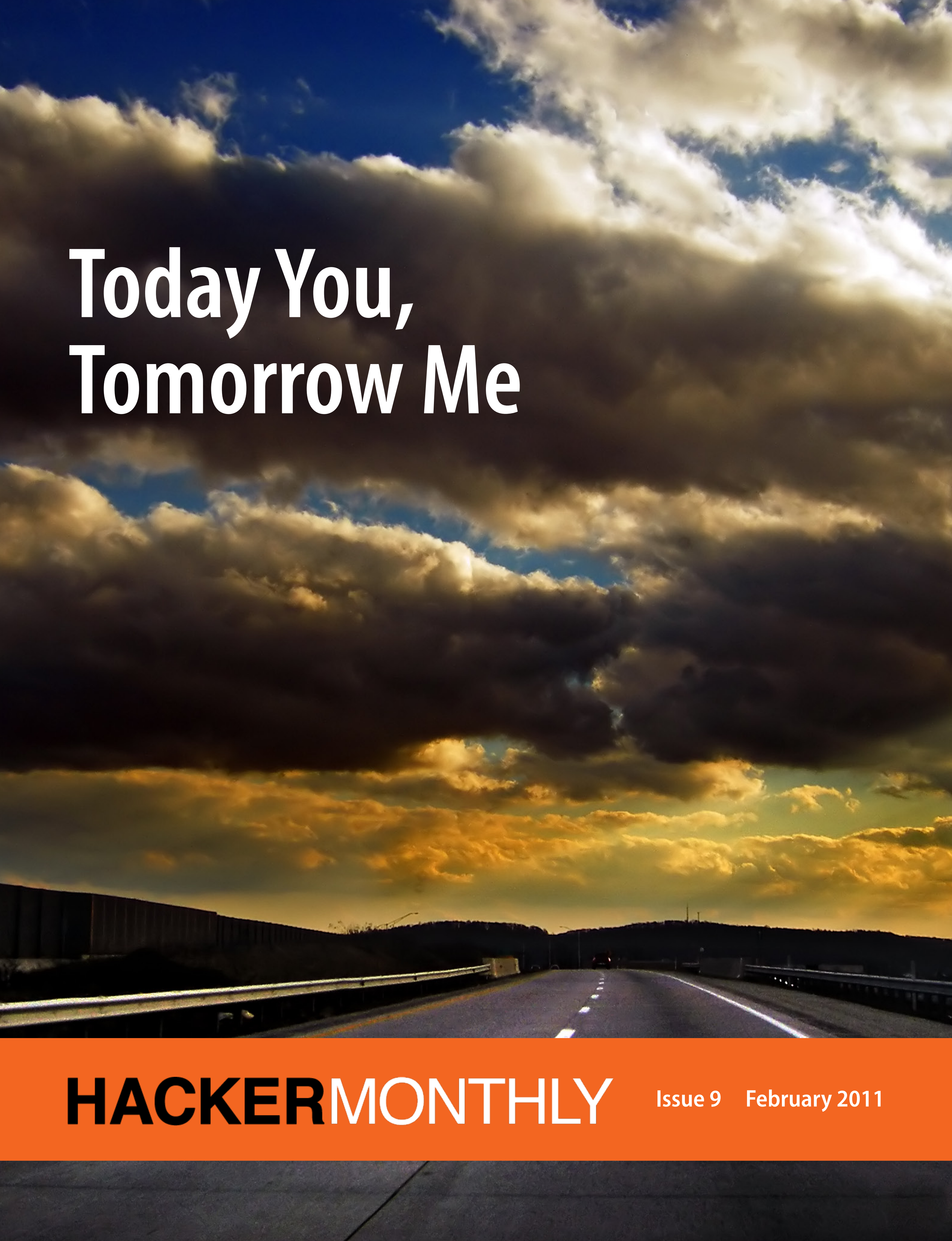


Today You, Tomorrow Me

HACKERMONTHLY Issue 9 February 2011

Curator
Lim Cheng Soon

Proofreaders
Lane Rapp
Molly O'Donnell

Printer
MagCloud

Contributors
ARTICLES
Carlos Bueno
Rhoner
Patrick McKenzie
Rhett Creighton
Kristof KOVACS
Peter Norvig
Dan Mayer
Chad Fowler
Mark Feldman
Jason Shen

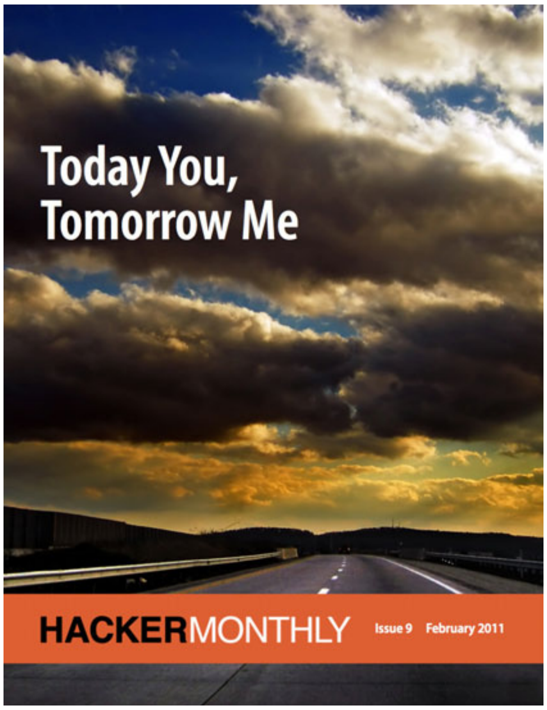
COMMENTARIES
SandB0x
Jake Voytko
Sukotto
Allen Freeman
Drew Haven

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: Nicholas_T [www.flickr.com/photos/nicholas_t/]

Contents

For links to the posts on Hacker News, visit *hackermonthly.com/issue-9*. All articles and comments are reprinted with permission of their original author. Hacker Monthly is published by Netizens Media and not affiliated with Y Combinator in any way.

The Full Stack, Part I

By CARLOS BUENO

One of my most vivid memories from school was the day our chemistry teacher let us in on the Big Secret: every chemical reaction is a joining or separating of links between atoms. Which links form or break is completely governed by the energy involved and the number of electrons each atom has. The principle stuck with me long after I'd forgotten the details. There existed a simple reason for all of the strange rules of chemistry, and that reason lived at a lower level of reality. Maybe other things in the world were like that too.

A "full-stack programmer" is a generalist, someone who can create a non-trivial application by themselves. People who develop broad skills also tend to develop a good mental model of how different layers of a system behave. This turns out to be especially valuable for performance & optimization work. No one can know everything about everything, but you should be able to visualize what happens up and down the stack as an application does its thing. An application is shaped by the requirements of its data, and performance is shaped by how quickly hardware can throw data around.

Consider this harmless-looking SQL query:

```
DELETE FROM some_table WHERE id =
1234;
```

If the `id` column is not indexed, this code will usually result in a table scan: all of the records in `some_table` will be examined one-by-one to see if `id` equals `1234`. Let's assume id is the indexed primary key. That's a good as it gets, right? Well, if the table is in InnoDB format it will result in one disk-seek, because the data is stored next to the primary key and can be deleted in one operation. If the table is MyISAM it will result in at least two seeks, because indexes and data are stored in different files. A hard drive can only do one seek at a time, so this detail can make the difference between 1X or 2X transactions per second. Digging deeper into how these storage engines work, you can find ways to trade safety for even more speed.

