of the severe temperature, which could heat the leading edge of the wing to 1,100 degrees. To prevent cracking, expansion joints had been built into the plane. Sealant resembling rubber glue covered the seams, but when the plane was subsonic, fuel would leak through the joints.

The SR-71 was the brainchild of Kelly Johnson, the famed Lockheed designer who created the P-38, the F-104 Starfighter, and the U-2. After the Soviets shot down Gary Powers' U-2 in 1960, Johnson began to develop

an aircraft that would fly three miles higher and five times faster than the spy plane—and still be capable of photographing your license plate. However, flying at 2,000 mph would create intense heat on the aircraft's skin. Lockheed engineers used a titanium alloy to construct more than 90 percent of the SR-71, creating special tools and manufacturing procedures to hand-build each of the 40 planes. Special heat-resistant fuel, oil, and hydraulic fluids that would function at 85,000 feet and higher also had to be developed.

In 1962, the first Blackbird successfully flew, and in 1966, the same year I graduated from high school, the Air Force began flying operational SR-71 missions. I came to the program in 1983 with a sterling record and a recommendation from my commander, completing the weeklong interview and meeting Walter, my partner for the next four years. He would ride four feet behind me, working all the cameras, radios, and electronic jamming equipment. I joked that if we were ever captured, he was the spy and I was just the driver. He told me to keep the pointy end forward.

We trained for a year, flying out of Beale AFB in California, Kadena Airbase in Okinawa, and RAF Mildenhall in England. On a typical training mission, we would take off near Sacramento, refuel over Nevada, accelerate into Montana, obtain high Mach over Colorado, turn right over New Mexico, speed across the Los Angeles Basin, run up the West Coast, turn right at Seattle, then return to Beale. Total flight time: two hours and 40 minutes.

One day, high above Arizona, we were monitoring the radio traffic of all the mortal airplanes below us. First, a Cessna pilot asked the air traffic controllers to check his ground speed. 'Ninety knots,' ATC replied. A twin Bonanza soon made the same request. »



» ‘One-twenty on the ground,’ was the reply. To our surprise, a navy F-18 came over the radio with a ground speed check. I knew exactly what he was doing. Of course, he had a ground speed indicator in his cockpit, but he wanted to let all the bug-smashers in the valley know what real speed was ‘Dusty 52, we show you at 620 on the ground,’ ATC responded. The situation was too ripe. I heard the click of Walter’s mike button in the rear seat. In his most innocent voice, Walter startled the controller by asking for a ground speed check from 81,000 feet, clearly above controlled airspace. In a cool, professional voice, the controller replied, ‘Aspen 20, I show you at 1,982 knots on the ground.’ We did not hear another transmission on that frequency all the way to the coast.

The Blackbird always showed us something new, each aircraft possessing its own unique personality. In time, we realized we were flying a national treasure. When we taxied out of our revetments for takeoff, people took notice. Traffic congregated near the airfield fences, because everyone wanted to see and hear the mighty SR-71. You could not be a part of this program and not come to love the airplane. Slowly,

she revealed her secrets to us as we earned her trust.

ONE MOONLESS NIGHT, while flying a routine training mission over the Pacific, I wondered what the sky would look like from 84,000 feet if the cockpit lighting were dark. While heading home on a straight course, I slowly turned down all of the lighting, reducing the glare and revealing the night sky. Within seconds, I turned the lights back up, fearful that the jet would know and somehow punish me. But my desire to see the sky overruled my caution, I dimmed the lighting again. To my amazement, I saw a bright light outside my window. As my eyes adjusted to the view, I realized that the brilliance was the broad expanse of the Milky Way, now a gleaming stripe across the sky. Where dark spaces in the sky had usually existed, there were now dense clusters of sparkling stars. Shooting stars flashed across the canvas every few seconds. It was like a fireworks display with no sound. I knew I had to get my eyes back on the instruments, and reluctantly I brought my attention back inside. To my surprise, with the cockpit lighting still off, I could see every gauge, lit by starlight. In the

plane’s mirrors, I could see the eerie shine of my gold spacesuit incandescently illuminated in a celestial glow. I stole one last glance out the window. Despite our speed, we seemed still before the heavens, humbled in the radiance of a much greater power. For those few moments, I felt a part of something far more significant than anything we were doing in the plane. The sharp sound of Walt’s voice on the radio brought me back to the tasks at hand as I prepared for our descent.

The SR-71 was an expensive aircraft to operate. The most significant cost was tanker support, and in 1990, confronted with budget cutbacks, the Air Force retired the SR-71. The Blackbird had outrun nearly 4,000 missiles, not once taking a scratch from enemy fire.

On her final flight, the Blackbird, destined for the Smithsonian National Air and Space Museum, sped from Los Angeles to Washington in 64 minutes, averaging 2,145 mph and setting four speed records.

The SR-71 served six presidents, protecting America for a quarter of a century. Unbeknownst to most of the country, the plane flew over North Vietnam, Red China, North Korea, the Middle East, South Africa, Cuba,

Photo credit: SR-71 Blackbird by Marcin Wichary (www.flickr.com/photos/mwichary/3422253299/), Museum of Flight by Susie Gallaway (www.flickr.com/photos/susiegallaway/3298500593/),



Nicaragua , Iran , Libya , and the Falkland Islands . On a weekly basis, the SR-71 kept watch over every Soviet nuclear submarine and mobile missile site, and all of their troop movements. It was a key factor in winning the Cold War.

I am proud to say I flew about 500 hours in this aircraft. I knew her well. She gave way to no plane, proudly dragging her sonic boom through enemy backyards with great impunity. She defeated every missile, outran every MiG, and always brought us home. In the first 100 years of manned flight, no aircraft was more remarkable.

With the Libyan coast fast approaching now, Walt asks me for the third time, if I think the jet will get to the speed and altitude we want in time. I tell him yes. I know he is concerned. He is dealing with the data; that's what engineers do, and I am glad he is. But I have my hands on the stick and throttles and can feel the heart of a thoroughbred, running now with the power and perfection she was designed to possess. I also talk to her. Like the combat veteran she is, the jet senses the target area and seems to prepare herself.

For the first time in two days, the inlet door closes flush and all vibration

is gone. We've become so used to the constant buzzing that the jet sounds quiet now in comparison. The Mach correspondingly increases slightly and the jet is flying in that confidently smooth and steady style we have so often seen at these speeds. We reach our target altitude and speed, with five miles to spare. Entering the target area, in response to the jet's newfound vitality, Walt says, 'That's amazing' and with my left hand pushing two throttles farther forward, I think to myself that there is much they don't teach in engineering school.

Out my left window, Libya looks like one huge sandbox. A featureless brown terrain stretches all the way to the horizon. There is no sign of any activity. Then Walt tells me that he is getting lots of electronic signals, and they are not the friendly kind. The jet is performing perfectly now, flying better than she has in weeks. She seems to know where she is. She likes the high Mach, as we penetrate deeper into Libyan airspace. Leaving the footprint of our sonic boom across Benghazi, I sit motionless, with stilled hands on throttles and the pitch control, my eyes glued to the gauges.

Only the Mach indicator is moving,

steadily increasing in hundredths, in a rhythmic consistency similar to the long distance runner who has caught his second wind and picked up the pace. The jet was made for this kind of performance and she wasn't about to let an errant inlet door make her miss the show. With the power of forty locomotives, we puncture the quiet African sky and continue farther south across a bleak landscape.

Walt continues to update me with numerous reactions he sees on the DEF panel. He is receiving missile-tracking signals. With each mile we traverse, every two seconds, I become more uncomfortable driving deeper into this barren and hostile land. I am glad the DEF panel is not in the front seat. It would be a big distraction now, seeing the lights flashing. In contrast, my cockpit is 'quiet' as the jet purrs and relishes her newfound strength, continuing to slowly accelerate.

The spikes are full aft now, tucked twenty-six inches deep into the nacelles. With all inlet doors tightly shut, at 3.24 Mach, the J-58s are more like ramjets now, gulping 100,000 cubic feet of air per second. We are a roaring express now, and as we roll through the enemy's backyard, I hope »

» our speed continues to defeat the missile radars below. We are approaching a turn, and this is good. It will only make it more difficult for any launched missile to solve the solution for hitting our aircraft.

I push the speed up at Walt's request. The jet does not skip a beat, nothing fluctuates, and the cameras have a rock steady platform. Walt received missile launch signals. Before he can say anything else, my left hand instinctively moves the throttles yet farther forward. My eyes are glued to temperature gauges now, as I know the jet will willingly go to speeds that can harm her. The temps are relatively cool and from all the warm temps we've encountered thus far, this surprises me but then, it really doesn't surprise me. Mach 3.31 and Walt are quiet for the moment.

I move my gloved finger across the small silver wheel on the autopilot panel, which controls the aircraft's pitch. With the deft feel known to Swiss watchmakers, surgeons, and 'dinosaurs' (old-time pilots who not only fly an airplane but 'feel it'), I rotate the pitch wheel somewhere between one-sixteenth and one-eighth inch location, a position which yields the 500-foot-per-minute climb I desire. The jet raises her nose one-sixth of a degree and knows, I'll push her higher as she goes faster. The Mach continues to rise, but during this segment of our route, I am in no mood to pull throttles back.

Walt's voice pierces the quiet of my cockpit with the news of more missile launch signals. The gravity of Walter's voice tells me that he believes the signals to be a more valid threat than the others. Within seconds he tells me to 'push it up' and I firmly press both throttles against their stops. For the next few seconds, I will let the jet go as fast as she wants. A final turn is coming up and we both know that if we can hit that turn at this speed, we most likely

will defeat any missiles. We are not there yet, though, and I'm wondering if Walt will call for a defensive turn off our course.

With no words spoken, I sense Walter is thinking in concert with me about maintaining our programmed course. To keep from worrying, I glance outside, wondering if I'll be able to visually pick up a missile aimed at us. Odd are the thoughts that wander through one's mind in times like these. I found myself recalling the words of former SR-71 pilots who were fired upon while flying missions over North Vietnam. They said the few errant missile detonations they were able to observe from the cockpit looked like implosions rather than explosions. This was due to the great speed at which the jet was hurtling away from the exploding missile.

I see nothing outside except the endless expanse of a steel blue sky and the broad patch of tan earth far below. I have only had my eyes out of the cockpit for seconds, but it seems like many minutes since I have last checked the gauges inside. Returning my attention inward, I glance first at the miles counter telling me how many more to go, until we can start our turn. Then I note the Mach, and passing beyond 3.45, I realize that Walter and I have attained new personal records. The Mach continues to increase. The ride is incredibly smooth.

There seems to be a confirmed trust now, between me and the jet; she will not hesitate to deliver whatever speed we need, and I can count on no problems with the inlets. Walt and I are ultimately depending on the jet now - more so than normal - and she seems to know it. The cooler outside temperatures have awakened the spirit born into her years ago, when men dedicated to excellence took the time and care to build her well. With spikes and doors as tight as they can get, we are racing against the time it could take a missile to reach our altitude.

It is a race this jet will not let us lose. The Mach eases to 3.5 as we crest 80,000 feet. We are a bullet now - except faster. We hit the turn, and I feel some relief as our nose swings away from a country we have seen quite enough of. Screaming past Tripoli, our phenomenal speed continues to rise, and the screaming Sled pummels the enemy one more time, laying down a parting sonic boom. In seconds, we can see nothing but the expansive blue of the Mediterranean. I realize that I still have my left hand full forward and we're continuing to rocket along in maximum afterburner.

