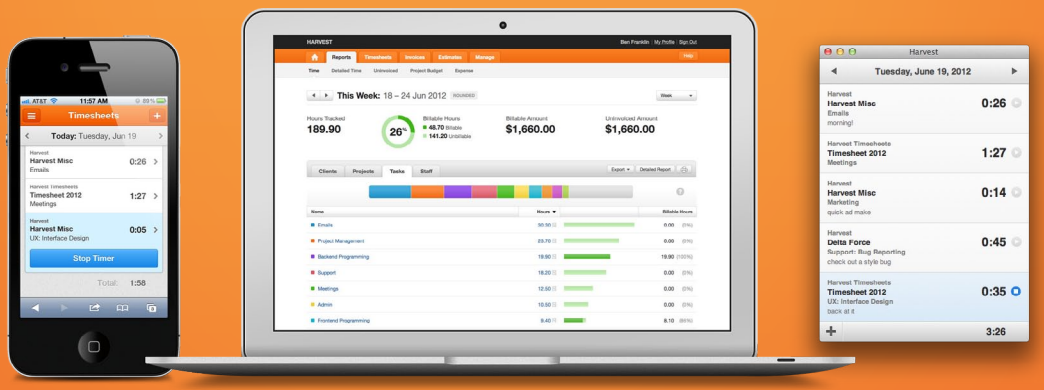
Elon Musk

On Entrepreneurship

HACKERMONTHLY

Issue 26 July 2012



HARVEST

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

getHarvest.com/hackers

Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com

Curator
Lim Cheng Soon

Contributors
Nikos Michalakis
Andrew Chen
Chris Strom
David Valdman
Justin Kan
Alexandru Nedelcu
Harvey Green
Hynek Schlawack
James Hague
Andreas Zwinkau
Joe Peacock
Tommy MacWilliam
Lou Montulli

Proofreaders
Emily Griffin
Sigmarie Soto

Printer
MagCloud

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

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

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

Cover Photo: Brian Solis (briansolis.com)

Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-26*

On Entrepreneurship

By ELON MUSK

I ACTUALLY ORIGINALLY CAME to California to study energy physics at Stanford, but I ended up putting it on hold in 1995 to start Zip2. I'll tell you a little about the process and exactly what happened there. In 1995, it wasn't at all clear that the internet was going to be a big commercial thing. In fact, most of the venture capitalists that I talked to hadn't even heard of the internet, which sounds bizarre on Sand Hill Road. However, I wanted to do something there, and I thought it would be a pretty huge thing. It was one of those things that came along once in a very long while, so I got a deferment at Stanford and thought I'd give the idea a couple of quarters. If it didn't work out, which I thought it probably wouldn't, then I'd go back to school. When I told one of my professors this, he said, "Well, I don't think you'll be coming back." And that was the last conversation I had with him.

The only way I could think to get involved in the internet in 1995 was by starting a company. Apart from Netscape and one or two others, there weren't a lot of companies specializing in this area. Since I didn't have any money, I decided to create something that would return money very, very quickly. So, we thought the media industry would need help converting its content from print to electronic media, and they clearly had the money. We decided that finding a way to help them root their media to the internet would be a sure way to generate revenue. There was no advertising revenue on the internet at the time.

That was really the basis of Zip2. We ended up pulling quite a bit of software for the media industry and primarily, the print media industry. We had plenty of investors and customers, such as Hearst Corporation, Knight Ridder, and most of the major US print publishers. We grew the company and then had the opportunity to sell it to Compaq in early 1999. And basically, we sold it for a little over $300 million dollars in cash. That's the currency I highly recommend.

I started Zip2 by writing a program that allowed you to keep maps and directions on the internet and a tool that allowed you to do online manipulation of content; kind of a really advanced blogging system. Once we started talking to small newspapers and media companies, we started gaining some

interest and getting a little bit of money from them. There were only six of us at Zip2: three sales people we hired on contingency by putting an ad in a newspaper; myself; my brother, who I convinced to come down from Canada; and a friend of my Mom's.

Things were pretty tough in the beginning because I didn't have any money. In fact, I had negative money because of huge student debts. At one point, I had to choose between renting a place to live or an office, so I rented the office instead because it was cheaper than renting a place to stay. For awhile, I slept on the futon and shouted the YMCA on Paige Mullen. It was the best shape I've ever been in.

There was a small ISP on the floor below us, so we drilled a hole through the floor and connected to the main cable, which gave us our internet connectivity for like a hundred bucks a month. So we had just an absurdly tiny burn rate as well as a really tiny revenue stream. However, since we actually had more revenue than expenses, we were able to say we had positive cash flow when we talked to VPs. That helped, I think.

Founding of PayPal

I automatically wanted to do something more after Zip2. Immediately after the sale, I'd normally take time off, but I wanted to find other opportunities in the internet since it was early 1999. I noticed there hadn't been a lot of innovation in the financial services sector. And when you think about it, money is low bandwidth. You don't need some sort of big infrastructure improvement to do things with it. It's really just an entry in the database.

Since the paper form of money is really only a small percentage of all the money that's out there, why not innovate financial services on the Internet? So, we thought of a couple of different things we could do. One of the things was to combine all consumers' financial services needs into one website, such as banking, brokerage, and insurance. And that was actually quite a difficult problem to solve, but we solved most of the issues associated with that.

Then, we had a little feature that took us about a day. It was about emailing money from one customer to another. Basically, you could type in an email address or, actually, any unique identifier, and transfer funds

or conceivably stocks or mutual funds from one account holder to another. If you tried to transfer money to somebody who didn't have an account in the system, it would forward them an email inviting them to open an account.

When we would pitch the idea to investors for a central financial services portal for consumers, we'd tell them how much effort it took to develop the convenient features. And people would go, "Hmmm." We would throw the email payment feature in as an afterthought and they would say, "Wow!" After this reaction, we focused the company's business on email payments.

In the early days, our company was called X.com. There was also another company called Confinity, which started out from a different area. Confinity had Palm Pilot cryptography and the demo application they were using had the ability to beam token payments from one Palm Pilot to another via the infrared port. They also had a website named PayPal where users could reconcile the beamed payments. It didn't take too long for them to notice the website portion was actually far more interesting to users than the Palm Pilot cryptography.

They started leaning their business in that direction, and in early 2000, X.com acquired Confinity. About a year later, we changed the company name to PayPal. And that's a summary of the evolution of the company.

Success through Viral Marketing

PayPal is really a perfect case example of viral marketing, just like Hotmail. In this case, customers act like a sales person for you by bringing in other customers. In PayPal's case, they would send money to a friend and, essentially, recruit that friend into the network, so we had this exponential growth. The more customers, the faster it grew. It was like bacteria in a Petri dish; it just keeps going like an S-curve.

I ran PayPal for about the first two years of its existence. By the end of year two, we had a million customers. It gives you a sense of how fast things grow in that scenario. And we didn't have a sales force. Actually, we didn't even have a VP of Sales or a VP of Marketing. And we didn't spend any money on advertising.

Selling PayPal

In 2002, PayPal went public. We were the only internet company to go public in the first part of that year. It went reasonably well, although we had more SEC rewrites than any company I can imagine. I think we set a record on SEC rewrites. This was right around the time when there were all sorts of corporate scandals. So, they put us through the ringer. Shortly thereafter, about June or July, we struck a deal and sold the company to eBay for over $4 billion. But that was when eBay's stock price was about $55 and they hadn't split. So, I guess, in today's dollars we were about $3 billion. So it worked out pretty well.

Comparing Zip2 and PayPal

I guess both Zip2 and PayPal involved software as the heart of the technology, even though Zip2 was servicing the media sector and PayPal was servicing the financial sector. However, the heart of it was really the software and the internet Both companies were also in Palo Alto, where I live.

We also took a similar approach to building both companies by having a small group of very talented people and keeping it small. PayPal, at its height, probably had 30 engineers for a system that, I would say, is more sophisticated than the Federal Reserve clearing system. I'm pretty sure it is actually because the Federal Reserve clearing system sucks.

So, what else is there? Generally, both Zip2 and PayPal operated as your canonical "Silicon Valley" start up. You know, a pretty flat hierarchy. And anyone could talk to anyone. We have to go for the best idea as opposed to a person proposing an idea that is considered to be a winner just because of who they are.

Obviously, everyone was an equity stake holder. If there were two paths that, let's say, we had to choose between and one wasn't obviously better than the other, then instead of spending a lot of time trying to figure out which one was slightly better, we would just pick one and do it. Sometimes we'd be wrong and we'd pick ourselves up. But often it's better to pick a path and do it than to just vacillate endlessly on a choice. We didn't worry too much about intellectual property, paperwork, or legal stuff. We were really just focused on building the best product that we possibly could.

Both Zip2 and PayPal were very product-focused companies. We were incredibly obsessive about creating something that would provide the best possible customer experience. And that was a far more effective selling tool than having a giant sales force or thinking of marketing gimmicks or twelve-step processes, or whatever.

> **“Really liking what you do is important because even if you’re the best of the best, there’s always a chance of failure.”**

The Right Time to Sell

We had several offers from a number of different entities for PayPal, and in fact, the closer we got to IPO, the more offers we got. However, we always felt that those offers undervalued the company and subsequently we went public. I think the public markets kind of indicated the value of the company, and that’s one of the good things about public markets. It’s difficult for private companies to say how much they’re worth because they need some kind of metric. Are you going to go for future earnings? Are you going to base it on revenue? What are your comparables going to be? There are all sorts of questions and the value of a company is really up for debate. When you’re public, however, you’re worth what the market says you’re worth. Yes, eBay made a number of offers prior to our IPO that would substantially blown the value once we went public.

eBay initially had Billpoint and then there was eBay Payments. It was a really tough, long running battle of PayPal versus eBay’s payment system. It was certainly very challenging. There were times when it felt like we were trying to win a land war in Asia and they kind of set the ground rules, or trying to beat Microsoft in their own operating system. It’s really pretty hard and it took a lot of our effort to actually beat eBay on their own system. One of the long-term risks for the company was that eBay would one day prevail, and one way to retire that risk obviously was to sell to eBay.

Qualities of an Entrepreneur

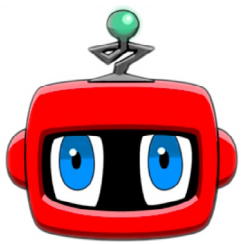
Successful entrepreneurs probably come in all sizes, shapes, and flavors. I’m not sure there’s any one particular critical quality. For me, some of the things I’ve described already are very important, such as an obsessive nature with respect to the quality of the product. Being obsessive compulsive is a good thing in this context. Also, really liking what you do is important because even if you’re the best of the best, there’s always a chance of failure, so I think it’s important that you really like whatever you’re doing. If you don’t like it, life is too short. And if you really like what you’re doing, you think about it even when you’re not working. It’s something that your mind is drawn to and if you don’t like it, you just really can’t make it work. ■

Elon Musk is the co-founder of SpaceX, Tesla Motors and PayPal.