The shape of the data

One way to visualize a system is how its data is shaped and how it flows. Here are a some useful factors to think about:

- **Working data size:** This is the amount of data a system has to deal with during normal operation. Often it is identical to the total data size minus things like old logs, backups, inactive accounts, etc. In time-based applications such as email or a news feed the working set can be much smaller than the total set. People rarely access messages more than a few weeks old.
- **Average request size:** How much data does one user transaction have to send over the network? How much data does the system have to touch in order to serve that request? A site with 1 million small pictures will behave differently from a site with 1,000 huge files, even if they have the same data size and number of users. Downloading a photo and running a web search involve similar-sized answers, but the amounts of data touched are very different.
- **Request rate:** How many transactions are expected per user per minute? How many concurrent users are there at peak (your busiest period)? In a search engine you may have 5 to 10 queries per user session. An online ebook reader might see constant but low volumes of traffic. A game may require multiple transactions per second per user.
- **Mutation rate:** This is a measure of how often data is added, deleted, and edited. A webmail system has a high add rate, a lower deletion rate, and an almost-zero edit rate. An auction system has ridiculously high rates for all three.
- **Consistency:** How quickly does a mutation have to spread through the system? For a keyword advertising bid, a few minutes might be acceptable. Trading systems have to reconcile in milliseconds. A comments system is generally expected to show new comments within a second or two.
- **Locality:** This has to do with the probability that a user will read item B if they read item A. Or to put it another way, what portion of the working set does one user session need access to? On one extreme you have search engines. A user might want to query bits from anywhere in the data set. In an email application, the user is guaranteed to only access their inbox.

Knowing that a user session is restricted to a well-defined subset of the data allows you to shard it: users from India can be directed to servers in India.

- **Computation:** what kinds of math do you need to run on the data before it goes out? Can it be precomputed and cached? Are you doing intersections of large arrays? The classic flight search problem requires lots of computation over lots of data. A blog does not.
- **Latency:** How quickly are transactions supposed to return success or failure? Users seem to be ok with a flight search or a credit card transaction taking their time. A web search has to return within a few hundred milliseconds. A widget or API that outside systems depend on should return in 100 milliseconds or less. More important is to maintain application latency within a narrow band. It is worse to answer 90% of queries in 0.1 seconds and the rest in 2 seconds, rather than all requests in 0.2 seconds.
- **Contention:** What are the fundamental bottlenecks? A pizza shop's fundamental bottleneck is the size of its oven. An application that serves random numbers will be limited by how many random-number generators it can employ. An application with strict consistency requirements and a high mutation rate might be limited by lock contention. Needless to say, the more parallelizability and the less contention, the better.

This model can be applied to a system as a whole or to a particular feature like a search page or home page. It's rare that all of the factors stand out for a particular application; usually it's 2 or 3. A good example is ReCAPTCHA. It generates a random pair of images, presents them to the user, and verifies whether the user spelled the words in the images correctly. The working set of data is small enough to fit in RAM, there is minimal computation, a low mutation rate, low per-user request rate, great locality, but very strict latency requirements. I'm told that ReCAPTCHA's request latency (minus network latency) is less than a millisecond.

A horribly oversimplified model of computation

How an application is implemented depends on how real computers handle data. A computer really does only two things: read data and write data. Now that CPU cycles are so fast and cheap, performance is a function of how fast it can read or write, and how much data it must move around to accomplish a given task. For historical reasons we draw a line at operations over data on the CPU or in memory and call that "CPU time". Operations that deal with storage or network are lumped under "I/O wait". This is terrible because it doesn't distinguish between a CPU that's doing a lot of work, and a CPU that's waiting for data to be fetched into its cache. A modern server works with five kinds of input/output, each one slower but with more capacity than the next:

- **Registers & CPU cache (1 nanosecond):** These are small, expensive and very fast memory slots. Memory controllers try mightily to keep this space populated with the data the CPU needs. A cache miss means a 100X speed penalty. Even with a 95% hit rate, CPU cache misses waste half the time.
- **Main memory (10^2 nanoseconds):** If your computer was an office, RAM would be the desk scattered with manuals and scraps of paper. The kernel is there, reserving Papal land-grant-sized chunks of memory for its own mysterious purposes. So are the programs that are either running or waiting to run, network packets you are receiving, data the kernel thinks it's going to need, and (if you want your program to run fast) your working set. RAM is hundreds of times slower than a register but still orders of magnitude faster than anything else. That's why server people go to such lengths to jam more and more RAM in.
- **Solid-state drive (10^5 nanoseconds):** SSDs can greatly improve the performance of systems with working sets too large to fit into main memory. Being "only" one thousand times slower than RAM, solid-state devices can be used as ersatz memory. It will take a few more years for SSDs to replace magnetic disks. And then we'll have to rewrite software tuned for the RAM / magnetic gap and not for the new reality.
- **Magnetic disk (10^7 nanoseconds):** Magnetic storage can handle large, contiguous streams of data very well. Random disk access is what kills performance. The latency gap between RAM and magnetic disks is so great that it's hard to overstate its importance. It's like the difference between having a dollar in your wallet and having your mom send you a dollar in the mail. The other important fact is that access time varies wildly. You can get at any part of RAM or SSD in about the same time, but a hard disk has a physical metal arm that swings around to reach the right part of the magnetic platter.
- **Network (10^6 to 10^9 nanoseconds):** Other computers. Unless you control that computer too, and it's less than a hundred feet away, network calls should be a last resort.

Trust, but verify

The software stack your application runs on is well aware of the memory/disk speed gap, and does its best to juggle things around such that the most-used data stays in RAM. Unfortunately, different layers of the stack can disagree about how best to do that, and often fight each other pointlessly. My advice is to trust the kernel and keep things simple. If you must trust something else, trust the database and tell the kernel to get out of the way.

Thumbs and envelopes

I'm using approximate powers-of-ten here to make the mental arithmetic easier. The actual numbers are less neat. When dealing with very large or very small numbers it's important to get the number of zeros right quickly, and only then sweat the details. Precise, unwieldy numbers usually don't help in the early stages of analysis.

Suppose you have ten million (10^7) users, each with 10MB (10^7) bytes of data, and your network uplink can handle 100 megabits (10^7 bytes) per second. How long will it take to copy that data to another location over the internet? Hmm, that would be 10^7 seconds, or about 4 months: not great, but close to reasonable. You could use compression and multiple uplinks to bring the transfer time down to, say, a week. If the approximate answer had been not 4 but 400 months, you'd quickly drop the copy-over-the-internet idea and look for another answer.

movies.example.com

So can we use this model to identify the performance gotchas of an application? Let's say we want to build a movies-on-demand service like Netflix or Hulu. Videos are professionally produced and 20 and 200 minutes long. You want to support a library of 100,000 (10^5) films and 10^5 concurrent users. For simplicity's sake we'll consider only the actual watching of movies and disregard browsing the website, video encoding, user comments & ratings, logs analysis, etc.

- **Working data size:** The average video is 40 minutes long, and the bitrate is 300kbps. 40 * 60 * 300,000 / 8 is about 10^8 bytes. Times 10^5 videos means that your total working set is 10^13 bytes, or 10TB.
- **Average request size:** A video stream session will transfer somewhere between 10^7 and 10^9 bytes. In Part One we won't be discussing networking issues, but if we were this would be cause for alarm.
- **Request rate:** Fairly low, though the concurrent requests will be high. Users should have short bursts of browsing and long periods of streaming.
- **Mutation rate:** Nearly nil.
- **Consistency:** Unimportant except for user data. It would be nice to keep track of what place they were in a movie and zip back to that, but that can be handled lazily (eg in a client-side cookie).
- **Locality:** Any user can view any movie. You will have the opposite problem of many users accessing the same movie.
- **Computation:** If you do it right, computation should be minimal. DRM or on-the-fly encoding might eat up cycles.
- **Latency:** This is an interesting one. The worst case is channel surfing. In real-world movie services you may have noticed that switching streams or skipping around within one video takes a second or two in the average case. That's at the edge of user acceptability.
- **Contention:** How many CPU threads do you need to serve 100,000 video streams? How much data can one server push out? Why do real-world services seem to have this large skipping delay? When multiple highly successful implementations seem to have the same limitation, that's a strong sign of a fundamental bottleneck.