The TDI now shows us Mach numbers, not only new to our experience but flat out scary. Walt says the DEF panel is now quiet, and I know it is time to reduce our incredible speed. I pull the throttles to the min 'burner range and the jet still doesn't want to slow down. Normally the Mach would be affected immediately, when making such a large throttle movement. But for just a few moments old 960 just sat out there at the high Mach, she seemed to love and like the proud Sled she was, only began to slow when we were well out of danger. I loved that jet. ■

Brian Shul was an Air Force fighter pilot for 20 years. Shot down in Vietnam, he spent one year in hospitals and was told he'd never fly again. He flew for another 15 years, including the world's fastest jet, the SR-71. As an avid photographer Brian accumulated the world's rarest collection of SR-71 photographs and used them to create the two most popular books ever done on that aircraft, *Sled Driver*, and *The Untouchables*. Brian today is an avid nature photographer and in high demand nationwide as a motivational speaker.

Reach the hackers and
startup founders who are
building tomorrow's web.

Advertise with Hacker Monthly

Email us at ads@hackermmonthly.com. Don't forget to ask us about our introductory advertising offer.



A Dismal Guide to Concurrency

By CARLOS BUENO

TWO PEOPLE CAN paint a house faster than one can. Honeybees work independently but pass messages to each other about conditions in the field. Many forms of concurrency⁰, so obvious and natural in the real world, are actually pretty alien to the way we write programs today. It's much easier to write a program assuming that there is one processor, one memory space, sequential execution and a God's-eye view of the internal state. Language is a tool of thought as much as a means of expression, and the mindset embedded in the languages we use can get in the way.¹

We're going through an inversion of scale in computing which is making parallelism and concurrency much

more important. Single computers are no longer fast enough to handle the amounts of data we want to process. Even within one computer the relative speeds of processors, memory, storage, and network have diverged so much that they often spend more time waiting for data than doing things with it. The processor (and by extension, any program we write) is no longer a Wizard of Oz kind of character, sole arbiter of truth, at the center of everything. It's only one of many tiny bugs crawling over mountains of data.

Many hands make light work

A few years ago Tim Bray decided to find out where things stood. He put a computer on the Internet, which contained over 200 million lines of text in

one very large file. Then he challenged programmers to write a program to do some simple things with this file, such as finding the ten most common lines, which matched certain patterns. To give you a feel for the simplicity of the task, Bray's example program employed one sequential thread of execution and had 78 lines of code, something you could hack up over lunch.

The computer was unusual for the time: it had 32 independent hardware threads, which could execute simultaneously. The twist of the WideFinder challenge was that your program had to use all of those threads at once to speed up the task, while adding as little code as possible. The purpose was to demonstrate how good or bad everyday

Photo credit: *Tight Spin* by Aaron Waagner (www.flickr.com/photos/copilot/62083698).
Licensed under Creative Commons Attribution 2.0 Generic licence (creativecommons.org/licenses/by/2.0/deed.en).

programming is at splitting large jobs into parallel tracks.

How hard could it be? I thought. Very hard, as it happened. I got up to 4 parallel processes before my program collapsed under its own weight. The crux of the problem was that the file was stored on a hard drive. If you've never peeked inside a hard drive, it's like a record player with a metal disc and a magnetic head instead of a needle. Just like a record it works best when you "play" it in sequence, and not so well if you keep moving the needle around. And of course it can only play one thing at a time. So I couldn't just split the file into 32 chunks and have each thread read a chunk simultaneously. One thread had to read from the file and then dole out parts of it to the others. It was like trying to get 31 housepainters to share the same bucket.

When I looked at other people's entries for hints I was struck by how almost all of them, good and bad, looked complicated and steampunky. Part of that was my unfamiliarity with the techniques, but another part was the lack of good support for parallelism, which forced people to roll their own abstractions. (Ask four programmers to create a new abstraction and you'll get five and a half answers.) The pithiest entry was 130 lines of OCaml, a language with good support for "parallel I/O" but which is not widely used outside of academia. Most of the others were several hundred lines long. Many people like me were not able to complete the challenge at all. If it's this difficult to parallelize a trivial string-counting program, what makes us think we're doing it right in complex ones?

Ideally, concurrency shouldn't leak into the logic of programs we're trying to write. Some really smart people would figure out the right way to do it. They would write papers with lots of equations in them and fly around to conferences for a few years until some other smart people figured out what the hell they were saying. Those people would go develop libraries in our favorite programming languages. Then we could just put `import concurrent`; at the top of our programs and

be on our way. Concurrency would be another thing we no longer worry about unless we want to, like memory management. Unfortunately there is evidence that it won't be this clean and simple.² A lot of things we take for granted may have to change.

There are at least two concurrency problems to solve: how to get many components inside one computer to cooperate without stepping all over each other, and how to get many computers to cooperate without drowning in coordination overhead. These may be special cases of a more general problem and one solution will work for all. Or perhaps we'll have one kind of programming for the large and another for the small, just as the mechanics of life are different inside and outside of the cell.

At the far end of the spectrum are large distributed databases, such as those used by search engines, online retailers, and social networks. These things are enormous networks of computers that work together to handle thousands of writes and hundreds of thousands of reads every second. More machines in the system raise the odds that one of them will fail at any moment. There is also the chance that a link between groups of machines will fail, cutting the brain in half until it is repaired. There is a tricky balance between being able to read from such a system *consistently* and *quickly* and writing to it *reliably*. The situation is summed up by the CAP Theorem, which states that large systems have three desirable but conflicting properties: Consistency, Availability, and Partition-tolerance. You can only optimize for two at the expense of the third.

A **Consistent/Available** system means that reading and writing always works the way you expect, but requires a majority or quorum of nodes to be running in order to function. Think of a parliament that must have more than half of members present in order to hold a vote. If too many can't make it, say because a flood washes out the bridge, a quorum can't be formed

and business can't proceed. But when enough members are in communication the decision-making process is fast and unambiguous.

Consistent/Partitionable means that the system can recover from failures, but requires so much extra coordination that it collapses under heavy use. Imagine having to send and receive a status report for every decision made at your company. You'll always be current, and when you come back from vacation you will never miss a thing, but making actual progress would be very slow.

Available/Partitionable means that you can always read and write values, but the values you read might be out of date.

A classic example is gossip: at any point you might not know the latest on what Judy said to Bill but eventually word gets around. When you have new gossip to share you only have to

tell one or two people and trust that in time it will reach everyone who cares. Spreading gossip among computers is a bit more reliable because they are endlessly patient and (usually) don't garble messages.⁴

After lots of groping around with billions of dollars of revenue at stake, people who build these large systems are coming to the conclusion that it's most important to always be able to write to a system quickly and read from it even in the face of temporary failures. Stale data is a consequence of looser coupling and greater autonomy needed to make that possible. It's uncomfortable to accept the idea that as the computing power of an Available/Partitionable system scales up, the fog of war descends on consistency, but in practice it's not the end of the world.

This was not a whimsical nor easy choice. Imagine Ebenezer Scrooge is making so much money that Bob Cratchit can't keep up. Scrooge needs more than one employee to receive »



» and count it. To find out the grand total of his money at any point, he has to ask each of them for a subtotal. By the time Scrooge gets all the answers and adds them up, his employees have counted more money, and his total is already out of date. So he tells them to stop counting while he gathers subtotals. But this wastes valuable working time. And what if Scrooge adds another counting-house down the street? He'll have to pay a street boy, little Sammy Locke, to a) run to the other house and tell them to stop counting, b) gather their subtotals, c) deliver them to Scrooge, then d) run back to the other house to tell them to resume counting. What's worse, his customers can't pay *him* while this is happening. As his operation gets bigger Scrooge is faced with a growing tradeoff between stale information and halting everything

import concurrent;

Shared memory can be pushed fairly far, however. Instead of explicit locks, Clojure and many newer languages use an interesting technique called software transactional memory. STM simulates a sort of post-hoc, fine-grained, implicit locking. Under this scheme semi-independent workers, called threads, read and write to a shared memory space as though they were alone. The system keeps a log of what they have read and written. When a thread is finished the system verifies that no data it read was changed by any other. If so the changes are committed. If there is a conflict the transaction is aborted, changes are rolled back and the thread's job is retried. While threads operate on non-overlapping parts of memory, or even non-overlapping parts of the same data structures, they can do whatever they

always have the last laugh. A concurrent system is fundamentally limited by how often processes have to coordinate and the time it takes them to do so. As of this writing computer memory can be accessed in about 100 nanoseconds. Local network's latency is measured in microseconds to milliseconds. Schemes that work well at local memory speeds don't fly over a channel one thousand times slower. Throughput is a problem too: memory can have one hundred times the throughput of network, and is shared among at most a few dozen threads. A large distributed database can have tens of thousands of independent threads contending for the same bandwidth.

If we can't carry the shared-memory model outside of the computer, is there some other model we can bring inside? Are threads, ie semi-indepen-

“In essence, transactional memory allows threads to ask for forgiveness instead of permission.”

to wait on Locke. If there's anything Scrooge likes less than old numbers, it's paying people to do nothing.

Scrooge's dilemma is forced upon him by basic physics. You can't avoid it by using electrons instead of street urchins. It's impossible for an event happening in one place (eg data changing inside one computer or process) to affect any other place (eg other computers or processes) until the information has had time to travel between them. Where those delays are small relative to performance requirements, Scrooge can get away with various forms of locking and enjoy the *illusion* of a shared, consistent memory space. But as programs spread out over more and more independent workers, the complexity needed to maintain that illusion begins to overwhelm everything else.³

want without the overhead of locking. In essence, transactional memory allows threads to ask for forgiveness instead of permission.

As you might have guessed from those jolly hints about conflict and rollback, STM has its own special problems, like how to perform those commit/abort/retry cycles efficiently on thousands of threads. It's fun to imagine pathological conflict scenarios in which long chains of transactions unravel like a cheap sweater.⁵ STM is also not able to handle actions that aren't undoable. You can't retry most kinds of I/O for the same reason you can't rewind a live concert. This is handled by queueing up any non-reversible actions, performing them outside of the transaction, caching the result in a buffer, and replaying as necessary. Read that sentence again.

Undeniably awesome and clever as STM threads are, I'm not convinced that shared memory makes sense outside of the “cell membrane” of a single computer. Throughput and latency

dent workers that play inside a shared memory space, absolutely necessary? In his “standard lecture” on threads Xavier Leroy details three reasons people use them:

- Shared-memory parallelism using locks or transactions. This is explicitly disowned in both Erlang and Leroy's OCaml in favor of message-passing. His argument is that it's too complex, especially in garbage-collected languages, and doesn't scale.
- Overlapping I/O and computation, ie while thread A is waiting for data to be sent or received, threads B-Z can continue their work. Overlapping (aka non-blocking I/O) is needed to solve problems like WideFinder efficiently. This is often thwarted by low-level facilities inside the operating system that were written without regard to parallelism. Leroy thinks this should be fixed at the OS level instead of making every program solve it again and again.

- Coroutines, which allow different functions to call each other repeatedly without generating an infinitely long stack of references back to the first call. This looks suspiciously like message-passing.

Message-passing, which first appeared in Smalltalk, is the core abstraction of Joe Armstrong's programming language Erlang. Erlang programs do things that make programmers take notice, like run some of the busiest telephone switches for years without fail ⁶. It approaches concurrency with three iron rules: no shared memory even between processes on the same computer, a standard format for messages passed between processes, and a guarantee that messages are read in the order in which they were received. The first rule is meant to avoid the heartaches described above and embraces local knowledge over global state. The second and third keep programmers from endlessly reinventing schemes for passing messages between processes. Every Erlang process has sovereign control over its own memory space and can only affect others by sending well-formed messages. It's an elegant model and happens to be a cleaned-up version of the way the Internet itself is constructed. Message-passing is already one of the axioms of concurrent *distributed* computation, and may well be universal.

There are probably more axioms to discover. Languages become more powerful as abstractions are made explicit and standardized. Message-passing says nothing about optimizing for locality, ie making sure that processes talk with other processes that are located nearby instead of at random. It might be cool to have a standard way to measure the locality of a function call. Languages become even more powerful when abstractions are made first-class entities. For example, languages that can pass functions as arguments to other functions can generate new types of higher-order functions without the programmer having to code them by hand. A big part of distributed

computing is designing good protocols. I know of no language that allows *protocols* as first-class entities that can be passed around and manipulated like functions and objects are. I'm not even sure what that would look like but it might be interesting to try out.