Originally appeared on Stanford Technology Ventures Program (STVP) Entrepreneurship Corner in video format: *hn.my/elon* (ecorner.stanford.edu)

Photograph by Brian Solis (briansolis.com)

How To Train Your Robot



By NIKOS MICHALAKIS

> **"The only second language you should worry about your kids learning is programming."**
>
> **— Ray Kurzweil**

Last Sunday, I taught 6 kids ages 5 to 7 how to program. "In what programming language?" you may ask. Well…I didn't use a programming language, at least none that you know of. In fact, I didn't even use a computer. Instead, I devised a game called "How To Train Your Robot." Before I explain how the game works, let me tell my motivation.

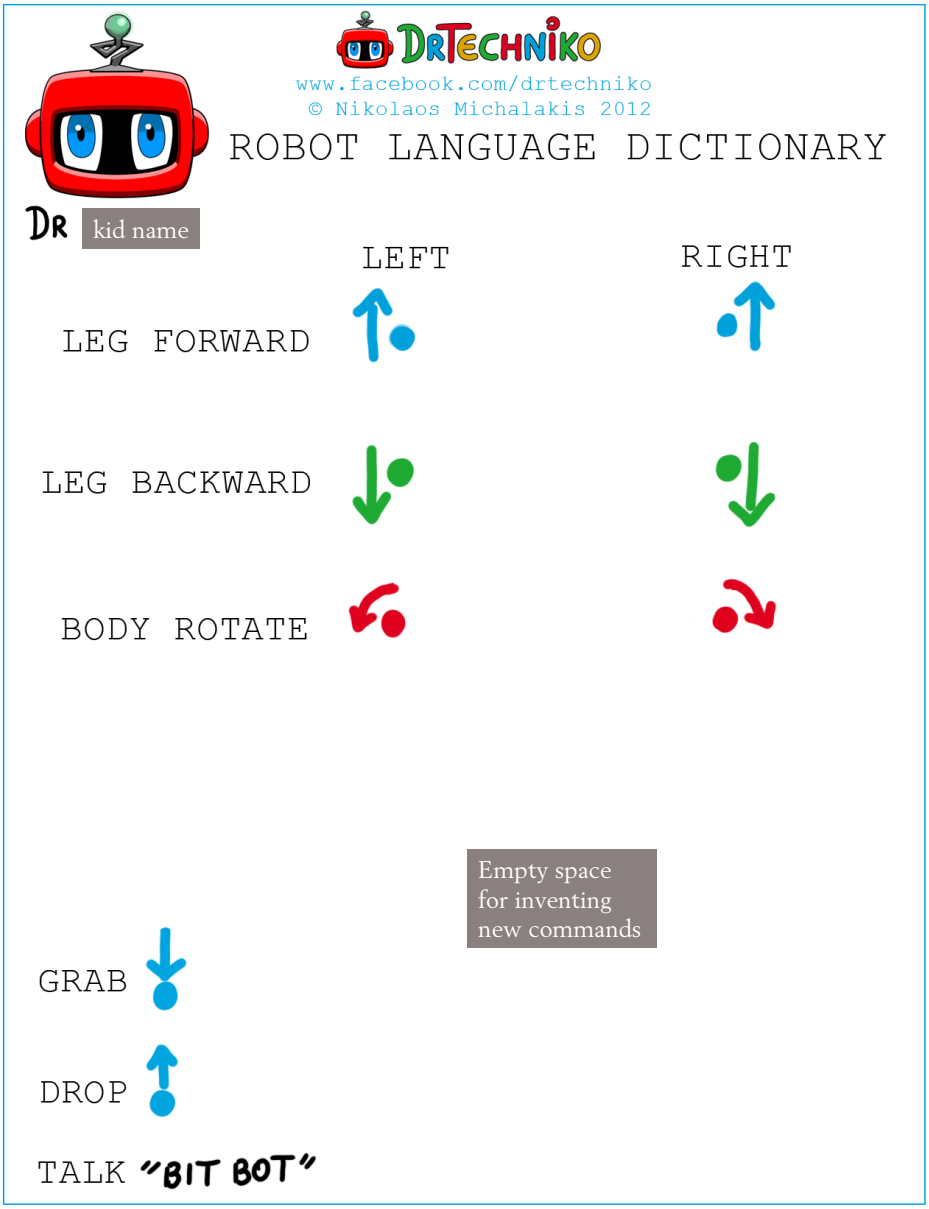
I learned how to program during my freshman year at MIT when I was 19. It's not because I didn't have a computer at home or I hadn't heard about programming languages. It was because (a) I thought programming was boring, and (b) no one had told me why I should bother. In fact, my computer teacher in high school had told me "you don't need to waste your time learning how to program. Now we have visual tools to build programs. Programming languages are already obsolete." That was in 1994 and he was referring to Visual Basic. Luckily for me, MIT wiped all that nonsense away in a matter of weeks. But does one need to wait to go to college to get the proper education?

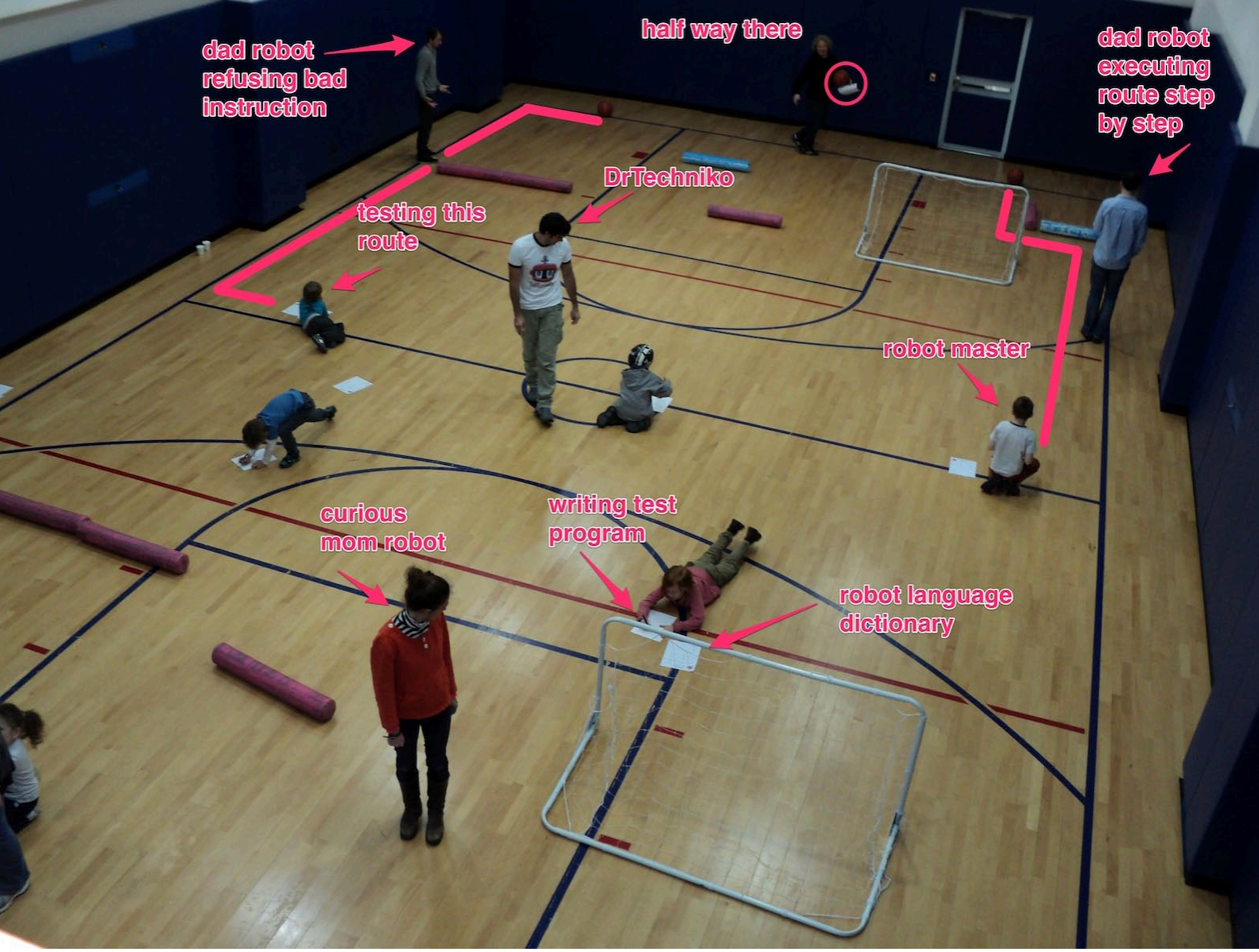
Learning how to program is going to be the most useful new skill we can teach our kids today. More than ever our lives depend on how smart we are when we instruct computers. They hold our personal data and they make decisions for us. They communicate for us, and they are gradually becoming an extension of our brains. If we don't learn programming as part of our childhood, we will never evolve. As the famous futurist, Ray Kurzweil, put it "The only second language you should worry about your kids learning is programming."

How To Train Your Robot

The game works as follows: every kid is turned into a "robot master" and their mom or dad becomes their "robot." I give each kid a "Robot Language Dictionary" and explain to them that this is the language their robot understands. The dictionary has symbols for "move left leg forward," "turn left," "grab," "drop," etc.

The goal is for the robots to go through an obstacle course, pick up a ball, and bring it back. The kids have to write a program that will tell the robot how to do all that. Every time they write a program, they hand it to their robot, and the robot executes it. To do that, I give each kid a pen and paper where they copy symbols from the dictionary to write their programs and off their robots go!





The fun part begins when each robot retrieves the ball. Now I let kids invent their own moves and symbols that they add to their dictionary and then teach their robots. There is no limit to what the kids come up with.

I designed the class to teach some very basic principles of computer science and programming:

- Programming languages are just another way to communicate to an entity (via programs).
- Programs are recipes for automating stuff.

However, I was pleasantly surprised on how much more the kids learned. On their own they figured out the following things (in a 30-min session):

- Program Parameterization: Instead of putting a forward step ten times, they put a 10 in front of the "step" symbol (A five-year-old figure it out and asked me if she could do it).
- Composition: Grouping of a set of moves ("move left leg forward, then move right leg forward and do this combo 10 times")
- Abstraction: "Run in a circle, then say "I'm dizzy!", then call this the "Run Dizzy" program and do it 100 times. (For some reason, kids loved making their parents repeat stuff 100 times over.)
- Unit testing: They'd write a test program to get the parents moving a few steps, have their parents run it, then fix it and run it again, and then add a few more steps until they reach the goal.

mom robot is running the "lie down, hug and kiss me" program

This is my favorite program (written by a five year old girl)

I've ran the class twice now and I've seen the same patterns, which support my belief that when kids have fun, they get very smart and creative about programming. Many of the parents plan to play the game at birthday parties. If you have questions about how to set up the game, don't hesitate to write. You can find my contact info at *facebook.com/drtechniko*

You can also find instructions on how to teach the class as well as materials I used here [hn.my/robotm].

I hope we learned something useful today,

DrTechniko ■

Nikos Michalakis graduated from MIT with a degree in Electrical Engineering and Computer Science. As DrTechniko, in his spare time he teaches kids about computer science and technology through storytelling and games. He lives in Brooklyn with his wife and their son and works for Knewton, an education technology startup.

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

Photographs by Nikos Michalakis.

Why You'll Always Think Your Product Is Shit

By ANDREW CHEN

> **"My product isn't quite there yet."**

You've said this before. We all have.