It's possible to build a single server that holds 10TB of data, but what about throughput? A hundred thousand streams at 300kbps (10^5 * 3 * 10^5) is 30 gigabits per second (3 * 10^10). Let's say that one server can push out 500mbps in the happy case. You'll need at least 60 servers to support 30gbps. That implies about 2,000 concurrent streams per server, which sounds almost reasonable. These guesses may be off by a factor or 2 or 4 but we're in the ballpark.

You could store a copy of the entire 10TB library on each server, but that's kind of expensive. You probably want either:

- A set of origin servers and a set of streaming servers. The origins are loaded with disks. The streamers are loaded with RAM. When a request comes in for a video, the streamer first checks to see if it has a local cache. If not, it contacts the origins and reads it from there.
- A system where each video is copied to only a few servers and requests are routed to them. This might have problems with unbalanced traffic.

An important detail is the distribution of popularity of your video data. If everyone watches the same 2GB video, you could just load the whole file into the RAM of each video server. On the other extreme, if 100,000 users each view 100,000 different videos, you'd need a lot of independent spindles or SSDs to keep up with the concurrent reads. In practice, your traffic will probably follow some kind of power-law distribution in which the most popular video has X users, the second-most has 0.5X users, the third-most 0.33X users, and so on. On one hand that's good; the bulk of your throughput will be served hot from RAM. On the other hand that's bad, because the rest of the requests will be served from cold storage.

Whatever architecture you use, it looks as though the performance of movies.example.com will depend almost completely on the random seek time of your storage devices. If I were building this today I would give both SSDs and non-standard data prefetching strategies a serious look.

It's been fun

This subject is way too large for a short writeup to do it justice. But absurd simplifications can be useful as long as you have an understanding of the big picture: an application's requirements are shaped by the data, and implementations are shaped by the hardware's ability to move data. Underneath every simple abstraction is a world of details and cleverness. The purpose of the big fuzzy picture is to point you where to start digging. ■

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

WUFOO

Making web forms easy + fast + fun

WUFOO

Mortgage Application

Purpose of Mortgage or Loan

Home Loan

WUFOO

Buy a T-Shirt!

Which one do you want? *

Robot Shirt

Color *

Black

WUFOO

Customer Satisfaction Survey

Please take a few moments to complete this satisfaction survey.

How long have you used our product / service?

- Less than a mo
- 1-6 months
- 1-3 years
- Over 3 Years
- Never used

How often do you

- Once a week
- 2 to 3 times a
- Once a month
- Less than onc

Overall, how satisf

- Very Satisfied
- Satisfied
- Neutral

WUFOO

Join our Mailing List

Stay up to date on the latest news and info!

Your Email Address *

Submit

WUFOO

Workshop Registration

Register now while seats are available!

Name *

Title First Last

Address *

Street Address

Address Line 2

City

Postal / Zip Code

WUFOO

Contact Form

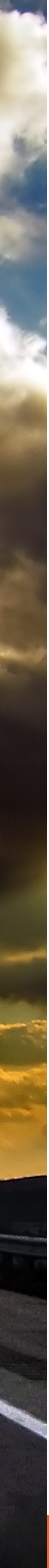
Message *

Today You, Tomorrow Me

By RHONER

JUST ABOUT EVERY time I see someone I stop. I kind of got out of the habit in the last couple of years, moved to a big city and all that, my girlfriend wasn't too stoked on the practice. Then some shit happened to me that changed me and I am back to offering rides habitually. If you would indulge me, it is long story and has almost nothing to do with hitch hiking other than happening on a road.

Photo: Empty Road, *www.flickr.com/photos/nicholas_t/361161401/*



This past year I have had 3 instances of car trouble. A blow out on a freeway, a bunch of blown fuses and an out of gas situation. All of them were while driving other people's cars which, for some reason, makes it worse on an emotional level. It makes it worse on a practical level as well, what with the fact that I carry things like a jack and extra fuses in my car, and know enough not to park, facing downhill, on a steep incline with less than a gallon of fuel.

Anyway, each of these times this shit happened I was disgusted with how people would not bother to help me. I spent hours on the side of the freeway waiting, watching roadside assistance vehicles blow past me, for AAA to show. The 4 gas stations I asked for a gas can at told me that they couldn't loan them out "for my safety" but I could buy a really shitty 1-gallon one with no cap for $15. It was enough, each time, to make you say shit like "this country is going to hell in a handbasket."

But you know who came to my rescue all three times? Immigrants. Mexican immigrants. None of them spoke a lick of the language. But one of those dudes had a profound affect on me.

He was the guy that stopped to help me with a blow out with his whole family of 6 in tow. I was on the side of the road for close to 4 hours. Big jeep, blown rear tire, had a spare but no jack. I had signs in the windows of the car, big signs that said "need a jack" and offered money. No dice. Right as I am about to give up and just hitch out there a van pulls over and dude bounds out. He sizes the situation up and calls for his youngest daughter who speaks english. He conveys through her that he has a jack but it is too small for the Jeep so we will need to brace it. He produces a saw from the van and cuts a log out of a downed tree on the side of the road. We rolled it over, put his jack on top, and bam, in business. I start taking the wheel off and, if you can believe it, I broke his tire iron. It was one of those collapsible ones and I wasn't careful and I snapped the head I needed clean off. Fuck.

No worries, he runs to the van, gives it to his wife and she is gone in a flash, down the road to buy a tire iron. She is back in 15 minutes, we finish the job with a little sweat and cussing (stupid log was starting to give), and I am a very happy man. We are both filthy and sweaty. The wife produces a large water jug for us to wash our hands in. I tried to put a 20 in the man's hand but he wouldn't take it so I instead gave it to his wife as quietly as I could. I thanked them up one side and down the other. I asked the little girl where they lived, thinking maybe I could send them a gift for being so awesome. She says they live in Mexico. They are here so mommy and daddy can pick peaches for the next few weeks. After that they are going to pick cherries then go back home. She asks if I have had lunch and when I told her no she gave me a tamale from their cooler, the best fucking tamale I have ever had.

So, to clarify, a family that is undoubtedly poorer than you, me, and just about everyone else on that stretch of road, working on a seasonal basis where time is money, took an hour or two out of their day to help some strange dude on the side of the road when people in tow trucks were just passing me by. Wow...

But we aren't done yet. I thank them again and walk back to my car and open the foil on the tamale cause I am starving at this point and what do I find inside? My fucking $20 bill! I whirl around and run up to the van and the guy rolls his window down. He sees the $20 in my hand and just shaking his head no like he won't take it. All I can think to say is "Por Favor, Por Favor, Por Favor" with my hands out. Dude just smiles, shakes his head and, with what looked like great concentration, tried his hardest to speak to me in English:

"Today you.... tomorrow me."

Rolled up his window, drove away, his daughter waving to me in the rear view. I sat in my car eating the best fucking tamale of all time and I just cried. Like a little girl. It has been a rough year and nothing has broke my way. This was so out of left field I just couldn't deal.

In the 5 months since I have changed a couple of tires, given a few rides to gas stations and, once, went 50 miles out of my way to get a girl to an airport. I won't accept money. Every time I tell them the same thing when we are through:

"Today you.... tomorrow me." ■

Rhoner is a 20-something designer from Portland, Oregon. He likes spicy foods, walking in the rain, and now an ardent proponent for the rights of migrant workers in this country. They are the heroes we never even think about.

Staging Servers, Source Control & Deploy Workflows

And Other Stuff Nobody Teaches You

By PATRICK MCKENZIE

I WORKED FOR ALMOST three years as a cog in a Japanese megacorporation, and one of the best parts about that experience (perhaps even worth the 70 hour weeks) was that they taught me how to be a professional engineer. Prior to doing so, my workflow generally involved a whole lot of bubble gum, duct tape, and praying. I spent a lot of time firefighting broken software as a result, to the detriment of both my customers and myself. Talking to other software developers has made me realize that I'm not the only person who was never taught that there are options superior to bubblegum. If you aren't lucky enough to work at a company that has good engineering in its very DNA, you're likely to not know much about them.