There is a lot of sound and fury around parallelism and concurrency. I don't know what the answer will be. I personally suspect that a relaxed, shared-memory model will work well enough within the confines of one computer, in the way that Newtonian physics works well enough at certain scales. A more austere model will be needed for a small network of computers, and so on as you grow. Or perhaps there's something out there that will make all this lockwork moot. ■

Notes

0. Parallelism is the act of taking a large job, splitting it up into smaller ones, and doing them at once. People often use "parallel" and "concurrent" interchangeably, but there is a subtle difference. Concurrency is necessary for parallelism but not the other way around. If I alternate between cooking eggs and pancakes I'm doing both concurrently. If I'm cooking eggs while you are cooking pancakes, we are cooking concurrently and in parallel. Technically if I'm cooking eggs and you are mowing the lawn we are also working in parallel, but since no coordination is needed in that case there's nothing to talk about.

1. "The slovenliness of our language makes it easier for us to have foolish thoughts. The point is that the process is reversible." -- George Orwell, *Politics and the English Language*

"That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted." - George Boole, *The Laws of Thought*

2. Neither was the switch to memory management, come to think of it.

3. This is not about speed-of-light effects or anything like that. I'm only talking about reference frames in the

sense of "old news", such as when you find out your cousin had gotten married last year. Her wedding and your unawareness are both "true" relative to your reference frames until you receive news to the contrary.

4. The categories are not rigidly exclusive. The parliament problem is mitigated in real parliaments with quorum rules: say if a majority of members are in one place, or some minimum number is present in chambers, they can act as though they were the full body. The status report problem is usually handled by having hierarchies of supervisors and employees aka "reports". The gossip consistency problem can be helped by tagging data with timestamps or version numbers so you can reconcile conflicting values.

5. There is a recent paper about an interesting variation on this theme called HyTM, which appears to do a copy-on-write instead of performing writes to shared memory.

6. A lot of writeups repeat a "nine nines", ie 99.999999% reliability claim for Erlang-based Ericsson telephone switches owned by British Telecoms. This works out to 31 *milliseconds* of downtime per year, which hovers near the edge of measurability, not to say plausibility. I was present at a talk Armstrong gave in early 2010 during which he was asked about this. There was a little foot shuffling as he qualified it: it was actually 6 or so seconds of downtime in one device during a code update. Since BT had X devices over Y years, they calculated it as 31ms of average downtime per device per year. Or something like that. Either way it's an impressive feat.

Carlos Bueno is an engineer at Facebook. He writes occasionally about general programming topics, performance, security, and internationalization. His long-term project is to "save the web": to build a network of independent, redundant, Internet archives.

iPhone Developer: “T

By JAMIE ZAWINSKI

DALI CLOCK 2.31 is out now, I finally got the iPhone/iPad port working.

It was ridiculously difficult, because I refused to fork the MacOS X code base: the desktop and the phone are both *supposedly* within spitting distance of being the same operating system, so it should be a small matter of `ifdefs` to have the same app compile as a desktop application and an iPhone application, right?

Oh ho ho ho.

I think it's safe to say that MacOS is more source-code-compatible with NextStep than the iPhone is with MacOS. It's full of all kinds of idiocy like this -- Here's how it goes on the desktop:

```
NSColor fg = [NSColor colorWithCalibratedHue:h saturation:s
brightness:v alpha:a];
[fg getRed:&r green:&g blue:&b alpha:&a];
[fg getHue:&h saturation:&s brightness:&v alpha:&a];
```

But on the phone:

```
UIColor fg = [UIColor colorWithHue:h saturation:s brightness:v alpha:a];
const CGFloat *rgba = CGColorGetComponents ([fg CGColor]);
// Oh, you wanted to get HSV? Sorry, write your own.
```

It's just full of nonsense like that. Do you think someone looked at the old code and said, “You know what, to make this code be efficient enough to run on the iPhone, we're going to have to rename all the classes, and also make sure that the new classes have an *arbitrarily different API* and use *arbitrarily different arguments* in their methods that do exactly the same thing that the old library did! It's the only way to make this platform succeed.”

No, they got some intern who was completely unfamiliar with the old library to just write a new one from scratch without looking at what already existed.

It's 2010, and we're still innovating on how you pass

color components around. Seriously?

You can work around some of this nonsense with `#defines`, but the APIs are randomly disjoint in a bunch of ways too, so that trick only goes so far. If you have a program that manipulates colors a lot, you can imagine the world of `#ifdeffy` hurt you are in.

Preferences are the usual flying circus as well. I finally almost understood bindings, and had a vague notion of when you should use `NSUserDefaultsController` versus `NSUserDefaults`, and now guess what the iPhone doesn't have? Bindings. Or `NSUserDefaultsController`. But it does have `NSUserDefaults`. I can't explain.

Also!

I basically gave up on trying to have any kind of compatible version of either Cocoa or Quartz imaging that worked on both platforms at the same time — my intermediate attempts were a loony maze of `#ifdefs` due to arbitrary API wankery like the above, scathing examples of which I have mercifully forgotten — so finally I said “Fuck it, the iPhone runs OpenGL, right? I'll just rewrite the display layer



"This is why I sell beer"

in GL and throw away all this bullshit Quartz code." (Let's keep in mind here the insanely complicated thing I'm doing in this program: I have a *bitmap*. I want to put it on the screen, fast, using *two whole colors*. And the colors change some times. This should be fucking trivial, right? Oh, ho ho ho.)

So I rewrote it in OpenGL, just dumping my bitmap into a luminance texture, and this is where some of you are laughing at me already, because I didn't know that the iPhone actually runs OpenGL! Which has, of course, even less to do with OpenGL than iPhones have to do with Macs.

I expected the usual crazy ifdef-dance around creating the OpenGL context and requesting color buffers and whatnot, since OpenGL never specified any of that crap in a cross-platform way to begin with, but what I didn't expect — and I'm still kind of slack-jawed at this — is that OpenGL removed `glBegin()` and `glVertex()`.

No, really, it really did.

That's like, the defining characteristic of OpenGL. So OpenGL is just a slight variant of OpenGL, in the way that

unicycle is a slight variant of a city bus. If you can handle one, the other should be pretty much the same, right?

Again, what the hell — I can almost understand wanting to get rid of display lists for efficiency reasons in an embedded API (I don't like it, because my screen savers tend to use display lists a lot, but I can sort-of understand it), but given that you could *totally implement* `glBegin()` and `glVertex()` *in terms of* `glDrawArrays()` why the hell did they take them out! Gaah!

Anyway, where was I?

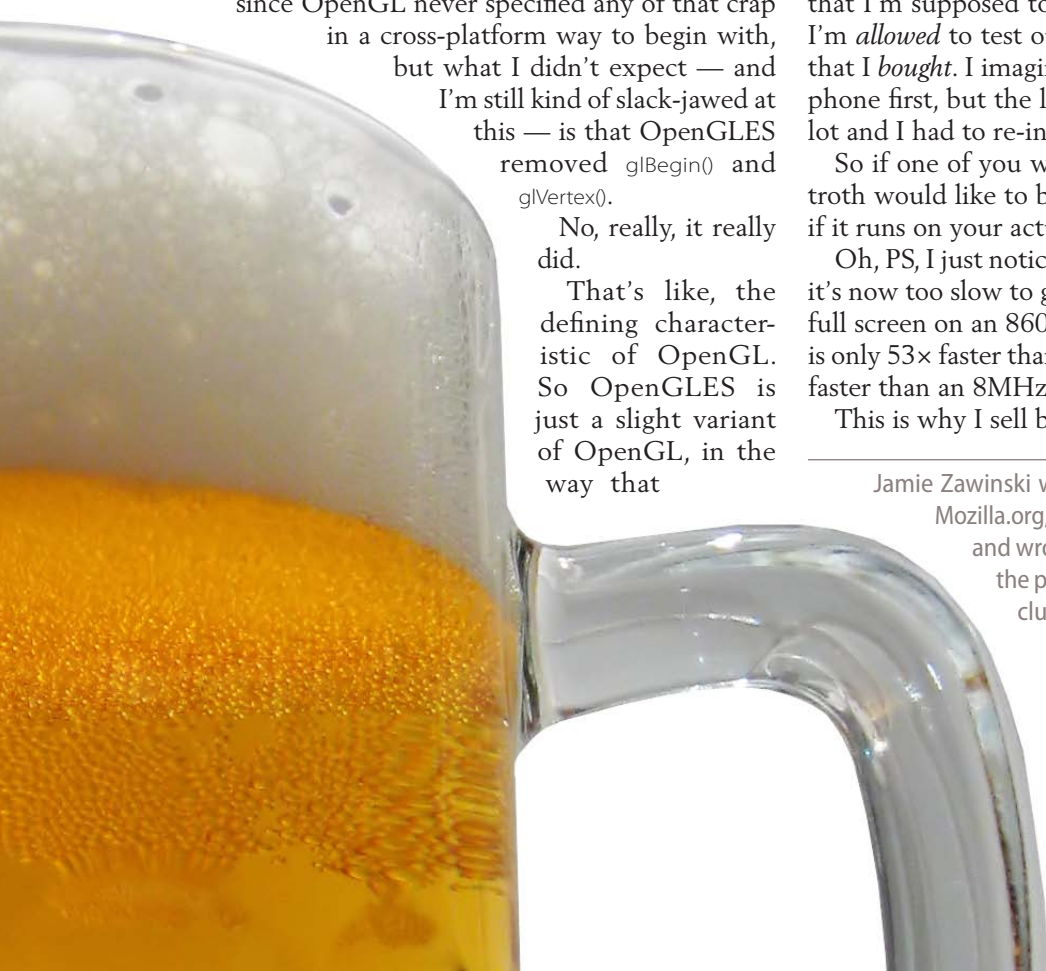
Oh, yeah. So Dali Clock works on the iPhone and iPad now, I think. I can't actually run it on my phone, because I haven't gotten over my righteous indignation at the idea that I'm supposed to tithe \$100 to Captain Steve before I'm *allowed* to test out the program I *wrote* on the phone that I *bought*. I imagine I could manage it if I jailbroke my phone first, but the last time I did that it destabilized it a lot and I had to re-install.

So if one of you who has supplicated at the App Store troth would like to build it from source and let me know if it runs on your actual device, that'd be cool.

Oh, PS, I just noticed that since I rewrote it in OpenGL, it's now too slow to get a decent frame rate when running full screen on an 860MHz PPC G4. I mean, that machine is only 53× faster than a 16MHz Palm Pilot, and only 107× faster than an 8MHz Mac128k.

This is why I sell beer. ■

Jamie Zawinski was one of the founders of Netscape and Mozilla.org, was the primary developer of Lucid Emacs, and wrote most of your screen savers. Today he is the proprietor of DNA Lounge, an all ages dance club and live music venue in San Francisco.



2 Steps to Becoming a Great Developer

By ERIC DAVIS

I WANT TO SHARE the two steps that I'm using to walk the path to becoming a great developer. Becoming a great developer is a constant work in progress, but it's a pattern that I've seen many other great developers follow.

Step One: Write More Code

This might sound easy but trust me - it's not easy. There are an infinite number of reasons we developers don't write code:

- I don't have the time
- I don't know that code base
- I don't have the right environment setup
- I don't know what to work on
- I'm tired

They all boil down to fear. You're afraid of something. Afraid of wasting time, afraid of being embarrassed publicly, afraid of making a mistake, afraid of being afraid.

Let me share two stories with you about my fears:

I've been a contributor to Redmine for a couple of years now, but I haven't been very active in the code base. Why? Redmine is a large complex code base and I didn't know how everything worked. So I stayed in my corner and only committed minor changes. Yet I still found a way to break those sections. Self-fulfilling prophecy?