Anyone working on getting their first product out to market will often have the feeling that their product isn't quite ready. Or even once it's out and being used, nothing will seem as perfect as it could be, and if you only did X, Y, and Z, then it would be a little better. In a functional case, this leads to a great roadmap of potential improvements, and in a dysfunctional case, it leads to unlaunched products that are endlessly iterated upon without a conclusion.

About a year ago I visited Pixar's offices and learned a little about this product, and I wanted to share this story:

Over at Pixar…

Matt Silas, a long-time Pixar employee offered to take me on a tour of their offices and I accepted his gracious offer. After an hour-long drive from Palo Alto to Emeryville, Matt showed up while I was admiring a glass case full of Oscars, and started a full tour.

I've always been a huge fan of Pixar — not just their products, but also their process and culture. There's a lot to say about Pixar and their utterly fascinating process for creating movies, and I'd hugely recommend this book: To Infinity and Beyond [hn.my/pixarbook]. It gave me a kick to know that Pixar uses some very collaborative and iterative methods for making their movies — after all, a lot of what they do is software. Here are some quick examples:

- Pixar's teams are ultimately a collaboration of creative people and software engineers. This is reflected at the very top by John Lasseter and Ed Catmull.
- The process of coming up with a Pixar movie starts with the story, then the storyboard, then many other low-fidelity methods to prototype what they are ultimately make.
- They have a daily "build" of their movies in progress so they know where they stand, with sketches and crappy CGI filling holes where needed. Compare this to traditional moviemaking where it's only at the end.
- Sometimes, as with the original version of Toy Story, they have to stop doing what they're doing and restart the entire moviemaking process since the whole thing isn't clicking. Sound familiar, right?

The other connection to the tech world is that Steve Jobs personally oversaw the design of their office space. Here's a great little excerpt on this, from director Brad Bird (who directed The Incredibles):

"Then there's our building. In the center, he created this big atrium area, which seems initially like a waste of space. The reason he did it was that everybody goes off and works in their individual areas. People who work on software code are here, people who animate are there, and people who design are over there. Steve put the mailboxes, the meetings rooms, the cafeteria, and, most insidiously and brilliantly, the bathrooms in the center — which initially drove us crazy — so that you run into everybody during the course of a day. [Jobs] realized that when people run into each other, when they make eye contact, things happen. So he made it impossible for you not to run into the rest of the company."

Anyway, I heard a bunch of stories like this and more. As expected, the tour was incredible, and near the end, we stopped at the Pixar gift shop.

There, I asked Matt a casual question that had an answer I remember well, a year later:

Me: "What's your favorite Pixar movie?"

*Matt: *SIGH**

Me: "Haha! Why the sigh?"

Matt: "This is such a tough question, because they are all good. And yet at the same time, it can be hard to watch one that you've worked on, because you spend so many hours on it. You know all the little choices you made, and all the shortcuts that were taken. And you remember the riskier things you could have tried but ended up not, because you couldn't risk the schedule. And so when you are watching the movie, you can see all the flaws, and it isn't until you see the faces of your friends and family that you start to forget them."

Wow! So profound.

A company like Pixar, who undoubtedly produces some of the most beloved and polished experiences in the world, ultimately still cannot produce an outcome where everyone on the team thinks it is the best. And after thinking about why, the reason is obvious and simple: to have the foresight and the skill to refine something to the point of making it great also requires the ability to be hugely critical. More critical, I think, than your ability to even improve or resolve the design problems fast enough. And because design all comes to making a whole series of tradeoffs, ultimately you don't end up having what you want.

The lesson: You'll always be unhappy

What I took away from this conversation is that many of us working to make our products great will never be satisfied. A great man once said, your product is shit, and maybe you will always think it is. Yet at the same time, it is our creative struggle with what we do that ultimately makes our creations better and better. And one day, even if you still think your product stinks, you'll watch a customer use it and become delighted.

And for a brief moment, you'll forget what it is that you were unhappy about. ■

Andrew Chen is a blogger and entrepreneur focused on consumer internet, metrics and user acquisition. He is an advisor/angel for early-stage startups and is also a 500 Startups mentor.

Reprinted with permission of the original author.
First appeared in *hn.my/shit* (andrewchen.co)

How I Tricked Myself into Being Awesome

By CHRIS STROM

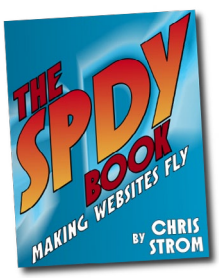
Like most developers, I am an introvert, so it is hard to say this:

I am awesome.

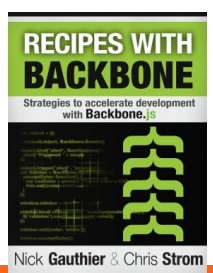
Fuuuuuu.... I can't even leave it at that. I look at so many amazing people in the Ruby, Javascript, and other communities that actually are amazing, and I feel like I haven't done anything. But even so, looking back at the 366 days of the last year, what I did was, well... amazing.

I wrote three books on very different technologies that I knew nothing about.

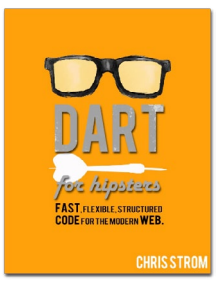
I wrote The SPDY Book, which is still the only book on SPDY:



Three months later, I co-authored Recipes with Backbone.js with Nick Gauthier:



Three months later, I wrote the first book on Dart, Dart for Hipsters:



Each of these technologies has two things in common:

1. They are game changing (or at least possibly).
2. I knew nothing about them before I started writing them.

What business did I have writing books on topics about which I knew nothing? Well, let me put it this way: I did it, so why shouldn't I (or anyone else)?

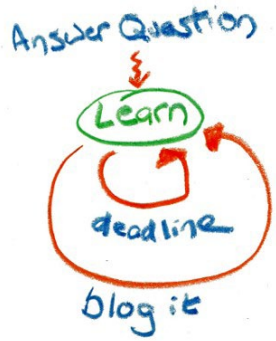
How did I do it?

I blogged every single day. For one full year. 366 days. Every day. No matter what.

I honestly don't know why I started doing this. One night I had a brilliant idea and before I stopped and thought about how stupid it was, I publicly committed myself to doing it.

And it worked. Every night, I ask a question to which I don't know the answer, and I try my damnedest to answer it.

Every time I do this, I learn. The daily deadline forces me to learn. Blogging about it challenges my assumptions and makes me learn even more.



And then, doing it again the next day reinforces the learning. As does writing the book. And the second edition.

I am proud that I didn't let this get in the way of what's important. I still took vacations with the family — drove to the beach and Disney World. Birthdays, anniversary, sickness — I was there for it all.

And in the end, what did I learn? Well aside from a ton about coos technologies, I learned that:

I tricked myself into being awesome.

I heard a story on RadioLab about a lady named Zelda. She tricked herself into quitting smoking by swearing that she would donate $5,000 to the KKK if she ever smoked another cigarette. And she never did. Would she have really donated that money if she had given in? Maybe not, but it was enough for her to have convinced herself that it would happen.

And, in the end, I did the same. Would the world have ended if I missed a day? Of course not. Very few, if any people would have noticed. But I would have noticed because I committed to doing this. And, after 366 days, I have more than not smoking to show for it. I have three books, the last of which is being published by The Pragmatic Programmers. ■

Chris is an author and web developer at EEE Computes LLC with more than 10 years professional experience in a variety of domains. Despite this extensive background, you could fill a book with what he does not know, which is rather the point.

Reprinted with permission of the original author.
First appeared in *hn.my/tricked* (japhr.blogspot.com)

The Psychology of Tackling Hard Problems

BY DAVID VALDMAN

THE THING ABOUT hard problems is that there are many difficulties and few solutions. Sounds obvious, but what's often overlooked is the psychological component to this asymmetry. There's a simple reason why tackling a hard problem can lead to depressive symptoms: you're necessarily wrong 99% of the time.

I'm getting my PhD in math, and developing a web app/startup on the side. I can tell you one thing from my PhD research that I can carry over to my entrepreneurial ambitions: you only have to be right 1% of the time. The hard part is, you need to be psychologically prepared to be wrong all other times.

I haven't seen much discussion of this idea, but I've faced it repeatedly myself, and I often see it in others. I've seen it so often I'm convinced of its pervasiveness. Here's an example. One of my peers tells me his numerics code isn't working:

Me: Have you tried this test case?
Him: No, actually.
Me: Well that may isolate the bug.
Him: But I'm afraid that it won't work.

Sound silly and contrived? It isn't, and I have complete sympathy for this situation. So many times in my work I've fantasized about the solution to an idea, and have been too afraid to implement it because of the subliminal fear that I will be, yet again, wrong. It's a Pavlovian response to the track record of being repeatedly disappointed. Meanwhile, I delight in having new ideas, and enjoy brainstorming them. But without implementing them, the process is worthless.

The point is to be aware. If you find yourself resisting an obvious step due to an irrational fear, step back and force yourself to push onward. You only need to be right 1% of the time. ■

David Valdman is finishing his PhD in applied math at UC Santa Barbara this summer. Soon to be "not that kind of doctor". He's also the founder of Quip Video, a web app for annotating online video. Follow David on twitter at @dmvaldman

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

What Good is Experience?

By JUSTIN KAN

When I didn't have any experience, I thought that experience was totally worthless. Emmett and I taught ourselves how to build web applications in a few months in college and built the first version of Kiko pretty quickly. I did the front end by piecing together JavaScript tutorials until we had something that resembled a calendar.

We thought we were pretty awesome. If we could build a web app that easily and drum up a bunch of public interest, then it seemed to us that everyone should be starting startups right out of college, and that anyone who wasn't was just too scared. What was the point of waiting? You aren't getting any younger.

When I think about that first codebase today I want to vomit in my own mouth. I am glad that I no longer have access because I want to deny it ever existed. It was a mess of spaghetti code, and even though we built it quickly, it took a lot longer than it should have.

Ironically, now that I have experience, I think experience is priceless. What's made me change my mind?

- **Experience makes you move more quickly.** It turns out I'm still not a wonderful programmer. I am, however, a pretty decent web developer, and this is entirely due to experience. Need a Rails CRUD app with an API? Boom, I've been doing that for seven years now. I built the entire backend, frontend, and API for Exec myself in three weeks in January.
- **Experience helps you focus on the right things.** When you don't know what's important, it is easy to think every decision is important. Most of them aren't. Having experience helps you know what decisions you can ignore, postpone, or delegate (almost all of them), and what things you actually need to do right now.
- **Experience gives you confidence.** We've raised venture money for our companies before; I know I can do it again. I've built web apps before; I know I don't need to hire a programmer to replace myself unless we find someone who is really excellent. In the meantime, I can wait. When you've done something before, you aren't worried you can't do it again.

I still think there are some potential downsides to having experience that are worth watching out for:

- **Experience tends to pre-empt innovation.** It's been said before, but when you have a lot of experience in a certain area, you generally think of solutions and approaches that have worked for you in the past. Sometimes this prevents you from taking a fresh approach which ultimately would work out better.
- **Experience takes time to get.** Waiting for experience is also an excuse not to get started. By the time you feel comfortable and confident enough to jump off, the moment might have passed.
- **You know some things to be impossible.** Most things that were impossible or impractical years ago became possible or will become possible some time later. Your experience might tell you that something you want to do can't be done. Other people will go on to do them.