This strikes me as the industry's attitude to source control a few years ago, prior to a concerted evangelization movement by people like Joel Spolsky. It is virtually impossible to overstate how much using source control improves software development. Our industry has changed in major ways since 2000, but our best practices (and knowledge of those best practices) are lagging a few years behind. We could really use a Joel Test 2010 edition, for a world where "you should have build scripts for desktop software which can complete the build in one step" is largely an anachronism and where the front page to the website is no longer hand-coded in Notepad but, rather, is a shipping piece of software which can break in two hundred ways.

You're not going to get the Joel Test 2010 here, mostly because I'm not Joel and there is no particular reason any company should judge its development practices relative to mine. What I would like to give is some practical pointers for implementing three practices which, if you're not already doing, will greatly improve the experience of writing software for the web:

1. Staging servers
2. Version control workflows
3. Tested, repeatable deployments

Staging Servers

What is a staging server? The basic idea is that it is staging = production - users. (If you're Facebook, Google, or IMVU, you are lightyears ahead of this article and have some system where there are multiple levels of staging/production and where you can dynamically change them. You already have geniuses working on your infrastructure. Listen to them. This article is for people who don't have any option between "code runs on developer's laptop" and "code runs in production.")

Why do we have staging servers? So that anything that is going to break on production breaks on the staging server first. For this reason, you want your staging server to be as similar to the production environment as you can possibly make it. If the production environment processes credit cards, the staging environment processes credit cards. This means that if, e.g., your configuration for the payment gateway is borked, you'll find out about that on the staging server prior to pushing it live to production and, whoopsie, not actually being able to get money from people. If your production server uses Ruby 1.9, your staging server

“Setting up a staging server should be easy. If it is not easy, you already have a problem in your infrastructure, you just don’t know it yet.”

uses 1.9. If the production server uses memcached on port 12345, the staging server uses memcached on port 12345.

(Many folks have systems which exist on more than one physical machine. I don’t—I’m a small business where 2 GB of RAM is enough for anything I want to do. If you have multiple machines, strike “staging server” and read as “staging system” below: all the benefits for having a separate staging server are still beneficial when your staging environment actually has fifteen physical servers running 47 VMs.)

Setting up a staging server should be easy. If it is not easy, you already have a problem in your infrastructure, you just don’t know it yet: you’ve cobbled together your production server over time, usually by manually SSHing into it and tweaking things without keeping a log of what you have done. (Been there, done that, got the “I Created A Monster” T-shirt.) There isn’t a written procedure or automated script for creating it from the bare metal. If you had that procedure written, you should be able to execute it and create a staging server that works inside of an hour.

Most people won’t be able to do this if they haven’t given thought to the matter before. That is fixable, and should be fixed. It has substantial benefits: if you have a repeatable procedure for provisioning a production system, then when disaster strikes you will be able to confidently execute that procedure and end up with a production system. (Confidence is important since you’ll probably be terrified and rushed when you need to do this, and rushed terrified people make unnecessary mistakes.)

If you’re working on Rails, I highly recommend using Deprec/Capistrano with all new projects. In addition to making it very easy to get a full Rails stack working on your deployment environment of choice , it helps automate routine deployment and server maintenance, and has mostly sensible defaults. (I have only one quibble with deprec: it installs software from source rather than using your system’s package manager. That means that upgrading e.g. Nginx two years down the road is needlessly hard and error prone, when instead you could just have used apt-get in the first place and then updating is a piece of cake.)

You can also use Fabric, Chef, Fog, or a similar system to script up building new environments. Pick whichever strikes your fancy. Try to recreate your production environment, down to a T, on another host at your VPS/cloud/etc provider, or on another physical machine if you actually still own machines. Keep tweaking the script until it produces something which actually matches your production environment. You now have a procedure for creating a staging server, and as an added bonus it also works for documenting your production environment in a reproducible fashion.

One nice thing about keeping your server configuration in scripts rather than just splayed across fifteen different places on the server (/etc/environment, /etc/crontab, /usr/local/nginx/conf/apps/AppName.conf, etc) is that it lives in source control. Your cron jobs? If they’re in source control, you’ll have a written record of what they are, what they’re supposed to do, and why they just blew up when you bork the underlying assumptions eight months down the line. Your Nginx config? If it is in source control, you’ll understand why you added that new location setting for static images only. The voodoo in your postfix config? A suitably descriptive commit note means you'll never have to think about reproducing the voodoo again.

After you have the script which will produce your staging environment, you probably want to make a minimum number of alterations from production. Many companies will want their staging environment to be non-public—that way, customers don't see code before it is ready, and critical issues never affect the outside world. There are many ways to do this: ideally, you'd just tweak a setting on your firewall and bam, nobody from the public Internet can get to your staging environment. However, this is a wee bit difficult to pull off for some of us. For one, I don't actually have a hardware firewall (I use iptables on each VPS individually).

My staging environment simply includes a snippet in Nginx which denies access to everyone except a particular host (which I can proxy through). This breaks integration with a few outside services (e.g. Twilio and Spreedly, which needs callbacks), so I make exceptions for the URLs those two need to access. The more complicated your staging server configuration gets relative to production, the more likely you are to compromise its utility. Try to avoid exceptions.

That said, there are a couple that are too valuable to not make. For example, my staging server has a whitelist of email addresses and phone numbers owned by me. Through the magic of monkeypatching, attempting to contact anyone else raises an exception. That sounded a little paranoid until that day when I accidentally created an infinite loop and rang every number in the database a hundred times. (My cell phone company loves me, but folks who accidentally collided with test data sure would not have.)

How do you get data to populate the staging server? I use seed scripts and add more data by hand. (I also have DB dumps available, but they tend to go stale against the current schema distressingly quickly: I recommend seed scripts.) You can also dump the production DB and load it into the staging DB. Think long and hard before you do this. For one, it is likely to be way, way the heck out of bounds for regulated industries. For another, your staging server is probably going to periodically be insecure—insecurity is failure and failure is what the staging server is for. Slurping all of the data out of a staging environment has caused many companies smarter than you to have to go into disaster management mode. Please be careful.

So you've got a staging server? Now what?

At the simplest, you access your staging server with a browser and try to break things. When you break things, you fix things, then you redeploy the staging server and try to break them again. This is what you are probably doing right now with production, except that your customers don't have to see broken things when you break things.

Eventually, you can script up attempts to break things, using e.g. Selenium. Then when you break things, you add them to the list of things that Selenium tries to break. If you run that against the staging server after every code check in (a process known as continuous integration), you'll quickly catch regressions before they disrupt paying customers. This is a wee bit harder than just having a staging server—OK, a lot harder—but you'll get clear, obvious advantage out of every increment of work you do on this path, so don't let present inability to be Google prevent you from getting started.

Version Control & Deployment Workflows

Everyone should use version control, but people tend to use version control differently. Git is very popular in the Rails community, but there are probably no two companies using Git the same way. The key thing is that you agree with your team on how you use version control — document your assumptions, document your processes, then apply them religiously. This will reduce conflicts on the team, reduce mistakes, and help you get more out of your tools.

There are a million ways to use version control and most of them are perfectly OK. I'm going to mention mine, but it isn't the canonical Right Way, it is just one way which works for a (very) small company. Yours will likely be different, but you can see some of the things which go into design of a version control workflow.

Assumptions I Make About Life, the Universe, and Everything

1. I use Git. Git has notion of branches, tags, and remotes (physically distinct repositories) — if you don't know what these are, Google for "getting started with Git".

2. I generally work alone or with a very small team. (This assumption underpins very important parts of my workflow. It won't expand very well to a 200 man distributed team, but it might well work for 2 ~ 5 people.)

3. There is exactly one canonical repository, origin. Developers maintain other repositories on their workstation. Automated processes like deployment happen only with reference to the origin. Code existing outside of the origin does not officially exist yet, for any purpose. The history preserved in the origin is, in principle, sacred.

4. There is a branch called deploy. The HEAD of deploy (the most recent code on it) is presumptively ready to be put into production.