With my product, SeeProjectRun,

I have to charge users' credit cards. Taking actual money is scary. After hearing all of the horror stories about companies screwing this up, I became deathly afraid of this and put off writing any billing code. Yes, me a developer who has written four credit card interfaces for active_merchant was afraid of writing code to bill his users. WTF is going on here?

Fear is a mistress that will steal your life if you let her. So how do you get over your fear of writing more code?

Write more code

As odd as it sounds, the only way I found to get through my fear of writing code was to crank it out like it was going out of style. The easiest way to do this? Start new side projects and contribute simple patches to Open Source. Every time you write code, you will learn something about the code, your tools, or yourself. Did you really think my 57 plus daily refactoring posts were only about fixing bad code? Nope, they are my sledgehammers against coder's block.

Oh and the ending to my stories about fear:

I just spent last night rewriting a core component of Redmine and committed it to the project this morning. It if breaks, I'll fix it. If it's really crap code, I'll revert it. No one will care and no one will remember the mistake.

And for the billing code I strapped myself down and finished the credit card billing code for SeeProjectRun in

two days. Throwing two hundred test cases at it proved to me that it would work good enough to get over my fear.

Don't let fear hold you back from writing code.

Step Two: Work With Great Developers

Now that you're creating code, you need to work with great developers so you can see how to they write great code. Just take:

- 1 passionate developer (you)
- 1 great developer (them)
- a dash of code

Mix well daily and after a short rise in the oven, you'll have two great developers. Feel free to add a few nuts (other great developers) and bake again.

You don't need to search for the greatest developers of all time, you just need developers smarter and further along in their skills than yourself. This can be easy if you work in a company that has hired great developers. But what do you do if your company doesn't hire any great developers or you are a solo freelancer like me?

Start reading great developers' code

I'm making it a habit to start reading great developer's code. They put out so much code, you will find yourself reading so much of it that you start to dream about code¹.

Your online portfolio.



Getting Started

Now here's the call to action, because you will never become a great developer without taking action.

1 Write At Least One Line Of Code In A New Code Base Every Day For A Week. Switch Code Bases After Each Week.

This can be a new feature, a bugfix, a refactoring, or just monkeying around with an idea. It doesn't matter, the act of thinking through the code and writing is what you are after. Don't know one a good code base to start on? Do a refactoring on Redmine and tell me about it in the comments below.

2 Find A Way To Learn From A Great Developer Every Week.

If you are working with a great developer:

- do an informal code review their last commit
- ask to pair program with them, or
- buy them lunch and ask them about their favorite hack

If you are working solo:

- download some popular projects and read through a single class every week
- get some API documentation that shows the method's source code inline and read the source each time you look up a method, or
- find a mentor and work with them on some real code

So whatever you do, take action today. Unless you're afraid of becoming a great developer...But there is plenty of room at the top. ■

Notes

1. Notice that the smart developers are always producing new code.... they are following step #1.

Eric Davis runs Little Stream Software, where he builds custom software for businesses using Redmine.

Top Three Motivators for Developers

By DAVE RODENBAUGH

SOFTWARE HAS LONG since lost its glory-days status. We're not the go-to field anymore. Geeks are no longer revered as gods amongst humanity for our ability to manipulate computers. We get crappy jobs just like everyone else.

So, what is it that still motivates you to work as a software developer?

Is it your fat salary, great perks, and end-of-year bonuses? Unless you've been working on Mars for the past two years, I think Computerworld¹ would disagree with you. We've been getting kicked in the nads just as hard as everyone else. Between budget cut-backs, layoffs and reductions in benefits or increases in hours, clearly our paychecks are not our primary source of satisfaction.

If money were our primary motive, right now we'd be seeing a mass exodus from the tech sector. So, if it's not the money, then what is it that we hang on to when we get up each day? Are we really working for those options? That salary bonus?

Turns out, we're kidding ourselves if we think that's our real motive as developers.

The assumption

People perform better when given a tangible, and even substantial, reward for completing a task. Think bonuses, stock options, and huge booze-driven parties.

The reality

In a narrow² band of actual cases, this is true. By and large, the reward-based incentive actually creates poorer performance in any group of workers for cognitive tasks, regardless of economic background or complexity of the task involved³.

I'm not making this up, nor am I just drawing on anecdotal experience. Watch this 18-minute video from TED⁴ and I'll bet you're convinced too.

Daniel Pink gave this lecture at the 2009 TED. It's mind-blowing if you're stuck in the carrot-and-stick mentality. And I'll just bet, unless you work for Google, are self-employed, or extremely worldly, *you probably are*.

I'm not saying that to be mean or controversial. I'm saying that because this mentality has pervasively spread to every business, industry and country on the planet over the past 100 years. It's not just software development, but we're hardly immune from its effect.

While we're not immune to the impact, we do have a lot going for us that gives us an advantage in stepping outside this mentality:

- Developers tend to be social odd-balls and the normal conventions seem awkward to us. Social odd-balls tend to question things. We don't like what everyone else likes because, well, we're nerds and we don't think like sales people. Or accountants. Or athletes. We're willing to try things others find weird because we're weird too.
- Because we're odd, we tend to be forward thinking and revolutionary in our approaches to workplace advancements. Think about the good aspects of the Dot Com era: pets in the workplace, recreation rooms with pool tables and ping pong, better chairs and desks for people, free lunches. Those innovations didn't come out of Pepsi, Toyota, or Price Waterhouse Coopers, they came out of tech companies. Every one.
- In doing so, our weird becomes the new normal. Witness the output of the Dot Com era: Aside from the economic meltdown, how many

“Turns out, we’re kidding ourselves if we think that [money] is our real motive as developers.”

companies now regularly practice some, if not all of those things we did back in the late 90s? (Albeit with more restraint, thankfully)

With that in mind, let’s take Daniel’s idea of the results-oriented work environment (ROWE) forward and create something new for the 21st century. It focuses on three important ideas, which developers already love and embrace: Autonomy, Mastery, Purpose.

Autonomy

What developer out there doesn’t like to be given the freedom to do their own thing, on their terms, with their preferred hours, using their tools, environment, IDE, language, operating system and favorite t-shirt? Find me a single developer anywhere that doesn’t crave this kind of freedom and I’ll pay you \$10. Seriously. Drop me a contact above. I’m good for it. Of course, you’ll search for the rest of your life and won’t be able to do it.

Mastery

Every developer on the planet wants to get better at what they do. We crave new knowledge like some people quaff coffee after a hangover. Fortunately, the side effects of getting better at development are far more benign than caffeine binging.

Purpose

Nothing is more tedious, horrific, or uninspiring to developers to work on projects that lack any real meaning in the world. Or lack any real direction. Or lack any substantial need from the company. In fact, you can probably point to the brightest points of your career all stemming from those projects that had the deepest meaning to you personally. Maybe the darkest points are those soul-sucking projects that you waded through because you were glad to have a job but desperately waited for things to improve so you could find a better job elsewhere. Preferably where soul-vacuum didn’t exist.

Google gets it: They already advocate the 20% time concept and (near-) complete workplace freedom. Atlassian gets it: They have the Fedex challenge where everyone in the company gets 24 hours to work on something they are interested in, with the caveat you have to deliver it at the end of 24 hours and you must present it to the company. Think those don’t create passion for the company? How about the Nine Things Developers Want More than Money? These points all touch on the same three basic concepts: autonomy, mastery, and purpose.

Does your company “get it”? If the answer is NO, what can you do right now to change your workplace to “get it”? And if that is too Sisyphean a task for you, how about starting your own company instead, that does “get it”?



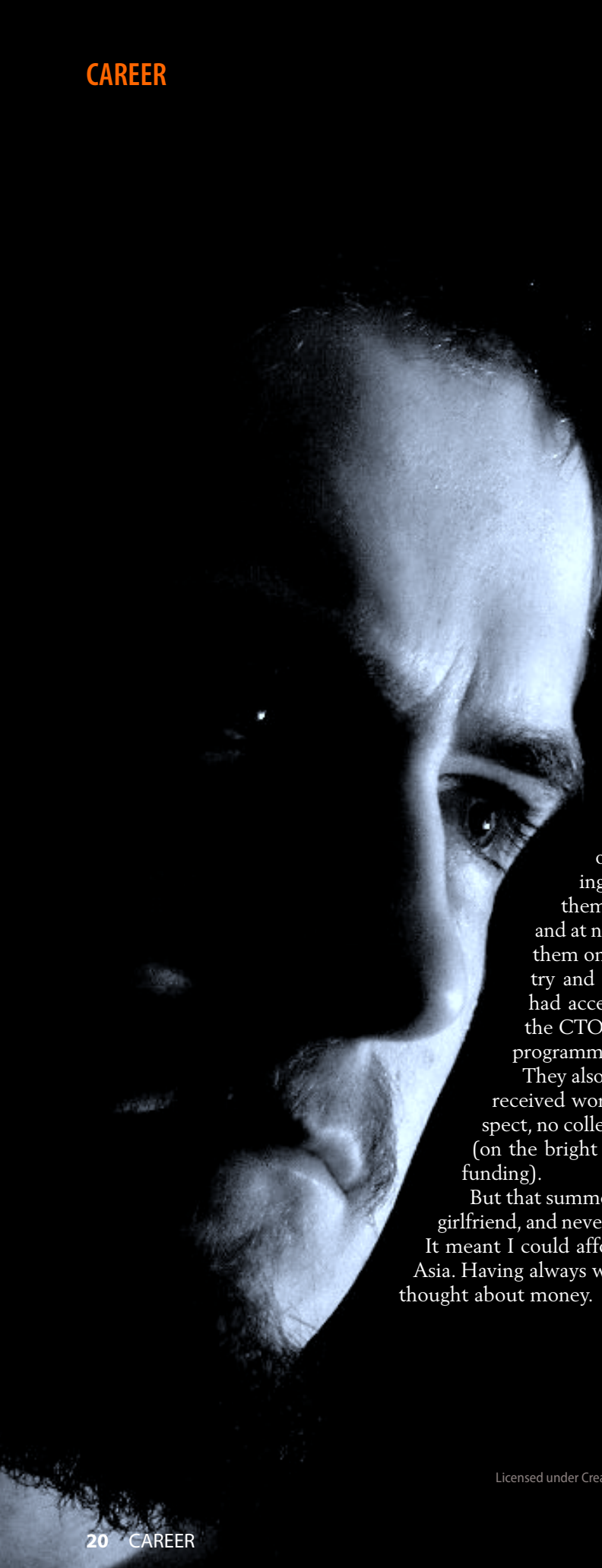
That’s my challenge for you in 2010. “Make software suck less in the 21st century”. Good luck. ■

Notes

1. http://www.computerworld.com/s/article/347538/The_Big_Squeeze
2. Anything that isn’t a cognitive task, simple or complex, according to the research I quote below.
3. Sorry, outsourcers...dangling the reward under your workers noses doesn’t help even when your home country is considerably poorer on average than Western economies. Yet another surprising finding of their research.
4. <http://www.youtube.com/watch?v=rrkrvAUbU9Y>

Dave Rodenbaugh is an independent software contractor with nearly 2 decades of enterprise project experience in a variety of companies and industries. Although he loves Java, he sometimes drinks a good black tea when the mood strikes. He’s still waiting for his first business trip to the Caribbean.

What Value



ONE SUMMER I thought I had the ultimate dream job. During the day I created software that accessed some of the world's largest financial databases and provided traders with real-time data and analysis for trade ideas. At night I worked with the CTO on a side project that analyzed huge amounts of transaction data to identify arbitrage opportunities. We figured that if we could start finding enough of these opportunities, we could present them as trade ideas to the bosses. So we wrote scripts, and at night, after everyone else left the office, we installed them on their computers and ran the scripts in parallel to try and crunch through the massive amount of data we had access to. This was fun. Really fun. And even better, the CTO was an awesome guy who taught me a lot about programming.

They also paid well. Really well. Even more than my friends received working 100 hour weeks at I-Banking jobs. In retrospect, no college student should ever have been paid that much (on the bright side, the savings were enough for Art.sy's initial funding).