And lastly, something I've been wondering: is it possible to fake experience by getting advice? Perhaps for highly specialized topics, like how to scale your exploding website. However, I think that there are a great many things that people are destined to learn themselves the hard way. So, don't worry too much about trying to find a hack to get experience, when you get enough experience you'll be experienced enough to know one doesn't exist. ■

Justin Kan is the founder and CEO of Exec, your on demand work force. Previously he founded Justin.tv, TwitchTV and Socialcam. He is a part time partner at Y Combinator.

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

How to Build a Naive Bayes Classifier

By ALEXANDRU NEDELCU

Some use-cases for building a classifier:

- Spam detection; for example, you could build your own Akismet API.
- Automatic assignment of categories to a set of items.
- Automatic detection of the primary language (e.g. Google Translate).
- Sentiment analysis, which in simple terms refers to discovering if an opinion is about love or hate for a certain topic.

In general, you can do a lot better with more specialized techniques, however the Naive Bayes classifier is general-purpose, simple to implement, and good-enough for most applications. And while other algorithms give better accuracy, I discovered that having better data in combination with an algorithm that you can tweak gives better results for less effort.

In this article I'm describing the math behind it. Don't fear the math, as this is simple enough that a high-schooler could understand. And even though there are a lot of libraries out there that already do this, you're far better off understanding the concept behind it. Otherwise, you won't be able to tweak the implementation in response to your needs.

0. The Source Code

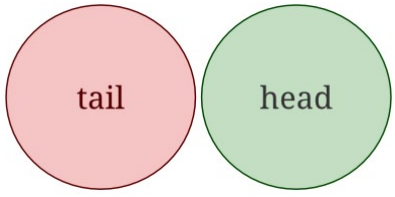
I published the source-code associated at *github.com/alexandru/stuff-classifier*. The implementation itself is at `lib/bayes.rb`, with the corresponding `test/test_002_naive_bayes.rb`.

1. Introduction to Probabilities

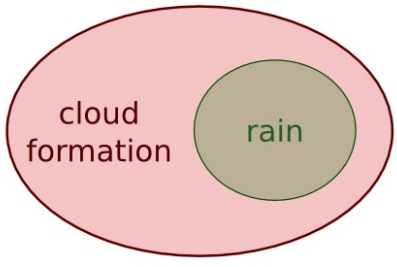
Let's start by refreshing forgotten knowledge. Again, this is very basic stuff, but if you can't follow the theory here, you can always go to the probabilities section on Khan Academy [hn.my/proba].

1.1. Events and Event Types

An "event" is a set of outcomes (a subset of all possible outcomes) with a probability attached. So when flipping a coin, we can have one of these two events: tail or head. Each of them has a probability of 50%. Using a Venn diagram, this would look as follows:



The example below clearly shows the dependence between "rain" and "cloud formation" since rain can only happen if there are clouds:



The relationship between events is very important, as you'll see next:

- 2 events are **disjoint (exclusive)** if they can't happen at the same time (a single coin flip cannot yield a tail and a head at the same time). For Bayes classification, we are not concerned with disjoint events.

- 2 events are **independent** when they can happen at the same time, but the occurrence of one event does not make the occurrence of another more or less probable. For example, the second coin-flip you make is not affected by the outcome of the first coin-flip.
- 2 events are **dependent** if the outcome of one affects the other. In the example above, clearly it cannot rain without a cloud formation. Also, in a horse race, some horses have better performance on rainy days.

What we are concerned with here is the difference between dependent and independent events because calculating the intersection (both happening at the same time) depends on it. So, for independent events, calculating the intersection is easy:

$$P(A \cap B) = P(A) * P(B)$$

Some examples:

- If you have 2 hard-drives, each of them having a 0.3 (30%) probability of failure within the next year, that means there's a 0.09 (9%) probability of them failing both within the next year.
- If you flip a coin 4 times, there's a 0.0625 probability of getting a tail 4 times in a row (0.5 ^ 4).

Things are not so simple for dependent events, which is where the Bayes Theorem comes into play.

1.2. Conditional Probabilities and the Bayes Theorem

Let's take one example with the following stats:

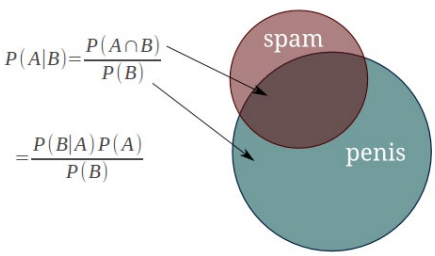
- 30 emails out of a total of 74 are spam messages.
- 51 emails out of those 74 contain the word "penis."
- 20 emails containing the word "penis" have been marked as spam.

So the question is: what is the probability that the latest received email is a spam message, given that it contains the word "penis"?

These 2 events are clearly dependent, which is why you must use the simple form of the Bayes Theorem:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

$$= \frac{P(B|A)\,P(A)}{P(B)}$$



With the solution being:

$$P(spam|penis) = \frac{P(penis|spam) * P(spam)}{P(penis)}$$

$$= \frac{\frac{20}{30} * \frac{30}{74}}{\frac{51}{74}} = \frac{20}{51} = 0.39$$

The above example is simple so you can see the result without complicating yourself with the Bayes formula.

1.3. The Naive Bayes Approach

Let us complicate the problem above by adding to it:

- 25 emails out of the total contain the word "viagra."
- 24 emails out of those have been marked as spam.

What's the probability that an email is spam, given that it contains both "viagra" and "penis"?

Shit just got more complicated, because now the formula is this one:

$$\frac{P(penis|spam \cap viagra) * P(viagra|spam) * P(spam)}{P(penis|viagra) * P(viagra)}$$

And you definitely don't want to bother with it if we keep adding words. But what if we simplified our assumptions and just say that the occurrence of penis is totally independent from the occurrence of viagra? Then the formula just got much simpler:

$$P(spam|penis, viagra)$$

$$= \frac{P(penis|spam) * P(viagra|spam) * P(spam)}{P(penis) * P(viagra)}$$

$$= \frac{\frac{24}{30} * \frac{20}{30} * \frac{30}{74}}{\frac{25}{74} * \frac{51}{74}} = 0.928$$

To classify an email as spam, you'll have to calculate the conditional probability by taking hints from the words contained. And the Naive Bayes approach is exactly what I described above: we make the assumption that the occurrence of one word is totally unrelated to the occurrence of another, to simplify the processing and complexity involved.

This does highlight the flaw of this method of classification, because clearly the 2 events we picked (viagra and penis) are correlated and our assumption is wrong. But this just means our results will be less accurate.

2. Implementation

I'll mention it again: you can take a look at the source-code published at *github.com/alexandru/stuff-classifier*

2.1. General Algorithm

You simply get the probability for a text to belong to each of the categories you test against. The category with the highest probability for the given text wins:

$$classify(word_1, word_2 \ldots word_n) = \underset{cat}{argmax}\, P(cat) * \prod_{i=1}^{n} P(word_i|cat)$$

Do note that above I also eliminated the denominator from our original formula because it is a constant that we do not need (called evidence).

2.2. Avoiding Floating Point Underflow

Because of the underlying limits of floating points, if you're working with big documents (not the case in this example), you do have to make one important optimization to the above formula:

- Instead of the probabilities of each word, you store the (natural) logarithms of those probabilities.
- Instead of multiplying the numbers, you add them instead.

So instead of the above formula, if you need this optimization, then use this one:

$$classify(word_1, word_2 \ldots word_n) = \underset{cat}{argmax} \log_e(P(cat)) + \sum_{i=1}^{n} \log_e(P(word_i|cat))$$

2.3. Training

Your implementation must have a training method. Here's how mine looks like:

```
def train(category, text)
  each_word(text) {|w| increment_word(w, category)}
  increment_cat(category)
end
```

And its usage:

```
classifier.train :spam, "Grow your penis to 20
inches in just 1 week"
classifier.train :ham,  "I'm hungry, no I don't
want your penis"
```

For the full implementation, take a look at `base.rb`

2.4. Getting Rid of Stop Words / Stemming

First of all, you must get rid of the junk. Every language has words that are so commonly used that they become meaningless for any kind of classification you may want to do. For instance, in English, you can safely strip out such words as "the," "to," "you," "he," "only," "if," and "it" from the text.

I've compiled a list of such words in this file: `stop_words.rb`. You can compile such a list by yourself if you're using a language other than English. Head over to Project Gutenberg [gutenberg.org], download some books in the language you want, count the words in them, sort by popularity in descending order, and keep the top words as words that you can safely ignore.

Also, our classifier is really dumb in the sense that it does not care about the meaning or context of a word. So there's a problem: consider the word "running." What you want is to treat this just as "run", which is the morphological root of the word. You also want to treat "parenting" and "parents" as "parent."

This process is called stemming and there are lots of libraries for it. I think currently the most up-to-date and comprehensive library for stemming is Snowball. It's a C library with lots of available bindings, including for Ruby and Python, and it even has support for my native language (Romanian).

Take a look at what I'm doing in `tokenizer.rb`, where I'm getting rid of stop words and stemming the remaining words.

```
each_word('Hello world! How are you?')

# => ["hello", "world"]

each_word('Lots of dogs, lots of cats!
  This is the information highway')

# => ["lot", "dog", "lot", "cat", "inform",
#     "highway"]

each_word("I don't really get what you want to
accomplish. There is a class TestEval2, you
can do test_eval2 = TestEval2.new afterwards.
And: class A ... end always yields nil, so your
output is ok I guess ;-)")

# => ["really", "want", "accomplish", "class",
#     "testeval", "test", "eval", "testeval",
#     "new", "class", "end", "yields", "nil",
#     "output", "ok", "guess"]
```

2.5. Implementation Guidelines

When classifying emails for spam, it is a good idea to be sure that a certain message is a spam message. Otherwise, users may get pissed by too many false positives.

Therefore it is a good idea to have thresholds. This is how my implementation looks:

```
def classify(text, default=nil)
  # Find the category with the highest probability

  max_prob = 0.0
  best = nil

  scores = cat_scores(text)
  scores.each do |score|
    cat, prob = score
    if prob > max_prob
      max_prob = prob
      best = cat
    end
  end

  # Return the default category in case the
  # threshold condition was not met. For
  # example, if the threshold for :spam is 1.2
  #
  #    :spam => 0.73, :ham => 0.40  (OK)
  #    :spam => 0.80, :ham => 0.70  (Fail, :ham
  #    is too close)

  return default unless best
  threshold = @thresholds[best] || 1.0

  scores.each do |score|
    cat, prob = score
    next if cat == best
    return default if prob * threshold > max_prob
  end

  return best
end
```

Final Words

My example involved spam classification, but this is not how modern spam classifiers work. Because the independence assumptions are often inaccurate, this type of classifier can be gamed by spammers to trigger a lot of false positives, which will make the user eventually turn the feature off.

But it is general purpose, being useful not only for spam detection, but also for lots of other use-cases, and it's enough to get you started. ■

Alexandru is an experienced software developer that ventured across anything he found interesting. Besides trying to make people's lives better, he also enjoys cooking and spending time with his toddler. He lives in Romania and works remotely for U.S. based startups.