5. Tags are used to take snapshots of the code base and preserve them in amber with a human readable name. Right before we deploy to either production or staging, the HEAD gets tagged, so that we can easily find it later, with a simple naming convention (I use production_release_X and staging_release_X, where X just increments upwards—some people might prefer timestamps). Production release tags are never deleted. Staging tags get periodically culled when convenient to do so.
6. Development of any feature expected to take longer than a few hours happens on a feature branch. (I do occasional work right on deploy locally, for issues of the "Minor copy edit on dashboard." variety. This would be one of the first things to go if I were working on a larger team.)

So how does this actually work in practice? Let's say I'm implementing a new feature. I create a new branch to work on. I code a bit, creating local commits with wild abandon any time I have accomplished something which I don't want to lose. When I believe code to be functional, I fire a capistrano task which tags the current head of my branch, pushes that tag to origin, and deploys it to the staging server. I then continue testing on the staging server, for example verifying that Twilio integration actually works with Twilio (which cannot conveniently access localhost:3000 on my laptop). I continue writing code, committing, tagging, and pushing to the staging server until the feature is ready.

Then, I switch back to the deploy branch and merge in my feature branch (with–no-ff, which creates a commit message just for the merge—this handily groups the twenty or thirty commits I just made into one easily readable story for myself later). I then tag a production release (manually—this is entirely to force me to think through whether I'm ready for a production release), verify that there is no diff between it and the most recent staging release, and then push the new tag to origin. I then fire the Capistrano task which checks out the new deployment tag and restarts the server.

What does this get me versus my previous SVN workflow for Bingo Card Creator, which was "Work only on one branch, commit stuff when I think it is ready, and deploy the trunk manually on occasion"?

1. I cause much less downtime for the production server due to reasons like *svn commit -m 'Whoops, forgot a setting in production.rb'* and *svn commit -m 'r1234 introduced dependency on foobar gem without putting it in environment file, causing rake gems:install to not load it. Mongrels then failed to restart.'*
2. My deploy branch has a relatively clean history, so when things start to break next year in production, finding the change sets which eventually caused the breakage will be less of a needle in the haystack search than finding them in SVN is. SVN's history is 1800 unedited commits, recording my stream of consciousness as they happened. My stream of consciousness is frequently stupid, particularly when I'm panicking because the server is down.
3. This decouples the staging server from production in a clean fashion (so that I can advance the staging server a feature or three at a time if I want to), but guarantees that when I'm actually ready to deploy, I'm deploying exactly what did not break on the staging server.
4. Tagging releases gives you an Oh Crikey button, as in Oh Crikey, that last release broke stuff. You can quickly rollback the deploy to a known good tag, isolate the changes which broke production, and fix them.
5. Deploy scripts manage releases with multiple moving parts a lot better than I do, even when I'm working from a checklist.

By the way, Git gives you many options for recovering from problems—even severe problems—without requiring either gymnastics or a full-blown CSI investigation to discover what happened later. For example, let's pretend I just deployed tag production_deploy_82, and have discovered some issue serious enough to require an immediate rollback to production_deploy_81, which is known to be good:

```
#Assuming we are on our local worksta-
tion on the deploy branch.
git branch something-in-here-is-broken
git reset --hard production_deploy_81
#All changes made between deploy 81
and 82 just vanished from the deploy
branch locally.
#Clean up the deploy, using any option
discussed below.
git checkout something-in-here-is-broken
#Those changes which you just disap-
peared are now living on this branch,
ready for you to fix. After you've fixed
and verified it works on the stag-
ing server (and, ahem,that you have
addressed the issue that allowed this
to get OKed for release last time), you
merge this branch back into deploy,
and do a tag-and-release cycle.
```

How you clean up the mess on the server is up to you: good options include "deploy 81 again", "tag a release 83 equivalent to 81, then deploy it", and "rollback to the copy of 81 which still exists on the server." (Capistrano includes deploy:rollback, which will do exactly this.) Any of these will work, just always do it the same way to avoid stepping on each others' toes. I prefer tagging a new release so that I can add a descriptive message explaining why 82 just created an emergency.

This is important because it leaves a paper trail—if you're pulling a release from production, something just went seriously wrong with your processes. Emergencies are not supposed to happen—anything that lets an issue get that far isn't just a one-off failure of whatever broke, it is a series of failures of the systems/processes designed to prevent failures from getting that far. After you've put out the fire, investigate what went wrong and tweak your processes such that a similar failure in the future gets caught prior to bringing down production. The sleep you save may be your own.

Scaling this to more programmers: Do whatever works for you! I would probably create a staging branch and have folks integrate stuff into the staging branch when it was ready to go to the official staging environment. I also might make per-developer staging environments: since creating one from the bare metal is supposed to be essentially free, let them all have their own where they can be reckless without spoiling the experience of other developers. We can worry about code interaction on the "real" staging server. Then, have folks communicate when they consider everything they have on staging ready for release, and release when everybody says it is ready.

The important thing is that, whatever process you use, you document it, teach it, and enforce it.

Stuff Your Deployment Script Might Not Do Today But Probably Should

1. Depending on your scale and how you use e.g. memcached, it might be safe to purge the cache on every re-deploy, which will prevent some hard to diagnose bugs. At a certain scale, this is virtually a recipe for taking your site down in a cache stampede, but I'm not Facebook and having capacity problems means that I am probably already vacationing at my hollowed-out volcano lair.

2. Tell everybody on the team that you just deployed. I know some teams who have an IRC channel with a bot who announces redeploys. A quick email CCed to five developers also probably suffices.

3. Restart worker processes. This is easy to forget but, if you do it by hand, you'll eventually forget and then have two versions of the application in production at once. If you're not prepared for that, it will bite you on the hindquarters when, e.g., the application servers ask the workers to execute methods that the workers do not know now exist in the code base.

4. Do sanity checks. You can go arbitrarily deep with complexity here. For a first cut, mine for Appointment Reminder restarts the application server, counts ten seconds, then tries to access an internal URL. If the application server isn't up, or if the action at that URL blows up for any reason, the deployment script fails the deploy, rolls back to a known-good version, and sends me a very crossly worded email. (You can do this for the staging server, too.)

5. Integrate with other systems which manage the state of your code/business. For example, I use Hoptoad. Hoptoad keeps track of exceptions and mails you when they happen, in such a fashion that your inbox doesn't get buried by e.g. Googlebot deciding to do an impromptu fuzz test on your website. I mark all exceptions as resolved every time I deploy to the environment they happened in. You could also e.g. update an internal wiki by adding a new page specific to the deployment, automatically update your bug tracker to change the status of the bugs that you (presumably) just squashed, or start a new cohort for your stats tracking. ■

Patrick McKenzie runs a small software business. His current focus is on Appointment Reminder, which solves small businesses' problems with missed appointments. He also made Bingo Card Creator and consults from time to time, mostly on software marketing.

SendGrid
Email Delivery. Simplified.

We make sure YOUR EMAILS ALWAYS reach your customers' INBOX.

Reliable delivery of transactional emails from your application:

- registration confirmations
- password reminders
- purchase receipts

www.SendGrid.com/hacker

simple integration via SMTP relay or Web API

Code Fearlessly

By RHETT CREIGHTON

Photo: The Golden Hall, *www.flickr.com/photos/maxfwilliams/3308461831/*

When Dane Jensen first started to work on Cam.ly, he dove right into things and wanted to learn how everything worked. It was no small feat, considering that our system already used (in addition to others) Ruby on Rails, Haml / Sass, JavaScript, Java, shell, C, and C++.

After a few days, he was ready for a task, and I gave him some tough problems that we had to solve. We had some great brainstorming sessions, and decided that there were a few potential ways that we could approach solving some of our major problems, which he would work on. A few days later, I came back to see how he was doing, and noticed that he had apparently lost a lot of his initial energy. I asked him how his work was going on the big problems and he explained that he was intimidated and didn't want to break anything. Then I said two words to Dane that changed him forever:

"Code Fearlessly"

All of the code he was working on was versioned in Git. He was working entirely on development machines (not production). There was absolutely no way for him to break anything.