But that summer it meant I could go out to nice dinners with my girlfriend, and never worry about paying for drinks at expensive clubs. It meant I could afford fancy clothes, an iPhone, and plane flights to Asia. Having always worked in labs prior to that job, it redefined how I thought about money.

Photo credit: Self by Jason Filsinger (www.flickr.com/photos/filsinger/409763398/).
Licensed under Creative Commons Attribution 2.0 Generic licence (creativecommons.org/licenses/by/2.0/deed.en).

e do We Create Here?

By CARTER CLEVELAND

So what is wrong with this picture? I had an extremely fun and challenging job, working with awesome people, that let me afford an incredible lifestyle. It was a dream comes true.

But at the end of the summer, the CTO brought me into the corner office and closed the door. I had worked with him all summer and this was my last day, so I was expecting a performance evaluation. Instead, after some chit chatting, he asked me a question:

“Have you ever wondered what value we create here?”

Value? This wasn’t what I was expecting at all.

“Not really.”

“I’ll tell you. We increase the liquidity of the secondary bond market. We shave basis points off of spreads.”

I’ll never forget that question. It turns out that our CTO was saving every penny and had plans of leaving as soon as he had enough cash to pursue his dream.

He didn’t care about the fancy clothes, the clubs, or being a master of the universe. All he cared about was how he would add value to the world. At this point, my story starts to sound cliché, but it was a cliché I needed to experience in person because it radically changed my perspective.

“How am I creating value?”

I realized that the programs I had spent all summer writing were great, if they could make people money and save them time. But if all it resulted in at the end of the day was slightly more efficient markets, well, what was the point of that?

I was so caught up in the fun and camaraderie of my job, so high with the rush of money, I never considered such a simple question.

This probably won’t change the minds of people who have already chosen career paths. But to any students who are thinking about their futures, I hope my story illustrates how easy it is to get swept up by short-term pleasures, and how important it is to always ask this question when making important decisions. ■

Carter Cleveland is the founder of Art.sy, a platform for connecting artists and galleries with collectors of original fine art. He is also the NYC Curator of The Startup Digest.

7 Tips for Successful Self-Learning

By BRADFORD CROSS *and* HAMILTON ULMER

SELF-LEARNING IS HARD. Regardless of where, when or how you learn - being a good self-learner will maximize your potential.

In this post, Hamilton Ulmer (an almost-done Stanford stats masters student) and I, will explore seven ways to become a great self-learner.

1 The longest path is the shortest and the shortest path is the longest

The shortest route to learning the craft of a field is the one that, at first glance, appears the longest. To really learn something, you must understand the basic concepts of your field. If you try to skip, you may end up spending

more time figuring out concepts than if you had started with learning basics.

Have you ever wanted to take up a new subject, bought a book, only to make a failed attempt at the first few chapters before submitting to a lack of foundation for the material?

Starting at the beginning might seem daunting, but trying to skip to the goal directly is likely to fail. If you are studying and unsure that you have the background for something, just stop when you don't understand something and go back to acquire that background.

2 **Avoid isolation** In school you have many effective feedback loops. If you are confused, you can ask the lecturer for a clarification. Your homework assignments and exams motivate you to internalize the content of the class, whether you want to or not.

Peers can help you smooth over small rough spots in your understanding.

A decent self-learner must find others who are familiar with the material. Naturally one prefers to find an expert, but discussing the material with a peer can also go a long way.

Having a community is vital. Often, a byproduct of finding or building a community is finding a mentor. The

“No matter what, you’re going to have to learn most everything on your own anyway.”

one element of graduate school that is hardest to replicate is the advisor-advisee relationship. They help guide you, smoothing out the uncertainties you have about certain topics, and help you make your own learning more efficient.

As a self-learner, you do not have the convenience of scheduled class time and required problem sets. You must be aggressive about finding people to help you.

3 Avoid multitasking

Another reason school is great for learning is that you plan your day around your classes. There are distractions, of course, but if you’re concerned with learning at school, you prioritize your classes over other things.

You don’t have to be

in a classroom or library to study, but notice the relative isolation and focus those environments afford over reading a book with your laptop on while writing emails and checking facebook or twitter with the TV on.

Remove the distractions and allocate large blocks of time. You might find that for more difficult material, you need larger blocks of time to study because it takes longer to shift into the context of harder problems.

4 You don’t read textbooks, you work through them

Imagine taking a 12-hour flight with two books, Machiavelli’s “The Prince” and Shilov’s “Elementary Functional Analysis.” It would be typical to finish the 100 pages of Machiavelli in two hours or so, and spent the rest of the time working through 10 pages of a Shilov’s “Elementary Functional Analysis,” minus some breaks for napping and eating undesirable airplane food.

Reading a technical book is nothing like reading a novel. You have to slow down and work carefully if you want to understand the material. Have you ever found yourself 10 pages further in a book and having forgotten what you’ve just read?

Successful self-learners don’t read, they toil. If there are proofs, walk them through, and try proving results on your own. Work through exercises, and make up your own examples. Draw various diagrams and invent visualisations to help you develop an intuition. If there is a real-world application for the work, try it out. If there are algorithms, implement them with your favorite programming language. If something remains unclear, hunt down someone who’s smarter than you and get them to explain. Sometimes you just need to put the material down, step away, relax, and think deeply to develop an intuition. »

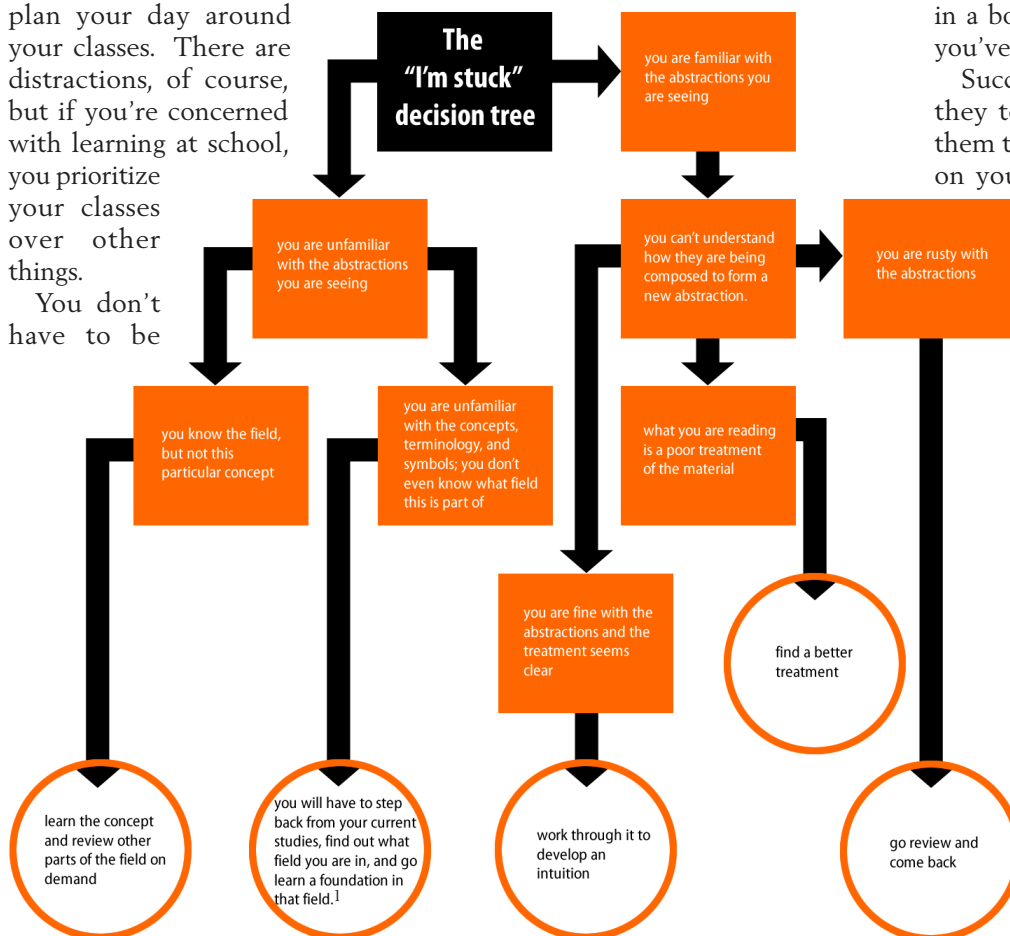


Figure 1. The "I'm stuck" decision tree.

“In theory, there is no difference between theory and practice. But, in practice, there is.”

- Jan L. A. van de Snepscheut

5 Build Eigencourses

Great self-learners spend a lot of time to find the best resources for learning. You can find all the textbooks, papers and other resources you need on the Internet. Many of the course materials from among the world's best universities are available for free online². Check out the great lists of links to video courses on this Data Wrangling post³.

You can pick and choose the best “eigencourse” with lecture slides, video lectures, textbooks, and other materials. The best way to find these materials is on Google. You will often only need to pay for the book, and sometimes even the book is free at the course website in pdf form.

Take the time to triangulate on the right material. Find the greats in the field, see what they use and recommend. Find other students and read the reviews on Amazon. Google is your friend.

6 What to do when you don't understand

Learning is all about abstractions. We build up abstractions on top of other abstractions. If you do not know the abstractions you are reading about that are being composed into new higher-level abstractions, then you aren't going to understand the new abstraction. If you get stuck, the way to get un-stuck is to follow the I'm stuck decision tree⁴.

7 There is nothing so practical as a good theory. -Kurt Lewin

Sometimes you are several hops away from something you can code up and apply to a problem directly. Not all textbooks can be read with application in mind, despite that they serve as the theoretical foundation for applied work. This is why you must have a deep sense of patience and commitment - which is why a prolonged curiosity and passion for a topic are so valuable.

Understanding analysis (particularly sets, measures, and spaces) will serve as your foundation for a deep understanding of probability theory, and both will then serve as your foundation for understating inference, and a deep understanding of inference is a mainstay of achieving high quality results on applied problems.

Avoid the dualistic mistakes of technical execution without intuition, and intuition without technical execution. ■

Notes

1. Keep in mind that you often just need to build a general foundation in the field, or mastery of some subset of a field - you don't have to master the entire field.
2. http://www.jimmyr.com/blog/1_Top_10_Universities_With_Free_Courses_Online.php
3. <http://www.datawrangling.com/hidden-video-courses-in-math-science-and-engineering>
4. Figure 1.

Bradford Cross has been doing applied research since 2001. His interests are in Maths, Statistics, Computer Science, Learning Theory, Network Theory, Information Retrieval, Natural Language Processing, and engineering at scale. Most recently, Bradford is co-founder and head of research for FlightCaster, where he is responsible for the statistical learning and supporting architecture that power FlightCaster's predictive algorithms.

Hamilton Ulmer is a Master's student in Statistics at Stanford. He has a great deal of experience as a data engineer, having helped startups of various sizes and shapes get on their feet with processing and visualizing their data, as well as helping them make data-driven decisions. In August he will join the Mozilla analytics team.

Adam? ...is there a reason your laptop is in the fridge?

By ADAM KEMPA

I'D READ A few times that bringing the temperature of a failing drive down will increase its reliability long enough to salvage important files. When the drive in my trusty Powerbook decided one day last week to stop booting and make horrible clicking sounds, I decided to test the theory.

Not feeling particularly motivated to dissect the Powerbook, since that would void the warranty I planned to invoke to get the drive replaced, I set it on a relatively uncluttered shelf of the fridge when I got home from work. Ten minutes later, I took it out, and the drive booted like new. I copied my iPhoto libraries to an external drive and once that was successful, begun the copying of the only other important file on the drive: a giant iMovie project (~ 30 GB). About halfway through, the drive had warmed up, the copy progress bar had stalled and the clicking was back.