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

Coding Tricks of Game Developers

By HARVEY GREEN

If you've got any real world programming experience, then no doubt at some point you've had to resort to some quick and dirty fix to get a problem solved or a feature implemented while a deadline loomed large. Game developers often experience a horrific "crunch" (also known as a "death march"), which happens in the last few months of a project leading up to the game's release date. Failing to meet the deadline can often mean the project gets cancelled or even worse, you lose your job. So what sort of tricks do they use while they're under the pump, doing 12+ hour days for weeks on end?

Below are some classic anecdotes and tips (many thanks to Brandon Sheffield who originally put together this article [hn.my/dirty] on Gamasutra). I have included a few of his stories and also added some more from newer sources.

The Programming Antihero –Noel Llopis

I was fresh out of college, still wet behind the ears, and about to enter the beta phase of my first professional game project, a late-90s PC title. It had been an exciting rollercoaster ride, as projects often are. All the content was in and the game was looking good. There was one problem though: we were way over our memory budget.

Since most memory was taken up by models and textures, we worked with the artists to reduce the memory footprint of the game as much as possible. We scaled down images, decimated models, and compressed textures. Sometimes we did this with the support of the artists, and sometimes over their dead bodies.

We cut megabyte after megabyte, and after a few days of frantic activity, we reached a point where we felt there was nothing else we could do. Unless we cut some major content, there was no way we could free up any more memory. Exhausted, we evaluated our current memory usage. We were still 1.5 MB over the memory limit!

At this point one of the most experienced programmers in the team, one who had survived many years of development in the "good old days," decided to take matters into his own hands. He called me into his office, and we set out upon what I imagined would be another exhausting session of freeing up memory.

Instead, he brought up a source file and pointed to this line:

```
static char
buffer[1024*1024*2];
```

"See this?" he said. And then deleted it with a single keystroke. Done!

He probably saw the horror in my eyes, so he explained to me that he had put aside those two megabytes of memory early in the development cycle. He knew from experience that it was always

impossible to cut content down to memory budgets, and that many projects had come close to failing because of it. So now, as a regular practice, he always put aside a nice block of memory to free up when it's really needed.

He walked out of the office and announced he had reduced the memory footprint to within budget constraints. He was toasted as the hero of the project.

As horrified as I was back then about such a "barbaric" practice, I have to admit that I'm warming up to it. I haven't gotten into the frame of mind where I can put it to use yet, but I can see how sometimes, when you're up against the wall, having a bit of memory tucked away for a rainy day can really make a difference. Funny how time and experience changes everything.

Cache It Up –Andrew Russell

To improve performance when you are processing things in a tight loop, you want to make the data for each iteration as small as possible, and as close together as possible in memory. That means the ideal is an array or vector of objects (not pointers) that contain only the data necessary for the calculation.

This way, when the CPU fetches the data for the first iteration of your loop, the next several iterations worth of data will get loaded into the cache with it.

There's not really much you can do with using fewer and faster instructions because the CPU is as fast as it's going to get, and the compiler can't be improved. Cache coherence is where it's at. This article [hn.my/coherence] contains a good example of getting cache coherency for an algorithm that doesn't simply run through data linearly.

Plan Your Distractions –Jay Barnson

The Internet is one of the greatest tools ever invented for both improving and destroying productivity. Twitter and forums and blogs and instructional websites can be extremely motivational and educational, but they can also be a distraction that completely destroys all hope of ever getting anything done. One thing I've done in the past which has proven pretty successful is to stick to a plan for when I can spend some minutes checking email and Twitter, or play a quick game or something. Either at the completion of a task, or after a period of time (say one five-minute break every hour). Otherwise, the browser's only use is for reading reference manual pages, if necessary. That way I turn a potential distraction into a motivating tool.

Collateral damage –Jim Van Verth

Don't know how many remember Force 21, but it was an early 3D RTS which used a follow cam to observe your current platoon. Towards the end of the project we had a strange bug where the camera would stop following the platoon — it would just stay where it was while your platoon moved on and nothing would budge it. The apparent cause was random because we couldn't find a decent repro case. Until, finally, one of the testers noticed that it happened more often when an air strike occurred near your vehicles. Using that info I was able to track it down.

Because the camera was using velocity and acceleration and was collidable, I derived it from our PhysicalObject class, which had those characteristics. It also had another characteristic: PhysicalObjects could take damage. The air strikes did enough damage in a large enough radius that they were quite literally "killing" the camera.

I did fix the bug by ensuring that cameras couldn't take damage, but just to be sure, I boosted their armor and hit points to ridiculous levels. I believe I can safely say we had the toughest camera in any game.

The Blind Leading the Blind –Maurício Gomes

At university, there was a team that made a FPS flash game. For some bizarre reason, the programmer, instead of checking if the character was colliding with the wall to keep you from going there, he did the inverse: he checked if there was a wall, and only allowed you to move parallel to it!

This sparked a bizarre bug: in crossings or T junctions in the level, you could not actually cross, only turn to the passage on your left or right. The deadline was closing, and they had no idea on how to fix it.

Then the team writer fixed the issue; he told the artist to add an animation of hands touching the walls, and then he added in the background story that the main character was blind and needed to constantly touch the walls to know where he was going.

You Wouldn't Like Me When I'm Angry –Nick Waanders

I once worked at THQ studio Relic Entertainment on The Outfit, which some may remember as one of the earlier games for the Xbox 360. We started with a PC engine (single-threaded), and we had to convert it to a complete game on a next-gen multi-core console in

about 18 months. About 3 months before shipping, we were still running at about 5 FPS on the 360. Obviously this game needed some severe optimization.

When I did some performance measurements, it became clear that as much as the code was slow and very "PC," there were also lots of problems on the content side as well. Some models were too detailed, some shaders were too expensive, and some missions simply had too many guys running around.

It's hard to convince a team of 100 people that the programmers can't simply "fix" the performance of the engine, and that some of the ways people had gotten used to working needed to change. People needed to understand that the performance of the game was everybody's problem, and I figured the best way to do this is with a bit of humor that had a bit of hidden truth behind it.

The solution took maybe an hour. A fellow programmer took 4 pictures of my face: one really happy, one normal, one a bit angry, and one where I am pulling my hair out. I put this image in the corner of the screen, and it was linked to the frame rate. If the game ran at over 30fps, I was really happy, if it ran below 20, I was angry.

After this change, the whole FPS issue transformed from, "Ah, the programmers will fix it." to, "Hmm, if I put this model in, Nick is going to be angry! I'd better optimize this a little first." People could instantly see if a change they made had an impact on the frame rate, and we ended up shipping the game at 30fps.

It's Not a Bug,It's a Feature! –Philip Tan

I worked on an RPG in which we were trying to get the NPCs (Non-player Characters) to spot when you were in range, walk up to you, and strike up a conversation with you by activating the dialog system.

We forgot to add code to distinguish NPCs from PCs (Player Characters), so we'd walk into town and all the NPCs would be talking with each other. Because all NPC AI code used the same dialog template, they actually got a few sentences in before the conversations became nonsensical. And because character dialog was broadcast, you could read everything they said if you were in range.

We decided to turn that bug into a major feature.

Dirty Deeds –Tim Randall

The engine team at Gremlin Interactive used to keep a single glove in their office. When someone asked why it was there, they were told it was only used when someone was about to type some really dirty code. It wasn't so much a case of not wanting to leave fingerprints but rather not wanting to actually touch the dirtiest fixes!

Explicit Conditional Hinting –ZorbaTHut

A very, very low-level tip, but one that can come in handy: most compilers support some form of explicit conditional hinting. GCC has a function called `__builtin_expect` which lets you inform the compiler what the value of a result probably is. GCC can use that data to optimize conditionals to perform as quickly as possible in the expected case, with slightly slower execution in the unexpected case.

```
if(__builtin_expect(entity->extremely_unlikely_flag, 0)) {
  // code that is rarely run
}
```

I've seen a 10-20% speedup with proper use of this.

Objective Oriented Programming –Anonymous

Back at a game studio, I think it was near the end of the project, we had an object in one of the levels that needed to be hidden. We didn't want to re-export the level and we did not use checksum names. So right smack in the middle of the engine code we had something like the following:

```
if(level==10 && object==56 )
{
    HideObject();
}
```

The game shipped with this in.

Maybe a year later, an artist using our engine came to us very frustrated about why an object in their level was not showing up after exporting. The level they had a problem with resolved to level 10. I wonder why?

Stack vs. Heap –Torbjörn Gyllebring

Stack allocation is much faster than heap allocation since all it really does is move the stack pointer. Using memory pools, you can get comparable performance out of heap allocation, but that comes with a slight added complexity and its own headaches.

Also, stack vs. heap is not only a performance consideration; it also tells you a lot about the expected lifetime of objects. The stack is always hot, and the memory you get is much more likely to be in

cache than any far heap allocated memory.

The downside of the stack is that it is actually a stack. You can't free a chunk of memory used by the stack unless it is on top of it. There's no management — you push or pop things on it. On the other hand, the heap memory is managed: it asks the kernel for memory chunks, maybe splits them, merges them, reuses them, and frees them. The stack is really meant for fast and short allocations.

I'm a Programmer, Not an Artist –Damian Connolly

For indie/solo developers who are working on an iPhone or Android game on their own, while you're looking for an artist, you should be developing your game at the same time. Use programmer art, stand-ins, free sprites — anything. Most of the time, before even thinking about final assets, I just want something up and running quickly to see if it's fun. Prototype the crap out of it and find the game. Then, when the gameplay's locked down, you can start putting in the proper art. Doing it the other way around leads to lost money, and work that needs to be redone multiple times, which aside from harming your project, sucks your motivation to finish it (and if you're making a game to get a job, showing that you can finish a project is a good thing). Another tip if you're lacking upfront finance is to find a freelance game artist who will accept a revenue sharing deal, e.g. typically something like 30% of game revenue, payable once it gets published to the AppStore.

Remove Unnecessary Branches –tenpn

On some platforms and with some compilers, branches can throw away your whole pipeline, so even insignificant `if()` blocks can be expensive.

The PowerPC architecture (PS3/x360) offers the floating-point select instruction, fsel. This can be used in the place of a branch if the blocks are simple assignments:

```
float result = 0;
if(foo > bar){ result = 2.0f; }
else { result = 1.0f; }
```

Becomes:

```
float result = fsel(foo-bar,
2.0f, 1.0f);
```

When the first parameter is greater than or equal to 0, the second parameter is returned, else the third. The price of losing the branch is that both the `if{}` and the `else{}` block will be executed, so if one is an expensive operation or dereferences a NULL pointer, this optimization is not suitable. Sometimes your compiler has already done this work, so check your assembly first.

Hack the Stack –Steve DeFrisco

I was one of a few interns at IMAGIC in 1982-83. We were all doing Intellivision carts. One of the programmers had to leave to go back to school, and I was chosen to fix the random crash bug in his game. It turned out to be a stack overflow in the timer interrupt handler. Since the only reason for the handler was to update the *display* of the on-screen timer, I added some code to test the depth of the stack at the beginning of the interrupt routine. If we were in danger of overflowing the stack, return without doing anything. Since the handler was called multiple times per second, the player never noticed, and the crash was fixed.