I decided that "Coding Fearlessly" was critical to being an extremely productive programmer by watching Nat Friedman. Nat is one of the best programmers I know, and he truly loves working on software.

One day I watched Nat deleting and changing a lot of code that people had obviously spent a lot of time writing. Some people might feel scared to even save the file after deleting so much code. Nat didn't hesitate at all. He said, "Ok, well this is all in Git," and just started deleting. He was right. There was nothing he could do that would set back anyone else's work, and even if he pushed to a development server (not likely unless he was sure it was a good commit), it would probably only take someone a few minutes to roll things back to the way things were.

"To truly code fearlessly, an environment must be created where there is truly nothing for the coder to fear."

There has been a lot of excitement, hype, and potentially disappointment when software development processes such as, XP (eXtreme Programming), TDD (Test Driven Development), or BDD (Behavior Driven Development), work really well for some teams, but not others. A huge benefit of TDD is that in some teams, on some projects, it creates a safety net where people are able to code fearlessly, and as long as all of the tests pass, they can push code. The benefits from having developers who work fearlessly without disrupting each other are enormous on any project.

Thinking about it further, I realized that this also reminded me of a story that the inventor, roboticist, and entrepreneur, Thomas Massie, once told me. When he was a child, he was fortunate enough that his parents bought a computer, and he desperately wanted to start making robots with it. However, he was smart enough to know that it was a bad idea to start sticking wires into the family computer that cost thousands of dollars. So, Thomas hatched a plan. He figured out that he could scotch-tape photo sensors to the computer screen and write programs that turned portions of the screen either on full brightness or full darkness. That way, he could write programs that controlled motors, without electrically connecting anything to the computer itself.

Many years later, at MIT, Thomas realized that as young child, he had re-invented the Opto-isolator, a device that gave him the freedom to work fearlessly with a computer.

While the benefits of "Coding Fearlessly" are clear to me, I think it's important to make the distinction from "Coding Recklessly." To truly code fearlessly, an environment must be created where there is truly nothing for the coder to fear. We developers are fortunate to finally have, as of the past few years, tools that can allow this for all developers cost effectively. Distributed version control (Git, Mercurial), virtual machines locally or in the cloud, laptops powerful enough to run databases, smartphone emulators, and many other pieces of technology (hardware or software), can all be used to put together development environments for software engineers that are very much unlike the days of the past.

Whoever is setting up the development environment for any project, whether your team is 1 person or 100 people, it doesn't matter if you choose "agile" or "waterfall." Your primary concern should be to create an environment where you developers can code fearlessly. ■

Rhett Creighton studied physics and nuclear engineering at MIT. He has worked on internet services for the past 10 years, including "Suse Studio" [susestudio.com], a service that allows you to create your own linux appliance through the web, and most recently Cam.ly [cam.ly], which makes it easy for people to install wireless security cameras anywhere.

Redis vs HBase vs Cassandra vs

By KRISTOF KOVACS

WHILE SQL DATABASES are insanely useful tools, their tyranny of ~15 years is coming to an end. And it was just time: I can't even count the things that were forced into relational databases, but never really fitted them.

But the differences between "NoSQL" databases are much bigger than it ever was between one SQL database and another. This means that it is a bigger responsibility on software architects to choose the appropriate one for a project right at the beginning.

In this light, here is a comparison of Redis, HBase, Cassandra, CouchDB, MongoDB and Riak:

Redis

- **Written in**: C/C++
- **Main point**: Blazing fast
- **License**: BSD
- **Protocol**: Telnet-like
- Disk-backed in-memory database,
- but since 2.0, it can swap to disk.
- Master-slave replication
- Simple keys and values,
- but complex operations like ZREVRANGEBYSCORE
- INCR & co (good for rate limiting or statistics)
- Has sets (also union/diff/inter)
- Has lists (also a queue; blocking pop)
- Has hashes (objects of multiple fields)
- Of all these databases, only Redis does transactions (!)
- Values can be set to expire (as in a cache)
- Sorted sets (high score table, good for range queries)
- Pub/Sub and WATCH on data changes (!)

Best used: For rapidly changing data with a foreseeable database size (should fit mostly in memory).

For example: Stock prices. Analytics. Real-time data collection. Real-time communication.

HBase

- **Written in**: Java
- **Main point**: Billions of rows X millions of columns
- **License**: Apache
- **Protocol**: HTTP/REST (also Thrift)
- Modeled after BigTable
- Map/reduce with Hadoop
- Query predicate push down via server side scan and get filters
- Optimizations for real time queries
- A high performance Thrift gateway
- HTTP supports XML, Protobuf, and binary
- Cascading, hive, and pig source and sink modules
- JRuby-based (JIRB) shell
- No single point of failure
- Rolling restart for configuration changes and minor upgrades
- Random access performance is like MySQL

Best used: If you're in love with BigTable. And when you need random, realtime read/write access to your Big Data.

For example: Facebook Messaging Database.

Cassandra

- **Written in**: Java
- **Main point**: Best of BigTable and Dynamo
- **License**: Apache
- **Protocol**: Custom, binary (Thrift)
- Tunable trade-offs for distribution and replication (N, R, W)
- Querying by column, range of keys
- BigTable-like features: columns, column families
- Writes are much faster than reads (!)
- Map/reduce possible with Apache Hadoop
- I admit being a bit biased against it, because of the bloat and complexity it has partly because of Java (configuration, seeing exceptions, etc)

Best used: When you write more than you read (logging). If every component of the system must be in Java. ("No one gets fired for choosing Apache's stuff.")

For example: Banking, financial industry (though not necessarily for financial transactions, but these industries are much bigger than that.) Writes are faster than reads, so one natural niche is real time data analysis.

CouchDB vs MongoDB vs Riak

CouchDB

- **Written in**: Erlang
- **Main point**: DB consistency, ease of use
- **License**: Apache
- **Protocol**: HTTP/REST
- Bi-directional (!) replication,
- continuous or ad-hoc,
- with conflict detection,
- thus, master-master replication. (!)
- MVCC — write operations do not block reads
- Previous versions of documents are available
- Crash-only (reliable) design
- Needs compacting from time to time
- Views: embedded map/reduce
- Formatting views: lists & shows
- Server-side document validation possible
- Authentication possible
- Real-time updates via _changes (!)
- Attachment handling
- thus, CouchApps (standalone js apps)
- jQuery library included

Best used: For accumulating, occasionally changing data, on which pre-defined queries are to be run. Places where versioning is important.

For example: CRM, CMS systems. Master-master replication is an especially interesting feature, allowing easy multi-site deployments.

MongoDB

- **Written in**: C++
- **Main point**: Retains some friendly properties of SQL. (Query, index)
- **License**: AGPL (Drivers: Apache)
- **Protocol**: Custom, binary (BSON)
- Master/slave replication
- Queries are JavaScript expressions
- Run arbitrary JavaScript functions server-side
- Better update-in-place than CouchDB
- Sharding built-in
- Uses memory mapped files for data storage
- Performance over features
- After crash, it needs to repair tables
- Better durablity coming in V1.8

Best used: If you need dynamic queries. If you prefer to define indexes, not map/reduce functions. If you need good performance on a big DB. If you wanted CouchDB, but your data changes too much, filling up disks.

For example: For all things that you would do with MySQL or PostgreSQL, but having predefined columns really holds you back.

Riak

- **Written in**: Erlang & C, some JavaScript
- **Main point**: Fault tolerance
- **License**: Apache
- **Protocol**: HTTP/REST
- Tunable trade-offs for distribution and replication (N, R, W)
- Pre- and post-commit hooks,
- for validation and security.
- Built-in full-text search
- Map/reduce in JavaScript or Erlang
- Comes in "open source" and "enterprise" editions

Best used: If you want something Cassandra-like (Dynamo-like), but no way you're gonna deal with the bloat and complexity. If you need very good single-site scalability, availability and fault-tolerance, but you're ready to pay for multi-site replication.