Fair enough. Back in the fridge, for 20 minutes this time. I took it out, booted up (painlessly), hooked it up to the external drive and started the copy again. This time it made it to 75% before the clicking took hold. At this point I considered going after the video clips that made up the iMovie project in small batches, but decided I didn't feel like doing that if it wasn't absolutely necessary. I also didn't want to play guess and check to discover the ideal length of time to chill a powerbook, so I devised a devious plot.

This plot consisted of cooling the Powerbook down again, carting my external drive to the kitchen, booting the laptop in the fridge, beginning the copy, and closing the door. Success! I share this experience with you, the Internet, in the hopes that it is useful. ■

Adam Kempa works as a web developer in Ann Arbor, Michigan (Yes, people still live in Michigan). His nerdy musings intermittently appear at kempa.com.



Reprinted with permission of the original author. First appeared in www.kempa.com/2006/10/02/adam-is-there-a-reason-your-laptop-is-in-the-fridge/.



The Scariest Pricing Idea Ever

By WALT KANIA

HERE'S A PRICING technique that sounds, at first, like the dumbest newbie move of all time.

Call it 'fill-in-the-blank' invoicing. Or 'pay what you want' pricing.

The notion is, you do the work first, then let the client decide how much to pay for it.

I know, that sounds like a sure way to end up working for nickels and peanuts. I once thought that way, too.

But it's actually an ingenious tactic

that should be in every freelancer's arsenal, ready to wheel out when the wind is right. (Notice I said when the wind is right. We'll come back to that.)

It goes like this.

Instead of quoting a fee or negotiating a price in advance, you tell the client:

"Here's what I suggest. Let me jump in and do the work as we discussed. I'll hit this as hard as I know how, and make it as good as can be done."

"When we're finished, just pay

whatever you feel the work was worth, based on what it contributed to your overall project."

"I'll accept whatever you decide, no questions asked. Provided it is more than a buck sixty-five."

Scary? Absolutely.

Risky? Maybe a little.

Foolhardy and stupid? Not at all.

I had dabbled with this tactic before, but only on those small, oddball projects a client would send me now and then.



“Let me just concentrate on getting this done for you, and we’ll settle up later. I trust you to be fair.”

As an added bonus, you will most likely do the best work of your life, and deliver obscenely wonderful service to your clients at the same time. (Mainly because you’ll be too scared not to.)

Making it pay. More.

Naturally, the sole reason for using fill-in-the-blank invoicing is to net more from a project than you could with “traditional” pricing.

The idea is to get paid for the value the client derives from the work, rather than for the number of hours it took. Or how hard it was. Or how many shots you had to take. Or what somebody else charged some other client somewhere.

And by value, I don’t mean only hard economic value, like sales or savings or new business. (Which in most cases is hard to quantify anyway.)

As I’ve discovered, clients are also willing to pay lavishly to get a nose-bleed project done and off the desk, to look like geniuses in front of their bosses, to have presentations that their sales people rave about. To finally get the bosses sold on videos for user training. To untangle a project that somebody else screwed up.

That kind of value has no relation to how long it took you to do the job. It’s irrelevant, immaterial. And it is difficult to guess what that value might be from our side of the glass. So it can pay to let the client set that value.

Example.

A client of mine was knee-deep in redoing all her company’s web site content. She was getting raw material from the various divisions that was ugly, undecipherable and unusable. The go-live date was looming. She called me in to figure out how to fix it all.

But she had no idea how many sections we’d be doing, how many pages, nor how bad the raw material would be, so it was impossible to estimate any sort of fee.

I said, “Let me just concentrate on getting this done for you, and we’ll settle up later. I trust you to be fair.” She agreed.

I did the work as it came in over a couple of weeks, revising, re-writing, re-building the content. We came up with a neat and tight format, a solid voice, sharp messaging. Everybody loved it.

I then told the client to let me know what she felt was a reasonable fee for the project. It was entirely her call.

Meanwhile, I went back and parsed out the work based purely on hours spent. Had I been pricing conventionally, it would have come to 3800 to 4200 bucks, depending on how I counted.

Next day, I get an email from the client. She says, “I’m thinking \$9,500. How does that sound?”

I wrote her back and said “Fine. Sold.”

Now, lest you think I’m just handing you rosy stories, here’s another. »

“I have no idea what to bill for this,” I’d say. “Just send me whatever seems right to you.” Sometimes they would send a hundred or two more than I anticipated, sometimes less. But it was always intriguing to see how the client perceived what I had done. And a little humbling, too, on occasion.

But over the past year or so I finally got the guts to try this on large projects for big clients. (Partly because, while developing “Talking Money,” I was thinking/obsessing about pricing issues pretty much all day long. I was itching to see how this worked.)

I can tell you this: the ‘pay what you want’ idea can be surprisingly and dumbfoundingly profitable.

Better still, I can guarantee you that it will shake up your thinking about fees and pricing. It will un-stick some old notions. And heaven knows we need that; most of us are way too myopic, constipated and chickenshit about fees.

“Don’t try this with one-time clients...Been there, done that, lost shirt.”

» A designer friend is working on a web site for a financial firm, two partners. He refers them to me for the writing. We have a few phone conversations. Seems simple enough. Not a ton of content, straightforward mission. The clients don’t know much about marketing or web stuff.

I say, “Tell you what. I’ll write everything for you, and when you’re happy with it, send me a check for what you think is reasonable.”

Ordinarily, I would have quoted about \$2500 for the project, although I don’t say that.

I do some drafts. There are some comments, some revisions. Slam-dunk. Site goes live. Time to settle up. And I’m thinking the Wall Street guys are seeing a fee with a lot of zeros.

They send a check for \$1200. And say, “Thanks for the great work.”

Ouch and a half.

What works, what doesn’t

After a few painful scorchings, and several delightfully lucrative wins, here is the bottom line.

This technique works only when:

- You have a long-term relationship with the client. You’ve done work for them before, at your usual rates. They trust you. They know your work. And mostly likely they need to work with you again.
- Don’t try this with one-time clients, clients who don’t use this work often, or clients who didn’t seek you out. Been there, done that, lost shirt.

- The client has a big personal stake in the project. They have skin in the game. They stand to look grand if all goes well, score some points, be a hero, win some kudos. This does not work for low-level backburner projects that no one cares about. (Like my Wall Street clients; to them, their website was just some bullshit thing they needed to have. They didn’t perceive it as critical.)
- The project looks hard, impossible, and indecipherable. (My Wall Street clients thought it was a cinch to bang out a few pages of drivel, and therefore paid accordingly. My technology client tried untangling her web content herself, and got scared. To her, it seemed insurmountable.)

How do clients react? Do clients like this idea?

A few will balk. They don’t want the responsibility of figuring out a fee. They don’t want the anguish. That’s okay. Give them a quote.

Most will be astonished that you offer the option. It shows you trust them. That you value their judgment. That you even thought to ask. Huge karma points translate to more dollars.

Sometimes (as one client confessed to me) they’ll reflexively crank up the fee when filling in the blank.

Sort of like the way we reflexively and fearfully crank down the price when the client says ‘How much will it cost?’

JUST SO YOU know I’m not the only crackpot using this idea, Matt Homann of LexThink, a consultant who works with law firms, offers this ‘you decide’ option to all of his clients. His experience with the technique mirrors mine exactly. There’s more about his approach here too, in *The Non-Billable Hour*. (It’s for lawyers, but the ideas apply to us, I think.)

Oh, and see the classic *Little Rascals* episode from 1936, “Pay as You Exit.” As the story goes, the gang was putting on a show in the barn, but the neighborhood kids were reluctant to pay the penny admission, fearing that the show might be lame.

Over Spanky’s objections, Alfalfa decided to let everyone in for free, and allow them to pay on the way out if they liked the show.

As it turned out, the gang botched the show horribly, but the result was so hilarious that the kids filed out laughing.

Leaving Alfalfa with cigar box full of pennies. ■

Walt Kania is a freelance writer who runs *The Freelancery* site (thefreelancery.com), and develops marketing content (waltkania.com) for B2B and technology companies. He has plied his trade independently his entire adult life, due to a congenital inability to tolerate conventional employment for more than three to five days.

5 Actions that Made Me Happier

By GARY HARAN

HAPPINESS IS NOT universally quantifiable but money is. At some point in my life I raced towards money because I could measure it. When I noticed it wasn't making me happier I set out to make happiness my main goal. Here is a list of actions I took.

1 Reduced Commute Time

Commuting is a side effect of many jobs and sadly the higher the salary the more commute time we're willing to do. Finding ways to shave off commute time has a proven benefit as measured by this study¹.

When changing jobs wasn't a possibility I used public transportation and got an Internet capable cell phone so I could deal with paperwork related annoyances during the commute. Instead of trying to find time at home I'd deal with them while in traffic. I also borrowed and bought a few books.

Today my job allows me to work from home and my commute takes about 38 seconds. I still need to commute a few days a week but I can choose to take the car and avoid rush hour traffic.

2 Removed Small Frustrations

I start every day by making some tea. I had this cheap kettle that would randomly turn off on me. One day after pouring cold water over tealeaves I decided to drive to the store. Now every morning I look at the testament of a foregone frustration with a smile from ear to ear.

Removing frustrations can be as simple as moving the furniture or spending a few bucks.

3 Played Sports

A Harvard University study started in 1937 that spanned 72 years determined that healthy play could relieve daily frustrations making us happier overall.

A few years ago I joined a volleyball team and now I play a minimum of once a week.

4 Attended Regular Meetups

Would doubling your income make you happier? Well it turns out that seeing a group of people that meets just once a month provided the same benefit as doubling your salary.

Once I started digging I found out that Montreal was vibrant and full of user groups and programming language enthusiasts that meet regularly. I've met some really interesting people through these groups and some of the contacts even helped me professionally.

5 Drank Socially With Co-Workers

When work sucks your life sucks. A good team feels comfortable cracking a joke to the CEO. Imagine how many valid concerns are not expressed if a team has to worry about everything they say.

Good communication is perhaps the reason why those who occasionally have a single drink after work with colleagues make significantly more money on average than those who do not drink at all. Team members who do drink are probably made aware of problems and can resolve situations before they occur. It's a different setting and we all know that a little alcohol can make shyness go away.

So it's perhaps a stretch to make this point but seriously having a drink has some beneficial effect on the time you spend at work and that can't all be bad since you're there a good portion of your day. ■

Notes

1. http://www.cces.ethz.ch/agsam2009/panels/AGSAM2009_panel_mobility_Stutzer.pdf

Gary is a programmer and entrepreneur in Montreal, Canada. He is a father and entrepreneur currently working at *SocialGrapes.com*. You can follow him on twitter *@xutopia*.

How Not to Run an A/B Test

By EVAN MILLER

IF YOU RUN A/B tests on your website and regularly check ongoing experiments for significant results, you might be falling prey to what statisticians call *repeated significance testing errors*. As a result, even though your dashboard says a result is statistically significant, there's a good chance that it's actually insignificant. This note explains why.

Background

When an A/B testing dashboard says there is a “95% chance of beating original” or “90% probability of statistical significance,” it’s asking the following question: Assuming there is no underlying difference between A and B, how often will we see a difference like we do in the data just by chance? The answer to that question is called the significance level, and “statistically significant results” mean that the significance level is low, e.g. 5% or 1%. Dashboards usually take the complement of this (e.g. 95% or 99%) and report it as a “chance of beating the original” or something like that.

However, the significance calculation makes a critical assumption that you have probably violated without even realizing it: that the sample size was fixed in advance. If instead of deciding ahead of time, “this experiment will collect exactly 1,000 observations,” you say, “we’ll run it until we see a significant difference,” *all the reported significance levels become meaningless*. This result is completely counterintuitive and all the A/B testing packages out there ignore it, but I’ll try to explain the source of the problem with a simple example.

Example

Suppose you analyze an experiment after 200 and 500 observations. There are four things that could happen:

	Scenario 1	Scenario 2	Scenario 3	Scenario 4
After 200 observations	Insignificant	Insignificant	Significant!	Significant!
After 500 observations	Insignificant	Significant!	trial stopped	trial stopped
End of experiment	Insignificant	Significant!	Significant!	Significant!