Meet My Dog, "Patches" –Mick West

There's an old joke that goes something like this:

> *Patient: "Doctor, it hurts when I do this."*
>
> *Doctor: "Then stop doing it."*

Funny, but are these also wise words when applied to fixing bugs? Consider the load of pain I found myself in when working on the port of a 3D third person shooter from the PC to the original PlayStation.

Now, the PS1 has no support for floating point numbers, so we were doing the conversion by basically recompiling the PC code and overloading all floats with fixed point. That actually worked fairly well, but where it fell apart was during collision detection.

The level geometry that was supplied to us worked reasonably well in the PC version of the game, but when converted to fixed point, all kinds of seams, T-Junctions, and other problems were nudged into existence by the microscopic differences in values between fixed and floats. This problem would manifest itself in one case with the main character touching a particular type of door in a particular level in a particular location; rather than fix the root cause of the problem, I simply made it so that if he ever touched the door, then I'd move him away, and pretend it never happened. Problem solved.

Looking back I find this code quite horrifying. It was patching bugs and not fixing them.

Unfortunately the real fix would have been to go and rework the entire game's geometry and collision system specifically with the PS1 fixed point limitations in mind. The schedule was initially aggressive, and since we always seemed close to finishing, the quick patch option won over against a comprehensive (but expensive) fix.

But it did not go well. Hundreds of patches were needed, and then the patches themselves started causing problems, so more patches were added to turn off the patches in hyper-specific circumstances. The bugs kept coming, and I kept beating them back with patches. Eventually I won, but at a cost of shipping several months behind schedule, and working 14 hour days for all of those several months.

That experience soured me against "the patch." Now I always try to dig right down to the root cause of a bug, even if a simple, and seemingly safe, patch is available. I want my code to be healthy. If you go to the doctor and tell him "it hurts when I do this," then you expect him to find out why it hurts, and to fix that. Your pain and your code's bugs might be symptoms of something far more serious. The moral: treat your code like you would want a doctor to treat you; fix the cause, not the symptoms.

Identity Crisis –Noel Llopis

This scene is familiar to all game developers: It's the day we're sending out the gold candidate for our Xbox 1 game. The whole team is playtesting the game all day long, making sure everything looks good. It's fun, it's solid, it's definitely a go in our minds.

In the afternoon, we make the last build with the last few game-balancing tweaks, and do one last playthrough session when disaster strikes: the game crashes hard! We all run to our workstations, fire up the debugger, and try to figure out what's going on. It's not something trivial, like an assert, or even something moderately hard to track down, like a divide by zero. It looks like memory is garbage in a few places, but the memory reporting comes out clean. What's going on?

One dinner and many hours later, our dreams of getting out on time shattered, we manage to track it down to one data file being loaded in with the wrong data. The wrong data? How's that possible? Our resource system boiled down every asset to a 64-bit identifier made out of the CRC32 of the full filename and the CRC32 of all the data contents. That was also our way of collapsing identical resource files into a single one in the game. With tens of thousands of files, and two years of development, we never had a conflict. Never.

Until now, that is.

It turns out that one of the innocent tweaks the designers had checked in that afternoon made it so a text file had the exact same filename and data CRC as another resource file, even though they were completely different!

Our hearts sank to our feet when we recognized the problem. There's no way we could change the resource indexing system in such a short period of time. Even if we pulled an all-nighter, there was no way to know for sure that everything would be stable in the morning.

Then, as quickly as despair swept over us, we realized how we could fix this on time for the gold candidate release. We opened up the text file responsible for the conflict, added a space at the end, and saved it. We looked at each other with huge grins on our faces and said:

"Ship it!"

The extra space meant the CRC32 checksum of the text file was altered and therefore no longer conflicted with the other resource.

HexEdit to the Rescue –Ken Demarest

Back on Wing Commander 1 we were getting an exception from our EMM386 memory manager when we exited the game. We'd clear the screen and a single line would print out, something like "EMM386 Memory manager error. Blah blah blah." We had to ship ASAP. So I hex edited the error in the memory manager itself to read "Thank you for playing Wing Commander."

8-bit Audio Stomper –Toonse

For a launch product of a certain console I had a nasty bug report from QA that took 20+ hours to reproduce. Finally (with 24 hours left to go to hit console launch) tracked it down to some audio drivers in the firmware that were erroneously writing 1 random byte "somewhere" at random times where the "somewhere" was always in executable code space. I finally figured out that any given run of the game that "somewhere" was always the same place, luckily. 1st party said sorry, can't fix it in time as we don't know why it's being caused! So I shipped that game with stub code at the very start of main that immediately saved off the 1 byte from the freshly loaded

executable in the place I knew it would overwrite for that particular version of the exe. There was then code that would run each frame after audio had run and restore that byte back to what it should be just in case it had been stomped that frame. Good times! We hit launch.

To this day I still feel very, very dirty about this hack, but it was needed to achieve the objectives and harmed no one.

Rainy Day Server Pool
–Potatolicious

I used to work for a company that had a horrific hardware requisition policy. If your team needed a server, it had to go through a lengthy and annoying approvals process — and even then, it took months before Infrastructure would actually provide said servers.

In other words, when a project gets handed down from above to launch in, say, 3 months, there's no way in hell you can get the servers requisitioned, approved, and installed in that time. It became standard practice for each team to slightly over-request server capacity with each project and throwing the excess hosts into a rainy day pool, immediately available and repurposeable as required.

New servers will still get requested for these projects, but since they took so long to approve, odds are they'd go right into the pool whenever they actually arrived, which sometimes took up to a year.

Of course, it was horrifyingly inefficient. Just on my team alone I think we had easily 50 boxes sitting around doing nothing (and powered on to boot) waiting to pick up the slack of a horrendously broken bureaucracy.

Bit Shifting Magic
–Steven Pigeon

In order to avoid stalls in the processor pipeline due to branching, one can often use a branchless equivalent, that is, code transformed to remove the if-then-else's and therefore jump prediction uncertainties. For example, a straightforward implementation of `abs( )` in C might be:

```
inline int abs(int x)
{
    return (x<0) ? -x : x;
}
```

Which is simple enough but contains an inline if-then-else. As the argument, x, isn't all that likely to follow a pattern that the branch prediction unit can detect, this simple function becomes potentially costly as the jump will be mispredicted quite often.

How can we remove the if-then-else, then? One solution is to use the right shift operator (>>) and the bitwise XOR operator (^) as following:

```
inline int abs_no_branch(int x)
{
  int m = (x >> (8 *
sizeof(int)-1));
  return ((x ^ m) - m);
}
```

Where the expression `(8 * sizeof(int) - 1)` evaluates to 15, 31, or 63 depending on the size of integers on the target computer. ■

Harvey Green has spent the past few years developing in .NET and C# for the Oil & Gas and related industries. He believes that core language skills plus good domain knowledge has been the key to most of the projects he's worked on.

Reprinted with permission of the original author. First appeared in *hn.my/game* (dodgycoder.net)

Python Deployment Anti-Patterns

By HYNEK SCHLAWACK

Deploying web applications is hard. No shiny continuous deployment talk and no DevOps coolness can change that. Or to use DevOp Borat's words: "Is all fun and game until you are need of put it in production." There are some mistakes I see people making again and again, so I'd like to address them here.

My background

Before I start preaching, let me tell you a bit about me and what I do in order to give you some perspective from which I'm writing.

I work for a German web hoster and domain registrar. And I'm deploying Python-based applications all the time. Most parts of our infrastructure are built using Python. And those that aren't, will be eventually.

The sizes range from tiny glue to mission-critical APIs. We have legacy Pylons [pylonsproject.org], new Pyramid, some Django, & a lot of Twisted apps [twistedmatrix.com]. And everything is seasoned with a hint of Celery [celeryproject.org].

So if I say "application," I don't mean just some Django CRUD front end. Python lives in all layers here. And all layers have to be deployed somehow.

Deploying so many diverse applications requires solid and consistent deployment standards if you don't want to go crazy. The main mantra is to go for simple solutions, not for easy ones. Something that is easy now, can become a major PITA down the road.

Don't use ancient system Python versions

Every time someone whines about lack of support for Python 2.4 in recent packages, I hear Kenneth Reitz saying:

> *Python 2.4 is not supported. It came out 8 years ago. That's older than YouTube. Upgrade.*

If you're serious about using Python you should be prepared to roll your own RPMs/DEBs. We're even running RHEL 4 on some of our servers; but we're a Python company, so we use the best thing we can get — even if it means extra work.

We also have to compile our own Apaches and MySQLs for our customer servers (we don't use any of them for our own systems, but our customers demand a solid LAMP-stack) because we need that fine-grained control. Why should Python be an exception? Rolling an own DEB/RPM is a lot less of a nuisance than writing code for Python < 2.6.

This works both ways. It's entirely possible that you have some mission-critical web app that isn't compatible with Python newer than 2.4. Are you going to install a single server with an ancient OS, just to accommodate? Key infrastructure must not be dictated by third parties.

On the other hand I'm not saying that you have to compile Python yourself! Oneiric and later have Python 2.7 on board — there's absolutely no reason to build it for yourself. The stress is on "ancient," not on "system" in this caption.

Use virtual environments

Gentlepeople, if you're deploying software, always use virtualenv. Actually, the same goes for local development: look into virtualenvwrapper which makes handling them a breeze. So never install into your global site packages! The only exception is the aforementioned virtualenv, which in turn installs pip in each environment it installs to.

Test your software against certain versions of packages, pinpoint them using `pip freeze` and be confident that the identical Python environment is just a `pip install -r requirements.txt` away. For the record, I split up my requirement files; more on that in the next installment.

Also, use real version pinning like `package==1.3`. Don't do `package>=1.3`, it will bite you eventually, just as it has bitten me and many others.

Never use Python packages from your distribution

This one is in fact an extreme version of the previous anti-pattern.

First of all, there's no reason to succumb to a dictate of your distribution which version of a package to use. They don't know your application. Maybe you need the latest version, maybe you need a slightly older one.

1. If I write and test software, I do it against certain packages. Packages tend to change APIs, introduce bugs, etc.
2. My software is supposed to run on any UNIXy platform as long as the Python it's written against is present.

What if the next Ubuntu ships with a different SQLAlchemyby default? Do I have to fix all my applications before upgrading our servers? Or what if I need to deploy an app to an older server? Do I have to rewrite it so it runs with older packages? I prefer not to.

I really wish the Linux distributions wouldn't ship anything more than the Python interpreter and virtualenv. Anything else just leverages bad behavior.

The only good they may be doing is automatically updating packages with security vulnerabilities that you may have missed. That said, I'm convinced that if you deploy software to the net, you have the responsibility to monitor them yourself anyway. Relying on your distribution gives you just a false sense of security; if your customer's data gets hacked, they don't care that Ubuntu was to slow to issue a security update.

Don't run your daemons in a tmux/screen

It seems to be part of everyone's evolution to do it, so be the first one to skip it!

Yes, tmux is full of awesome (and way better than screen), but please don't just ssh on your host and start the service in a tmux or screen. You have nothing that brings the daemon back up if it crashes. You can't restart it on 10 servers without ssh'ing on 10 servers, get the screen and Ctrl-C it. Granted, it's easy in the beginning, but it doesn't scale and lacks basic features that simple-to-use tools have to offer.