For example: Point-of-sales data collection. Factory control systems. Places where even seconds of downtime hurt. ■

Kristof is a software architect and consultant from Europe. He works on high-availability, high-security systems in real-time stock exchanges and in the energy industry. You can learn more about him at *kkovacs.eu*.

Reprinted with permission of the original author.
First appeared in *hn.my/nosql*.

How to Write a Spelling Corrector

By PETER NORVIG

IN THE PAST week, two friends (Dean and Bill) independently told me they were amazed at how Google does spelling correction so well and quickly. Type in a search like [speling] and Google comes back in 0.1 seconds or so with "Did you mean: spelling" (Yahoo and Microsoft are similar). What surprised me is that I thought Dean and Bill, being highly accomplished engineers and mathematicians, would have good intuitions about statistical language processing problems such as spelling correction. But they didn't, and come to think of it, there's no reason they should: it was my expectations that were faulty, not their knowledge.

I figured they and many others could benefit from an explanation. The full details of an industrial-strength spell corrector like Google's would be more confusing than enlightening, but I figured that on the plane flight home, in less than a page of code, I could write a toy spelling corrector that achieves 80 or 90% accuracy at a processing speed of at least 10 words per second.

So here, in 21 lines of Python 2.5 code, is the complete spelling corrector:

```
import re, collections

def words(text): return re.findall('[a-z]+', text.lower())

def train(features):
    model = collections.defaultdict(lambda: 1)
    for f in features:
        model[f] += 1
    return model

NWORDS = train(words(file('big.txt').read()))

alphabet = 'abcdefghijklmnopqrstuvwxyz'

def edits1(word):
   splits     = [(word[:i], word[i:]) for i in
                 range(len(word) + 1)]
   deletes    = [a + b[1:] for a, b in splits if b]
   transposes = [a + b[1] + b[0] + b[2:] for a, b in splits
                 if len(b)>1]
   replaces   = [a + c + b[1:] for a, b in splits for c in
                 alphabet if b]
   inserts    = [a + c + b for a, b in splits for c in
                 alphabet]
   return set(deletes + transposes + replaces + inserts)

def known_edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in
edits1(e1) if e2 in NWORDS)

def known(words): return set(w for w in words if w in NWORDS)

def correct(word):
    candidates = known([word]) or known(edits1(word)) or
known_edits2(word) or [word]
    return max(candidates, key=NWORDS.get)
```

The code defines the function `correct`, which takes a word as input and returns a likely correction of that word. For example:

```
>>> correct('speling')
'spelling'
>>> correct('korrecter')
'corrector'
```

The version of `edits1` shown here is a variation on one proposed by Darius Bacon; I think this is clearer than the version I originally had. Darius also fixed a bug in the function `correct`.

How It Works: Some Probability Theory

How does it work? First, a little theory. Given a word, we are trying to choose the most likely spelling correction for that word (the "correction" may be the original word itself). There is no way to know for sure (for example, should "lates" be corrected to "late" or "latest"?), which suggests we use probabilities. We will say that we are trying to find the correction *c*, out of all possible corrections, that maximizes the probability of *c* given the original word *w*:

$\text{argmax}_c \ P(c|w)$

By Bayes' Theorem this is equivalent to:

$\text{argmax}_c \ P(w|c)\ P(c) / P(w)$

Since P(*w*) is the same for every possible *c*, we can ignore it, giving:

$\text{argmax}_c \ P(w|c)\ P(c)$

There are three parts of this expression. From right to left, we have:

1. P(*c*), the probability that a proposed correction *c* stands on its own. This is called the **language model**: think of it as answering the question "how likely is *c* to appear in an English text?" So P("the") would have a relatively high probability, while P("zxzxzxzyyy") would be near zero.
2. P(*w*|*c*), the probability that w would be typed in a text when the author meant *c*. This is the **error model**: think of it as answering "how likely is it that the author would type *w* by mistake when *c* was intended?"
3. $\text{argmax}_{c,}$ the control mechanism, which says to enumerate all feasible values of *c*, and then choose the one that gives the best combined probability score.

One obvious question is: why take a simple expression like P(*c*|*w*) and replace it with a more complex expression involving two models rather than one? The answer is that P(*c*|*w*) is already conflating two factors, and it is easier to separate the two out and deal with them explicitly. Consider the misspelled word *w*="thew" and the two candidate corrections *c*="the" and *c*="thaw". Which has a higher P(*c*|*w*)? Well, "thaw" seems good because the only change is "a" to "e", which is a small change. On the other hand, "the" seems good because "the" is a very common word, and perhaps the typist's finger slipped off the "e" onto the "w". The point is that to estimate P(*c*|*w*) we have to consider both the probability of c and the probability of the change from *c* to *w* anyway, so it is cleaner to formally separate the two factors.

Now we are ready to show how the program works. First P(*c*). We will read a big text file, big.txt [norvig.com/big.txt], which consists of about a million words. The file is a concatenation of several public domain books from Project Gutenberg and lists of most frequent words from Wiktionary and the British National Corpus. (On the plane all I had was a collection of Sherlock Holmes stories that happened to be on my laptop; I added the other sources later and stopped adding texts when they stopped helping, as we shall see in the Evaluation section.)

We then extract the individual words from the file (using the function `words`, which converts everything to lowercase, so that "the" and "The" will be the same and then defines a word as a sequence of alphabetic characters, so "don't" will be seen as the two words "don" and "t"). Next we train a probability model, which is a fancy way of saying we count how many times each word occurs, using the function `train`. It looks like this:

```
def words(text): return re.findall('[a-z]+', text.lower())

def train(features):
    model = collections.defaultdict(lambda: 1)
    for f in features:
        model[f] += 1
    return model

NWORDS = train(words(file('big.txt').read()))
```

At this point, `NWORDS[w]` holds a count of how many times the word `w` has been seen. There is one complication: novel words. What happens with a perfectly good word of English that wasn't seen in our training data? It would be bad form to say the probability of a word is zero just because we haven't seen it yet. There are several standard approaches to this problem; we take the easiest one, which is to treat novel words as if we had seen them once. This general process is called **smoothing**, because we are smoothing over the parts of the probability distribution that would have been zero, bumping them up to the smallest possible count. This is achieved through the class `collections.defaultdict`, which is like a regular Python dict (what other languages call hash tables) except that we can specify the default value of any key; here we use 1.

Now let's look at the problem of enumerating the possible corrections *c* of a given word *w*. It is common to talk of the **edit distance** between two words: the number of edits it would take to turn one into the other. An edit can be a deletion (remove one letter), a transposition (swap adjacent letters), an alteration (change one letter to another) or an insertion (add a letter). Here's a function that returns a set of all words *c* that are one edit away from *w*:

```
def edits1(word):
   splits     = [(word[:i], word[i:]) for i in
                 range(len(word) + 1)]
   deletes    = [a + b[1:] for a, b in splits if b]
   transposes = [a + b[1] + b[0] + b[2:] for a, b in splits
                 if len(b)>1]
   replaces   = [a + c + b[1:] for a, b in splits for c in
                 alphabet if b]
   inserts    = [a + c + b     for a, b in splits for c in
                 alphabet]
   return set(deletes + transposes + replaces + inserts)
```

This can be a big set. For a word of length *n*, there will be *n* deletions, *n*-1 transpositions, 26*n* alterations, and 26(*n*+1) insertions, for a total of 54*n*+25 (of which a few are typically duplicates). For example, len(edits1('something')) — that is, the number of elements in the result of edits1('something') — is 494.