Assuming treatments A and B are the same and the significance level is 5%, then at the end of the experiment, we’ll have a significant result 5% of the time.

But suppose we stop the experiment as soon as there is a significant result. Now look at the four things that could happen:

	Scenario 1	Scenario 2	Scenario 3	Scenario 4
After 200 observations	Insignificant	Insignificant	Significant!	Significant!
After 500 observations	Insignificant	Significant!	Insignificant	Significant!
End of experiment	Insignificant	Significant!	Insignificant	Significant!

The first row is the same as before, and the reported significance levels after 200 observations are perfectly fine. But now look at the third row. *At the end of the experiment, assuming A and B are actually the same, we’ve increased the ratio of significant relative to insignificant results. Therefore, the reported significance level – the “percent of the time the observed difference is due to chance” – will be wrong.*

How big of a problem is this?

Suppose your conversion rate is 50% and you want to test to see if a new logo gives you a conversion rate of more than 50% (or less). You stop the experiment as soon as there is 5% significance, or you call off the experiment after 150 observations. Now suppose your new logo actually does nothing. What percent of the time will your experiment wrongly find a significant result? No more than five percent, right? Maybe six percent, in light of the preceding analysis?

Try 26.1% – *more than five times what you probably thought the significance level was*. This is sort of a worst-case scenario, since we’re running a significance test after every observation, but it’s not unheard-of. At least one A/B testing framework out there actually provides code for automatically stopping experiments after there is a significant result. That sounds like a neat trick until you realize it’s a statistical abomination.

Repeated significance testing always increases the rate of false positives, that is, you’ll think many insignificant results are significant (but not the other way around). The problem will be present if you ever find yourself “peeking” at the data and stopping an experiment that seems to be giving a significant result. The more you peek, the more your significance levels will be off. For example, if you peek at an ongoing experiment ten times, then what you think is 1% significance is actually just 5% significance. Here are other reported significance values you need to see just to get an actual significance of 5%:

You peeked...	To get 5% actual significance you need...
1 time	2.9% reported significance
2 times	2.2% reported significance
3 times	1.8% reported significance
5 times	1.4% reported significance
10 times	1.0% reported significance

Decide for yourself how big a problem you have, but if you run your business by constantly checking the results of ongoing A/B tests and making quick decisions, then this table should give you goosebumps.

What can be done?

If you run experiments: the best way to avoid repeated significance testing errors is to not test significance repeatedly. Decide on a sample size in advance and wait until the experiment is over before you start believing the “chance of beating original” figures that the A/B testing software gives you. “Peeking” at the data is OK as long as you can restrain yourself from stopping an experiment before it has run its course. I know this goes against something in human nature, so perhaps the best advice is: no peeking!

Since you are going to fix the sample size in advance, what sample size should you use? This formula is a good rule of thumb:

$$n = 16\sigma^2/\delta^2$$

Where δ is the minimum effect you wish to detect and σ^2 is the sample variance you expect. Of course you might not know the variance, but if it’s just a binomial proportion you’re calculating (e.g. a percent conversion rate) the

variance is given by:

$$\sigma^2 = p * (1 - p)$$

Committing to a sample size completely mitigates the problem described here.

If you write A/B testing software: Don’t report significance levels until an experiment is over, and stop using significance levels to decide whether an experiment should stop or continue. Instead of reporting significance of ongoing experiments, report how large of an effect can be detected given the current sample size. That can be calculated with:

$$\delta = (t_{\alpha/2} + t_{\beta})\sigma\sqrt{2/n}$$

Where the two t’s are the t-statistics for a given significance level $\alpha/2$ and power $(1-\beta)$.

Painful as it sounds, you may even consider excluding the “current estimate” of the treatment effect until the experiment is over. If that information is used to stop experiments, then your reported significance levels are garbage.

If you really want to do this stuff right: Fixing a sample size in advance can be frustrating. What if your change is a runaway hit, shouldn’t you deploy it immediately? This problem has haunted the medical world for a long time, since medical researchers often want to stop clinical trials as soon as a new treatment looks effective, but they also need to make valid statistical inferences on their data. Here are a couple of approaches used in medical experiment design that someone really ought to adapt to the web:

- **Sequential experiment design:** Sequential experiment design lets you set up checkpoints in advance where you will decide whether or not to continue the experiment, and it gives you the correct significance levels.
- **Bayesian experiment design:** With Bayesian experiment design you can stop your experiment at any time and make perfectly valid inferences. Given the real-time nature of web experiments, Bayesian design seems like the way forward.

Conclusion

Although they seem powerful and convenient, dashboard views of ongoing A/B experiments invite misuse. Any time they are used in conjunction with a manual or automatic “stopping rule”, the resulting significance tests are simply invalid. Until sequential or Bayesian experiment designs are implemented in software, anyone running web experiments should only run experiments where the sample size has been fixed in advance, and stick to that sample size with near-religious discipline. ■

Evan Miller is a graduate student in Economics at the University of Chicago, and the author of the Chicago Boss web framework.

How I Took My Web-App to Market in 3 Days

By TAWHEED KADER

I'M A HUGE fan of the 37Signals mantra of "scratch your own itch." Inspired by their book for "Getting Real" which I've read at least twice, and "Rework" which I'm reading now, I decided to write a small web application to scratch an itch around customer development emails.

Do note though, 37Signals mantra here probably roots back to a saying my Dad, also an entrepreneur, has always said to me: "Necessity is the mother of invention".

Either way, here's the problem I solved with Tout: as I've been ramping up customer development for Braintrust, I realized that typing, copying, pasting, re-typing all these emails was becoming a huge pain. Even worse, it became even harder to keep track of all these emails.

"There had to be a better way!" — and while there are tons of CRMs out there, the simple "get in, get out" type of solution didn't exist. So, I decided to create one.

Introducing Tout — the simplest way to templatize and track (like you do for websites) your customer development emails. It helps me create e-mail templates, send emails quickly, and track when someone's viewed my email, and whether they clicked on my link. It also let me track whether my overall email was a "success" or not.

It took me about 1 day to get the

app working to fit my own need. After realizing this could probably help other people, it took me another 2 days to get it production ready. WOW!

I think we're at amazing times right now. With all the different "common services" startups cropping up, building, releasing and opening up shop for a web application has never been easier.

Here are the common services/technologies I leveraged to take Tout to market in 3 days:

Heroku

All of my development is on Rails, and Heroku puts Rails on steroids. Thanks to their amazing cloud infrastructure, I had to do ZERO sysadmin stuff and was able to get my app online in literally 3 commands. More importantly, setting up DNS, E-Mailing, and SSL was all done through the web UI as well. I highly recommend them for starter applications, especially ones that are still testing out the market.

The only downside for Heroku is that they have no way to support real-time applications (i.e. run an XMPP or NodeJS server to push out real-time updates) — can you guys start working on this?

Sendgrid

Even though the biggest "feature" of my web-app is sending emails, I had to write next to no code for actually

sending out emails or even configuring e-mail servers. All of this got taken care of by Sendgrid.

They were also very diligent about validating my site and making sure I was compliant with CAN-SPAM laws and ensuring this doesn't turn into another spamming machine.

Chargify

Tout has a premium feature, and charges credit cards, handles recurring billing and even sends out invoices. However, I didn't have to write more than about 50 lines of billing code. Chargify takes care of all of this — all I have to do is build out hooks to keep the subscription level of the customer up to date.

The reality is, it has become so ridiculous easy to take web applications to market now that I don't have to spend time working on plumbing — instead, all of my time and energy goes toward the creative aspect of the product — which is the way it should be. ■

TK is the Founder and CEO of Braintrust (<http://braintrusthq.com>), a webapp that helps organize your team's conversations. He also blogs about his journey as a single founder for a bootstrapped company at <http://tawheedkader.com>. Prior to Braintrust, TK co-founded HipCal, which was sold to Plaxo in 2006.



Forget Servers. Forget Deployment. Build Apps.

heroku.com

Organic Startup Ideas

By PAUL GRAHAM

THE BEST WAY to come up with startup ideas is to ask yourself the question: what do you wish someone would make for you?

There are two types of startup ideas: those that grow organically out of your own life, and those that you decide, from afar, are going to be necessary to some class of users other than you. Apple was the first type. Apple happened because Steve Wozniak wanted a computer. Unlike most people who wanted computers, he could design one, so he did. And since lots of other people wanted the same thing, Apple was able to sell enough of them to get the company rolling. They still rely on this principle today, incidentally. The iPhone is the phone Steve Jobs wants.¹

Our own startup, Viaweb, was of the second type. We made software for building online stores. We didn't need this software ourselves. We weren't direct marketers. We didn't even know when we started that our users were called "direct marketers." But we were comparatively old when we started the company (I was 30 and Robert Morris was 29), so we'd seen enough to know users would need this type of software.²

There is no sharp line between the two types of ideas, but the most successful startups seem to be closer to the Apple type than the Viaweb type. When he was writing that first Basic interpreter for the Altair, Bill Gates was writing something he would use, as were Larry and Sergey when

they wrote the first versions of Google.

Organic ideas are generally preferable to the made up kind, but particularly so when the founders are young. It takes experience to predict what other people will want. The worst ideas we see at Y Combinator are from young founders making things they think other people will want.

So if you want to start a startup and don't know yet what you're going to do, I'd encourage you to focus initially on organic ideas. What's missing or broken in your daily life? Sometimes if you just ask that question you'll get immediate answers. It must have seemed obviously broken to Bill Gates that you could only program the Altair in machine language.

You may need to stand outside yourself a bit to see brokenness, because you tend to get used to it and take it for granted. You can be sure it's there, though. There are always great ideas sitting right under our noses. In 2004 it was ridiculous that Harvard undergrads were still using a Facebook printed on paper. Surely that sort of thing should have been online.

There are ideas that obvious lying around now. The reason you're overlooking them is the same reason you'd have overlooked the idea of building Facebook in 2004: organic startup ideas usually don't seem like startup ideas at first. We know now that Facebook was very successful, but put yourself back in 2004. Putting undergraduates' profiles online wouldn't have seemed like much

“Just fix things that seem broken, regardless of whether it seems like the problem is important enough to build a company on.”

of a startup idea. And in fact, it wasn't initially a startup idea. When Mark spoke at a YC dinner this winter he said he wasn't trying to start a company when he wrote the first version of Facebook. It was just a project. So was the Apple I when Woz first started working on it. He didn't think he was starting a company. If these guys had thought they were starting companies, they might have been tempted to do something more “serious,” and that would have been a mistake.

SO IF YOU want to come up with organic startup ideas, I'd encourage you to focus more on the idea part and less on the startup part. Just fix things that seem broken, regardless of whether it seems like the problem is important enough to build a company on. If you keep pursuing such threads it would be hard not to end up making something of value to a lot of people, and when you do, surprise, you've got a company.³

Don't be discouraged if what you produce initially is something other people dismiss as a toy. In fact, that's a good sign. That's probably why everyone else has been overlooking the idea. The first microcomputers were dismissed as toys. And the first planes, and the first cars. At this point, when someone comes to us with something that users like but that we could envision forum trolls dismissing as a toy, it makes us especially likely to invest.

While young founders are at a disadvantage when coming up with made-up ideas, they're the best source of organic ones, because they're at the forefront of technology. They use the latest stuff. They only just decided what to use, so why wouldn't they? And because they use the latest

stuff, they're in a position to discover valuable types of fixable brokenness first.

There's nothing more valuable than an unmet need that is just becoming fixable. If you find something broken that you can fix for a lot of people, you've found a gold mine. As with an actual gold mine, you still have to work hard to get the gold out of it. But at least you know where the seam is, and that's the hard part. ■

Notes

1. This suggests a way to predict areas where Apple will be weak: things Steve Jobs doesn't use. E.g. I doubt he is much into gaming.
2. In retrospect, we should have become direct marketers. If I were doing Viaweb again, I'd open our own online store. If we had, we'd have understood users a lot better. I'd encourage anyone starting a startup to become one of its users, however unnatural it seems.
3. Possible exception: It's hard to compete directly with open source software. You can build things for programmers, but there has to be some part you can charge for.