My favorite one is supervisord [supervisord.org]. A definition for a service looks as simple as:

```
[program:yourapp]
command=/path/to/venv/bin/gunicorn_django --config deploy/gunicorn-config.py settings/production.py
user=yourapp
directory=/apps/yourapp
```

You add the file to `/etc/supervisor/conf.d/`, make a `supervisorctl update` and your service is up and running. It's a no-brainer and much easier than juggling rc.d scripts. Crash recovery and optional web interface included.

Configuration is not part of the application

Your production configuration doesn't belong in the (same) source repository. There are configuration management tools like Puppet [puppetlabs.com] or Chef [opscode.com/chef] that do exactly that for you — just better and more reliably. While installing the configuration, Puppet can make sure that the directories always have certain permissions. Configuration templates make it perfect for mass deployments. Some service IP changed? Just fix it in Puppet's repo and deploy the changes. Eventually all services will catch up. If you want, you can always trigger a run, for example using a simple Fabric [fabfile.org] script.

But don't use Fabric for actual deployments! This is the perfect example of the battle between "simple" and "easy." At first, it's easier to put everything inside of the repo and run a Fabric script that does a `git pull` and restarts your daemon. In the long run, you'll regret it like many before you did.

Just to stress this point: I love Fabric and couldn't live without it. But it's not the right tool for orchestrating deployments — that's where Puppet and Chef step in.

Look into alternatives to Apache + mod_wsgi setups

Many people go for Apache and mod_wsgi by default, because everybody has already heard about Apache.

To me, Apache feels like a big ball of mud, and I find the modular combination of gunicorn [gunicorn.org] or uwsgi [hn.my/uwsgi] together with nginx much more pleasing and easier to control.

Enough negativity

I don't claim that I've discovered the sorcerer's stone. However, I've developed a system for us that proved solid and simple in the long run.

The trick is to build a debian package (but it can be done using RPMs just as well) with the application and the whole virtualenv inside. The configuration goes into Puppet, and Puppet also takes care that the respective servers always have the latest version of the DEB.

The advantage is that such a DEB is totally self-contained, doesn't require having to build tools and libraries on the target servers, and, paired with solid Puppet configuration, it makes consistent deployments over a wide range of hosts easy, fast, and reliable. But you have to do your homework first. ■

Hynek is a wine-loving software engineer from Berlin/Germany, creating robust systems for a living at Variomedia and hacking FOSS for fame at home. He occasionally blogs at *hynek.me* and regularly tweets as @hynek

Reprinted with permission of the original author. First appeared in *hn.my/pydev* (hynek.me)

This is Why You Spent All that Time Learning to Program

By JAMES HAGUE

THERE'S A STANDARD format for local TV news broadcasts that's easy to criticize.

There's an initial shock-value teaser to keep you watching. News stories are read in a dramatic, sensationalist fashion by attractive people who fill most of the screen. There's an inset image over the shoulder of the reader. Periodically there's a cutaway to a reporter in the field; it's often followed-up with side-by-side images of the newscaster and reporter while the former asks a few token questions to latter. There's pretend banter between newscasters after a feel-good story.

You get the idea. Now what if I wanted to change this entrenched structure?

I could get a degree in journalism and try to get a job at the local TV station. I'd be the new guy with no experience, so it's not likely I could just step-in and make sweeping reforms. All the other people there have been doing this for years or decades, and they've got established routines. I can't make dozens of people change their schedules and habits because I think I'm so smart. To be perfectly fair, a drastic reworking of the news would result in people who had no issues with old presentation getting annoyed and switching to one of the other channels that does things the old way.

When I sit down to work on a personal project at home, it's much simpler.

I don't have to follow the familiar standards of whatever kind of app I'm building. I don't have to use an existing application as a model. I can disregard history. I can develop solutions without people saying "That's not how it's supposed to work!"

That freedom is huge. There are so many issues in the world that people complain about, and there's little chance of fixing the system in a significant way. Even something as simple as reworking the local news is out of reach. But if you're writing an iOS game, an HTML 5 web app, a utility that automates work so you can focus on the creative fun stuff, then you don't have to fall back on the existing, comfortable solutions that developers before you chose simply because they, too, were trapped by the patterns of the solutions that came before them.

You can fix things. You can make new and amazing things. Don't take that ability lightly. ■

James Hague has been Design Director for Red Faction: Guerrilla, editor of "Halcyon Days: Interviews with Classic Computer and Video Game Programmers," co-founder of an indie game studio, and a published photographer. He started his blog "Programming in the 21st Century," in 2007.

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

Faster than C

By ANDREAS ZWINKAU

Judging the performance of programming languages, usually C is called the leader, though Fortran is often faster. New programming languages commonly use C as their reference, and they are really proud to be only so much slower than C. Few language designer try to beat C.

What does it take for a language to be faster than C?

Better Aliasing Information

Aliasing describes the fact that two references might point to the same memory location. For example, consider the canonical memory copy (not `memcpy` from `stdlib.h`!):

```
void* memcopy(void* dst, const
void* src, size_t count) {
   while (count--) *dst++ =
*src++;
   return dst;
}
```

Depending on the target architecture, a compiler might perform a lot of optimizations with this code. For example, on a modern x86 with the SSE instruction MOVDQU, it could copy 16 Byte blocks instead of 4 Byte (`sizeof(void*)`). Unfortunately, no. Due to aliasing, `dst` could for example be `src+1`. In this case, the result must be the first word `*src` repeated `count` times at `dst`. The compiler is not allowed to use `MOVDQU` due to the semantics of C.

In C99 the `restrict` keyword was added, which we could use here to encode that `src` and `dst` are different from all other references. This mechanism helps in some cases, but not in our example.

Fortran semantics say that function arguments never alias and there is an array type, where in C arrays are pointers. This is why Fortran is often faster than C. This is why numerical libraries are still written in Fortran. However, it comes at the cost of pointer arithmetic.

A language which wants to be faster than C should provide semantics where aliasing can be better analyzed by the compiler.

Push Computation to Compile-Time

Doing things at compile time reduces the run time. Of course, C compilers do this for trivial cases like `1+2`, where the addition is already handled at compile time.

However, languages with nice meta-programming support enable the programmer to do similar application specific optimizations. A simple example, we could optimize `fib(20)` to `6765`, without the compiler knowing about Fibonacci numbers.

For a real example, the Eigen C++ library for linear algebra uses C++ templates to avoid copies and be lazy about computations. Of course, Lisp is the grandfather of this technique with its macro system. For example, there is a nice anecdote [hn.my/jsobel] about a student using Scheme for an assignment. Basically, the programmer can modify the abstract syntax tree during compilation. The trade-off with such meta programming features is complexity. Programmers underestimate the difficulty to write correct macros like they underestimate the difficulty to write correct concurrent programs.

A language designer should think about meta programming. Something Turing-complete like C++ templates, seems to be beneficial for performance.

Runtime Optimization

At runtime there is dynamic information which is not available to a static compiler. Any specific example could be duplicated by a C program, but usually it is not feasible. The trick of profile-guided optimizations solves only a small part of the problem.

What becomes especially easy at runtime is whole-world optimization. While this is possible statically, the C semantics (compilation units) and the mandatory preprocessor make it difficult for the compiler. Even Python can beat C by inlining across file borders.

Of course, there are downsides to using a JIT and especially in systems and embedded programming it is not appropriate. So there might be examples where Java, C#, or others beat C, but they do not threaten C's niche.

Conclusion

Aliasing information is the only one where I am certain about speed improvements, because it is impossible to reach Fortran-speed in C. The other ideas are more about making it easier to write faster programs. ■

Andreas Zwinkau is a doctoral researcher at the IPD Snelting since 2010. He is working on the libFirm compiler within the InvasIC project. However, this is only true, while he is not occupied with managing and teaching students at the KIT, Germany's finest university for computer science.

Reprinted with permission of the original author.
First appeared in *hn.my/fasterc* (beza1e1.tuxen.de)

"That's Why You Don't Have Any Friends."

By JOE PEACOCK

Yesterday, I was at the gym. I was working out, as I am usually doing while I'm at the gym. And as it happens over the years spent going to the same gym, relationships form and people get to know each other, and groups form and jokes are shared and camaraderie takes place. And it was the same this day.

I was talking with a group of folks who are regularly in during the afternoons on Saturdays. Among them was a 14-year-old boy named Bradley (not his real name). He's a great kid. He's been coming to the gym with his parents for the past two or so years. While his parents walk around the track upstairs, he spends his time learning how to lift weights with us big guys. When he first started, he was wiry and awkward. He's still pretty awkward; being a teenager and all. But us big guys set him on a good path to maintain a healthy level of fitness.

We were cutting up and laughing. The guys made fun of me for liking hockey. "That's a Canadian sport, isn't it?" one asked. "What are you, part Canadian?"

"Only the part that likes real sports," I replied. "And maple syrup."

"I still don't get why you don't like college football," another asked. "You're in Georgia. SEC is bigger than NFL here."

"What can I say?" I asked. "Southerners like their little league sports. I prefer watching pros."

And so it goes, about the same way every Saturday. The topics change — what cars are best, what sports are better than other sports, what teams are better than other teams, what shows are better than other shows (but never politics or religion — something you learn really fast in a gym is to never bring up the two topics most likely to incite violence in a building filled with metal bars and heavy plates). Someone has a divergent interest, everyone else jumps on it, and laughs are had. And invariably, the topic turns to girls.

Husbands laugh about the young singles and their stories about weekend endeavors. Singles laugh at the guys stuck at home with their ball and chain. Whispers are shared about which girls in the gym are hot; warnings are issued by the more experienced about the dangers of dating people from your gym or your job (short version: it doesn't matter how hot the guy or girl is, it's stupid. Unless marriage is assured, don't do it.)

One of the guys asked Bradley if he had a girlfriend. If there were dirt on the gym floor, he'd have been kicking it.

"Nah, no girlfriend," he replied.

"Young strapping lad like you? Nonsense," I said, knowing full well that not only did he not have a girlfriend, he'd have absolutely no clue what to do with one if he did because I was him once. But as a grown up looking out for a younger kid, you have to act like it's completely ridiculous that girls don't flock to him. It's the right thing to do.

"I asked a girl out to the spring dance," he said. He then said something that hit me hard. "She called me lame and said, 'That is why you don't have any friends. Because you're weird.'"

The words rang in my head. Those exact words — I remembered hearing them. A lot. He didn't explain why she thought he was weird. He didn't have to. I knew the feeling very, very well.

"Come on now," one of the guys said. "Don't let her get to you."

"No, she's right," he said. "I don't have any friends. Not at school, anyway." His face got really sad. "I really am weird."

I was weird, back before I realized I wasn't. And it resulted in some extremely lonely times in my young life. My entire elementary and junior high school tenure was spent with no friends. In tenth grade, I found my tiny group of four friends.

> **"There are hundreds of thousands of us. All weird. All strange. All over, everywhere."**

I dated the wrong girl (they're all wrong, until you find the right one). The four of us fractured into two groups of two — Mike and I split off from Walter and Rod.

Then one day, Mike got tired of my bullshit and said those words to me. "That's why you don't have any friends," he said at very high volume. He deserved to say it — I'd just told him to go fuck himself when he tried to explain why my girlfriend at the time was screwing someone behind my back. I called him every name in the book, so he bailed and joined up with Walter and Jay while I spent the last few weeks of high school career. Even the furry had more friends than I did.