Paul Graham is an essayist, programmer, and programming language designer. In 1995 he developed with Robert Morris the first web-based application, Viaweb, which was acquired by Yahoo in 1998. In 2002 he described a simple statistical spam filter that inspired a new generation of filters. He's currently working on a new programming language called Arc, a new book on startups, and is one of the partners in Y Combinator.

Not Disruptive, and Proud of It

By JASON COHEN

I REMEMBER “DISRUPTIVE” WHEN it was called “paradigm shift.” That phrase died during the tech-bubble along with “portal” and “think outside the box,” yet the concept has returned. Don’t follow along.

When I get pitched — usually by someone raising money — that they “have something disruptive,” a little part of me dies. You should be worrying about making something useful, not how disruptive you can be.

“Disruptive” is the in-vogue word for the opposite of “incremental improvement.” A disruptive product causes such a large market shift that entire companies collapse (the ones who don’t “get it”) and new markets appear.

Disruptive is fascinating, disruptive changes the world, disruptive makes us think. Disruptive also sometimes generates billions of dollars, which is why venture capitalists have always loved it and always will.

But disruptive is rare and usually expensive. It’s hard to think of disruptive technologies or products that didn’t take many millions of dollars to implement. Most of us don’t have access to those resources, and many of us don’t care, because we’d rather work on an idea we actually understand and can build ourselves, an idea that might make us a living and be useful to people.

There’s nothing wrong with incremental improvement. What’s wrong with doing something interesting, useful, new, but not transcendental? What’s wrong with taking a known

problem with a known market and just doing it better or with a fresh perspective or with a modern approach? Do you have you create a new market and turn everyone’s assumptions upside down to be successful? Should you?

I’m not so sure. Here’s my argument:

1 It’s hard to explain the benefits of disruption.

Have you tried to explain Twitter someone? Not the “140 characters” part — the part about why it’s a fundamental shift in how you meet and interact with people?

Hasn’t the listener always responded by saying, “I don’t need to know what everyone had for lunch. Who cares? What’s next, ‘I’m taking a dump?’” They don’t get it, right? But it’s hard to explain.

There are ways to elucidate the utility of Twitter, but even the good ones are lengthy and require listeners with patience and open minds — two attributes in short supply.

“It’s hard to explain” should not be a standard part of your sales pitch. “You just need to try it” and “trust me” don’t cut it. That may be OK for Twitter — today — but what about the 100 other social-networking-slash-link-sharing networks that didn’t survive? Ask them about selling intangible benefits.

2 It’s hard to sell disruption, because people don’t want to be disrupted.

If you’re reading this you’re probably more open to new ideas and new

products than most, because you’re inventing a new product, starting a company, or you’re just ruffled because I’m pissing on “disruptive” and you’re looking for nit-picky things to argue with me about.

But most people are creatures of habit. They don’t want their lives turned upside down. They launch into a tirade of obscenities if you just rearrange their toolbar. When they hear about a new social media craze they cringe in agony, desperately hoping it’s a passing fad and not another new god-damn thing they’ll be aimlessly paddling around in for the next decade.

Change is hard, so a person has to be experiencing real pain to want change. Selling a point-solution for a point-problem is easier than getting people to change how they live their lives. Identifying specific pain points and explaining how your software addresses those is easier than trying to tap into a general malaise and promising a better world.

3 Most technology we now consider “disruptive” wasn’t conceived that way.

Google was the 11th major search engine, not the first. Their technology proved superior, but “a better search engine” was hardly a new idea. In retrospect we say that Google transformed how people find information, and further, how advertising works on the Internet.

Disruptive in hindsight, sure, but the genesis was just “incrementally better”

than the 10 search engines that came before. (Or 18.)

Scott Berkun gives several other examples in a recent BusinessWeek article. He highlights the iPod — an awesome device, but not the first of its kind. Rather, there were a bunch of crappy devices that sold well enough to prove there were a market, but no clear winners. Here an innovation in design alone was enough to win the market. Not inventing new markets, not innovative features, not even improving on existing features like sound quality or battery life — just a better design, unconcerned about “disrupting” anything else.

Setting your sights on being disruptive isn’t how quality, sustainable companies are built. Disruption, like expertise, is a side effect of great success, not a goal unto itself.

4 The disruptors often don’t make the money.

The construction of high-speed Internet fiber backbones and extravagant data centers fundamentally changed how business is conducted world-wide both between businesses and consumers, but many of the companies who built that system went bankrupt during the 2000 tech bubble, and those who managed to survive have still not recovered the cost of that infrastructure. They were the disruptors, but they didn’t profit from the disruption.

Disruptive technology often comes from research groups commissioned to produce innovative ideas but unable to capitalize on them. Xerox PARC invented the fax machine, the mouse, Ethernet, laser printers, and the concept of a “windowing” user interface, but made no money on the inventions. AT&T Bell Labs invented Unix, the C programming language, wireless Ethernet, and the laser, but made no money on the inventions.

Is it because disruptors are “before

their time,” able to create but not able to hold out long enough for others to appreciate the innovation? Is it because innovation and business sense are decoupled? Is it because “version 1” of anything is inferior to “version 3,” and by the time the innovator makes it to version 2 there are new competitors — competitors who don’t bear the expense of having invented version 1, who have silently observed the failures of version 1, and can now jump right to version 3?

“Why” is an interesting question, but the bottom line is clear: Disruption is rarely profitable.

5 Simple, modest goals are most likely to succeed, and most likely to make us happy.

It’s not “aiming low” to attempt modest success.

It’s not failure if you “just” make a nice living for yourself. Changing the world is noble, but you’re more likely

I would have failed.

I made less money personally at ITWatchDogs, but the company was profitable and sold for millions of dollars. We took a simple problem (when server rooms get hot, the gear fails) and provided a simple solution (thermometer with a web page that emails/pages you if it’s too hot). There were many competitors, both huge (APC with \$1.5 billion market cap), mid-sized (NetBotz with millions in revenue and funding), and small (sub-\$1m operations like us). We had something unique — an inexpensive product that still had 80% of the features of the big boys — but nothing disruptive.

Had we tried to fundamentally change how IT departments monitor server rooms, I’m sure we would have failed.

There’s nothing wrong with modesty. Modest in what you consider “success,” and modest in what you’re trying to achieve every day:

“My daughter convinced me that insisting something be Deeply Meaningful With Purpose could sometimes suck the joy from it.”

- Kathy Sierra

to change it if you don’t try to change everything at once.

I made millions of dollars at Smart Bear with a product that took an existing practice (peer code review) and solved five specific pain points (annoyances and time-wasters). Sure it wasn’t worth a hundred million dollars, and it didn’t turn anyone’s world inside out, but it enjoys a nice place in the world and it is incredibly fulfilling to see people happier to do their jobs with our product than without it.

Had I tried to fundamentally change how everyone writes software, I’m sure

Of course it’s wonderful that disruptive products exist, improving life in quantum leaps. And it’s not wrong to pursue such things! But neither is it wrong to have more modest goals, and modest goals are much more likely to be achieved. ■

Jason is the founder of three companies, all profitable and two exits. He blogs on startups and marketing at <http://blog.ASmartBear.com>.

Turning On Your Reality Distortion Field

By STEVE BLANK

I WAS CATCHING UP over coffee and a muffin with a student I hadn't seen for years who's now CEO of his own struggling startup. As I listened to him present the problems of matching lithium-ion battery packs to EV powertrains and direct drive motors, I realized that he had a built a product for a segment of the electric vehicle market that possibly could put his company on the right side of a major industry discontinuity.

But he was explaining it like it was his PhD dissertation defense.

Our product is really complicated

After hearing more details about the features of the product (I think he was heading to the level of Quantum electrodynamics) I asked if he could explain to me why I should care. His response was to describe even more features. When I called for a time-out the reaction was one I hear a lot. "Our product is really complicated I need to tell you all about it so you get it."

I told him I disagreed and pointed out that anyone can make a complicated idea sound complicated. The art is making it sound simple, compelling and inevitable.

Turning on your Reality Distortion Field

The ability to deliver a persuasive elevator pitch and follow it up with a substantive presentation is the difference between a funded entrepreneur and those having coffee complaining that they're out of cash. It's a litmus test of how you will behave in front of customers, employees and investors.

30-seconds

The common wisdom is that you need to be able to describe your product/company in 30-seconds. The 30 second elevator pitch is such a common euphemism that people forget its not about the time, it's about the impact and the objective. The goal is not to pack in every technical detail about the product. You don't even need to mention the product. The objective is to get the listener to stop whatever they had planned to do next and instead say, "Tell me more."

HOW DO YOU put together a 30-second pitch? Envision how the world will be different five years after people started using your product. Tell me. Explain to me why it's a logical conclusion. Quickly show me that it's possible. And do this in less than 100 words.

The CEOs reaction over his half- finished muffin was, "An elevator pitch is hype. I'm not a sales guy I'm an engineer."

The reality is that if you are going to be a founding CEO, investors want to understand that you have a vision big enough to address a major opportunity and an investment. Potential employees need to understand your vision of the future to decide whether against all other choices they will join you. Customers need to stop being satisfied with the status quo and queue up for whatever you are going to deliver. Your elevator pitch is a proxy for all of these things.

While my ex student had been describing the detailed architecture of middleware of electric

“Envision how the world will be different five years after people started using your product.”

vehicles I realized what I wanted to understand was how this company was going to change the world.

All he had to say was, “The electric vehicle business is like the automobile business in 1898. We’re on the cusp of a major transformation. If you believe electric vehicles are going to have a significant share of the truck business in 10 years, we are going to be on the right side of the fault zone. The heart of these vehicles will be a powertrain controller and propulsion system. We’ve designed, built and installed them. Every electric truck will have to have a product like ours.”

75 words.

That would have been enough to have me say, “Tell me more.” ■

Lessons Learned

- Complex products need a simple summary
- Tell me why I should quit my job to join you
- Tell me why I should invest in you rather than the line outside my door
- Tell me why I should buy from you rather than the existing suppliers
- Do it in 100 words or less.

Steve Blank is a retired serial entrepreneur and the author of Customer Development model for startups. Today he teaches entrepreneurship to both undergraduate and graduate students at U.C. Berkeley, Stanford University and the Columbia University/Berkeley Joint Executive MBA program.

Reprinted with permission of the original author. First appeared in steveblank.com/2010/04/22/turning-on-your-reality-distortion-field/.

Best Writing Advice for Engineers

By WILLIAM A. WOOD

HOW TO MAKE engineers write concisely with sentences? By combining journalism with the technical report format. In a newspaper article, the paragraphs are ordered by importance, so that the reader can stop reading the article at whatever point they lose interest, knowing that the part they have read was more important than the part left unread.

State your message in one sentence. That is your title. Write one paragraph justifying the message. That is your abstract. Circle each phrase in the abstract that needs clarification or more

contexts. Write a paragraph or two for each such phrase. That is the body of your report. Identify each sentence in the body that needs clarification and write a paragraph or two in the appendix. Include your contact information for readers who require further detail. ■

William A. Wood works for NASA at Langley Research Center. He has a Ph.D. in Aerospace Engineering from Virginia Tech, and he has published in IEEE Software (Digital Object Identifier: 10.1109/MS.2003.1196317).

Reprinted with permission of the original author. First appeared in www.edwardtufte.com/bboard/q-and-a-fetch-msg?msg_id=0001y8.

Hacker Monthly is an independent project by Netizens Media and not affiliated with Y Combinator in any way.

Tell us what you think

It's our first release, and we want feedback. Let us know what you liked, and what we need to work on. Please share your thoughts so we can improve the coming issues.

hackermonthly.com/feedback/