And now, 17 years later, life is fantastic. I belong to a studio full of amazing people who were all weird, just like me. I get to meet freaks from across the nation who all love anime and comics, just like me. I get to talk to people who read my weird stories about my weird life and relate to it because, just like me, they're weird.

There are thousands — no, hundreds of thousands — of us. All weird. All strange. All over, everywhere.

We all went to school and hated everyone because they didn't understand us. We dealt with the bullying and the isolation and the feeling that we were the weird ones. You want to know what's weird? Spending hundreds of dollars on clothes and shoes and purses that everyone else thinks are cool. Spending hours of your life doing things that everyone else is doing because it's cool. Liking the bands that everyone else likes because you're a loser if you don't.

You want to know what's weird? Hiding who you are just to have the company of people you don't even like. That's weird.

I looked him straight in the eye. My normally grinning mouth turned stern. With as serious a tone as I could muster, I said "Listen to me, okay? What I'm about to say is something I want you to take in and think about and really hold on to."

He nodded. "Okay," he said.

"This isn't just conversation, this is important," I said. "You listening?"

He nodded again. "I'm listening," he replied with a look that convinced me that he was.

I took a deep breath. "Right now, you're in high school in a small suburban town," I started.

He nodded.

"Everyone you know looks the same and acts the same," I explained. "They may dress differently from each other or belong to different crowds, but they're all the same. Hipsters, brainiacs, jocks, so-called "geeks" — they're all so caught up with not being left out that they're changing who they are to fit in with whoever it is that will accept them.

"When you show up and you're not like that, it scares them," I continued. "They don't know what to do with you, because they have no idea what it's like to think for themselves. So they try to make YOU feel like the loser, because there are more of them doing what they're doing than there are of you. In such a small group of small minds, the nail that sticks up gets hammered down.

"To them, you are weird," I said. "But weird is good. No, screw that — weird is great! Being weird to someone just proves that you are being you, which is the most important thing you can ever be. There's nothing wrong with you. There's something wrong with them. They can't understand what it's like to be themselves, much less what it's like to be you."

He smiled a little. "You really think that?" he asked.

I laughed. "Dude, look at me!" I said. "I'm 300 pounds of ex-football player covered in cartoon and comic book tattoos, who builds

websites and tours the world talking to people about his anime cel collection. Trust me, I know all about being weird."

He shrugged and said, "It just sucks, you know?"

"Oh, I know," I said with a smile. "And here's the little bit of bad news — It's gonna suck for a little while longer. But one day, you'll get out of school and go somewhere besides the small town you're in and you're going to discover that there are groups of people just like you — not that they do what you do or act how you act, but that they refused to change who they are to fit in, and that makes them just like you. And when you find them, you're finally going to feel at home.

"It might be college, or it might be visiting another city. Hell, it might even be on the internet. But at some point you're going to find them. And it's going to be great."

He smiled. "That would be awesome," he said.

"It WILL be awesome!" I replied. "But until then, it's going to be lonely and frustrating. You're going to do stupid things thinking it's going to impress them or change their opinion of you, and it won't, and you're going to get sad. Just know that it does end. It ends the day you realize that you never wanted to be them in the first place because they are losers. They lost the battle to be themselves. You're the winner."

I paused for a second, because it had just occurred to me that, at some point during my little motivational speech, his parents had walked up and were waiting a short distance behind him. I presumed it was to give him enough space to let the conversation be his own, but I knew they had heard me because when I looked at them, they both nodded and smiled.

So I put the cap on the whole thing. "And I know your parents are right there, but I'm going to say it anyway: Fuck. Them."

I kept my eyes on him, but could see just behind him that his mom reacted a little to my vulgarity. His dad placed his hand on her shoulder and just let it be.

The guys in the group all nodded and agreed with me and began talking to him about their perspectives on the situation (which, in previous conversations over the years, I knew to be similar to mine). His parents came up to me and thanked me for talking to him.

"He just thinks the world of you guys," his mom said. "He talks about coming here all the time to work out with you."

"He really needed to hear that," his dad said. "We try to tell him that high school is just that way, but you know how it is..."

"No teenager wants to listen to his parents," I said. "Hell, I'm an adult and I still don't."

They both laughed.

"He's a great kid," I said. "He's going to be just fine in a few years."

"Well, thank you," the dad said. "It means a lot."

"Hey," I said with a shrug, "That's what we're here for. We're his friends." ■

When he's not teaching the Internet how to fist-fight, why being weird is awesome or how to self-publish your own books, Joe Peacock tours the world, showing his extensive "Akira" art collection. He has 13 cats and loves you.

Reprinted with permission of the original author.
First appeared in *hn.my/friends* (joethepeacock.blogspot.com)

What I've Learned about Smart People

By TOMMY MACWILLIAM

Going to Harvard means I have the amazing opportunity to be around a lot of smart people. Now, when I say "smart people," I don't mean that guy who always wins trivia night. I mean blazingly intelligent individuals who are regarded as the pre-eminent scholars in their field. It's pretty amazing to pass by Turing Award winners and leading political science scholars grabbing a sandwich.

Before I go anywhere, let me make one thing clear: I am not one of these smart people. This is perhaps the biggest lesson I've learned after 3 years here. There is an absolutely incredible amount of smart people in the world, and I can name a whole bunch of students and professors alike who I know for a fact I will never ever be as smart as, no matter how hard I try. But honestly, that's okay — I don't need to be (and perhaps that's a story for another day). What that does mean, though, is that I would be doing a disservice to the ever-so-generous Financial Aid Office if I didn't learn from them. I don't mean learning in a lecture hall, but I refer to a more personal sense of learning. What is it that separates a "smart" person from me? How do they conduct themselves? What drives them?

I can of course make no authoritative claims here, but I have noticed one overarching theme among smart people: they ask questions. When someone explains something new to me, I usually just nod my head like I know what they're talking about. If I don't understand something, I just Google it later. After all, I don't want this person to think I'm a moron. Smart people are different. If they don't understand something, or even if they think they understand something, they ask questions. I distinctly remember, as an immature and perhaps arrogant freshman, a guest lecturer in one of my classes. After explaining what I thought was a straightforward concept, the guest lecturer asked if anyone had any questions. Looking around the room, every student simply nodded, indicating everything was clear. A question, however, came from a tenured professor who had undoubtedly been exposed to the material before. At the time, I thought nothing of it and perhaps even thought that I was smarter than the professor because I understood a concept he/she didn't. Now, I am confident that this professor did not ask the question just to make the guest lecturer feel better, to start a discussion, or anything else. The intonation of the question and the intensity with which the professor listened to the response definitively suggested that the professor's question was genuine and that the answer was of great importance.

Based on the research and findings of so many of the students and professors here, it's clear that this trend is no accident. Not only do smart people ask questions when they don't understand something, but they also ask questions when the world thinks it understands something. Smart people challenge the very limit of human understanding, and they push the envelope of what's possible farther than many people would argue it's meant to be pushed. Smart people don't take claims at face value, and smart people don't rest until they find an explanation they're comfortable accepting and understanding.

Smart people challenge everything. (You know who taught me that? A smart person.)

Maybe someday, people will call me a smart person. For now, I'm going to keep asking them questions. ■

Tommy is a Computer Science major at Harvard University known for his affinity for JavaScript. With a passion for promoting innovation, Tommy loves teaching CS courses and empowering students to build killer apps. He also loves cupcakes.

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

The Origins of the <Blink> Tag

By LOU MONTULLI

I AM WIDELY CREDITED as the inventor of the <blink> tag. For those of you who are relatively new to the Web, the <blink> tag is an HTML command that causes text to blink, and many, many people find its behavior to be extremely annoying. I won't deny the invention, but there is a bit more to the story than is widely known.

Back in 1994, I was a founding engineer at Netscape, and prior to that I had written the Lynx browser, which predated all of the other popular browsers at that time. Lynx had been and still is a text-only browser and is commonly used in a console window on UNIX machines. At Netscape we were building software that used a graphical user interface and could express vastly more text styles and layouts as well as images and other media. We spent a lot of time thinking about the future of the web and new technologies that would enable new classes of documents, applications, and uses. A few examples were HTML tables, SSL for secure communications, plugins for extensions, and JavaScript to enable dynamic HTML.

Sometime in late summer I took a break with some of the other engineers and went to a local bar on Castro Street in Mountain View. The bar was the St. James Infirmary and it had a 30 ft Wonder Woman statue, among other interesting things. At some point in the evening I mentioned that it was sad that Lynx was not going to be able to display many of the HTML extensions that we were proposing. I also pointed out that the only text style that Lynx could exploit given its environment was blinking text. We had a pretty good laugh at the thought of blinking text and talked about blinking this and that and how absurd the whole thing would be. The evening progressed pretty normally from there, with a fair amount more drinking and me meeting the girl who would later become my first wife.

Saturday morning rolled around and I headed into the office only to find what else but blinking text. It was on the screen blinking in all its glory and in the browser. How could this be, you might ask? It turns out that one of the engineers liked my idea so much that he left the bar sometime past midnight, returned to the office, and implemented the blink tag overnight. He was still there in the morning and quite proud of it.

At the time there were 3 versions of the browser that ran on UNIX, Windows, and Mac operating systems. For a short 12 hours the blinking was constrained only to the UNIX version, but it didn't take long for the blinking to spread to Windows and then Mac. I remember thinking that this would be a pretty harmless Easter egg; that no one would really use it, but I was very wrong. When we released Netscape Navigator 1.0 we did not document the blink functionality in any way, and for a while all was quiet. Then somewhere, somehow the arcane knowledge of blinking leaked into the real world and suddenly everything was blinking. "Look here," "buy this," "check this out" — all blinking. Large advertisements blinking in all their glory. It was a lot like Las Vegas, except it was on my screen, with no way of turning it off.

In the end, much was said — most of it in the form of flaming posts to various discussion boards, and the <blink> tag will probably be remembered as the most hated of all HTML tags. I would like to publicly state that at no time did I actually write code or even seriously advocate the <blink> tag. It is true that I put forth the initial inspiration, but it really was merely a thought experiment. I am not going to name any names of the people who coded the dastardly deed. If they wish to step forward, they will need to do it themselves. In the end, the thing that I am truly sad about is that Lynx never did get to blink. I am also sad to report that the St James Infirmary burned to the ground in 1997. It was a great place to hang out and will be missed.

<blink> on,
:lou ■

Lou Montulli is a programmer who is well known for his work in producing web browsers. He co-authored a text web browser called Lynx and programmed the networking code for the first versions of the Netscape web browser. He is currently working on a new Enterprise class cloud storage service at a company named Zetta.

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

MEMSET® HOSTING

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

MINISERVER™ CLOUD COMPUTE

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

MEMSTORE™ CLOUD STORAGE

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

SCAN THE CODE FOR MORE INFORMATION

MEMSET® HOSTING

CarbonNeutral® hosting

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

PC PRO Excellence AWARDS 2011 WINNER BEST WEB HOST

Best Managed Service WINNER The ISPAs 2011 www.ispaawards.org.uk

Find out more about us at www.memset.com

or chat to our sales team on 0800 634 9270.