The literature on spelling correction claims that 80 to 95% of spelling errors are an edit distance of 1 from the target. As we shall see shortly, I put together a development corpus of 270 spelling errors, and found that only 76% of them have edit distance 1. Perhaps the examples I found are harder than typical errors. Anyway, I thought this was not good enough, so we'll need to consider edit distance 2. That's easy: just apply `edits1` to all the results of `edits1`:

```
def edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in edits1(e1))
```

This is easy to write, but we're starting to get into some serious computation: len(edits2('something')) is 114,324. However, we do get good coverage: of the 270 test cases, only 3 have an edit distance greater than 2. That is, `edits2` will cover 98.9% of the cases; that's good enough for me. Since we aren't going beyond edit distance 2, we can do a small optimization: only keep the candidates that are actually known words. We still have to consider all the possibilities, but we don't have to build up a big set of them. The function `known_edits2` does this:

```
def known_edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in
edits1(e1) if e2 in NWORDS)
```

Now, for example, known_edits2('something') is a set of just 4 words: {'smoothing', 'seething', 'something', 'soothing'}, rather than the set of 114,324 words generated by `edits2`. That speeds things up by about 10%.

Now the only part left is the error model, P(*w*|*c*). Here's where I ran into difficulty. Sitting on the plane, with no internet connection, I was stymied: I had no training data to build a model of spelling errors. I had some intuitions: mistaking one vowel for another is more probable than mistaking two consonants; making an error on the first letter of a word is less probable, etc. But I had no numbers to back that up. So I took a shortcut: I defined a trivial model that says all known words of edit distance 1 are infinitely more probable than known words of edit distance 2, and infinitely less probable than a known word of edit distance 0. By "known word" I mean a word that we have seen in the language model training data — a word in the dictionary. We can implement this strategy as follows:

```
def known(words): return set(w for w in words if w in NWORDS)

def correct(word):
    candidates = known([word]) or known(edits1(word)) or
known_edits2(word) or [word]
    return max(candidates, key=lambda w: NWORDS[w])
```

The function `correct` chooses as the set of candidate words the set with the shortest edit distance to the original word, as long as the set has some known words. Once it identifies the candidate set to consider, it chooses the element with the highest P(*c*) value, as estimated by the `NWORDS` model.

Evaluation

Now it is time to evaluate how well this program does. On the plane I tried a few examples, and it seemed okay. After my plane landed, I downloaded Roger Mitton's Birkbeck spelling error corpus [www.ota.ox.ac.uk/headers/0643.xml] from the Oxford Text Archive. From that I extracted two test sets of corrections. The first is for development, meaning I get to look at it while I'm developing the program. The second is a final test set, meaning I'm not allowed to look at it, nor change my program after evaluating on it. This practice of having two sets is good hygiene; it keeps me from fooling myself into thinking I'm doing better than I am by tuning the program to one specific set of tests. Here I show an excerpt of the two tests and the function to run them; to see the complete set of tests (along with the rest of the program), see the file spell.py [norvig.com/spell.py].

```
tests1 = { 'access': 'acess', 'accessing': 'accesing',
'accommodation': 'accomodation acommodation acomodation',
'account': 'acount', ...}

tests2 = {'forbidden': 'forbiden', 'decisions': 'deciscions
descisions', 'supposedly': 'supposidly', 'embellishing':
'embelishing', ...}

def spelltest(tests, bias=None, verbose=False):
    import time
    n, bad, unknown, start = 0, 0, 0, time.clock()
    if bias:
        for target in tests: NWORDS[target] += bias
    for target,wrongs in tests.items():
        for wrong in wrongs.split():
            n += 1
            w = correct(wrong)
            if w!=target:
                bad += 1
                unknown += (target not in NWORDS)
                if verbose:
                    print '%r => %r (%d); expected %r (%d)'
                      % (wrong, w, NWORDS[w], target,
                         NWORDS[target])
    return dict(bad=bad, n=n, bias=bias, pct=int(100. -
             100.*bad/n), unknown=unknown,
             secs=int(time.clock()-start) )

print spelltest(tests1)
print spelltest(tests2) ## only do this after everything is
debugged
```

This gives the following output:

```
{'bad': 68, 'bias': None, 'unknown': 15, 'secs': 16, 'pct':
74, 'n': 270}
{'bad': 130, 'bias': None, 'unknown': 43, 'secs': 26,
'pct': 67, 'n': 400}
```

So on the development set of 270 cases, we get 74% correct in 13 seconds (a rate of 17 Hz), and on the final test set we get 67% correct (at 15 Hz).

In conclusion, I met my goals for brevity, development time, and runtime speed, but not for accuracy.

Future Work

Let's think about how we could do better. We'll again look at all three factors of the probability model: (1) P(*c*); (2) P(*w*|*c*); and (3) argmax_c. We'll look at examples of what we got wrong. Then we'll look at some factors beyond the three...

1. P(*c*), the language model. We can distinguish two sources of error in the language model. The more serious is unknown words. In the development set, there are 15 unknown words, or 5%, and in the final test set, 43 unknown words or 11%. Here are some examples of the output of `spelltest` with `verbose=True`:

```
correct('economtric') => 'economic' (121); expected 'econo-
metric' (1)
correct('embaras') => 'embargo' (8); expected 'embarrass'
(1)
correct('colate') => 'coat' (173); expected 'collate' (1)
correct('orentated') => 'orentated' (1); expected 'orien-
tated' (1)
correct('unequivocaly') => 'unequivocal' (2); expected
'unequivocally' (1)
correct('generataed') => 'generate' (2); expected 'gener-
ated' (1)
correct('guidlines') => 'guideline' (2); expected 'guide-
lines' (1)
```

In this output we show the call to `correct` and the result (with the `NWORDS` count for the result in parentheses), and then the word expected by the test set (again with the count in parentheses). What this shows is that if you don't know that "econometric" is a word, you're not going to be able to correct "economtric". We could mitigate by adding more text to the training corpus, but then we also add words that might turn out to be the wrong answer. Note the last four lines above are inflections of words that do appear in the dictionary in other forms. So we might want a model that says it is okay to add "-ed" to a verb or "-s" to a noun.

The second potential source of error in the language model is bad probabilities: two words appear in the dictionary, but the wrong one appears more frequently. I must say that I couldn't find cases where this is the only fault; other problems seem much more serious.

We can simulate how much better we might do with a better language model by cheating on the tests: pretending that we have seen the correctly spelled word 1, 10, or more times. This simulates having more text (and just the right text) in the language model. The function `spelltest` has a parameter, `bias`, that does this. Here's what happens on the development and final test sets when we add more bias to the correctly-spelled words:

Bias	Dev	Test
0	74%	67%
1	74%	70%
10	76%	73%
100	82%	77%
1000	89%	80%

On both test sets we get significant gains, approaching 80-90%. This suggests that it is possible that if we had a good enough language model we might get to our accuracy goal. On the other hand, this is probably optimistic, because as we build a bigger language model we would also introduce words that are the wrong answer, which this method does not do.

Another way to deal with unknown words is to allow the result of `correct` to be a word we have not seen. For example, if the input is "electroencephalographicallz", a good correction would be to change the final "z" to an "y", even though "electroencephalographically" is not in our dictionary. We could achieve this with a language model based on components of words: perhaps on syllables or suffixes (such as "-ally"), but it is far easier to base it on sequences of characters: 2-, 3- and 4-letter sequences.

2. P(*w*|*c*), the error model. So far, the error model has been trivial: the smaller the edit distance, the smaller the error. This causes some problems, as the examples below show. First, some cases where `correct` returns a word at edit distance 1 when it should return one at edit distance 2:

```
correct('reciet') => 'recite' (5); expected 'receipt' (14)
correct('adres') => 'acres' (37); expected 'address' (77)
correct('rember') => 'member' (51); expected 'remember' (162)
correct('juse') => 'just' (768); expected 'juice' (6)
correct('accesing') => 'acceding' (2); expected 'assessing' (1)
```

Here, for example, the alteration of "d" to "c" to get from "adres" to "acres" should count more than the sum of the two changes "d" to "dd" and "s" to "ss".

Also, some cases where we choose the wrong word at the same edit distance:

```
correct('thay') => 'that' (12513); expected 'they' (4939)
correct('cleark') => 'clear' (234); expected 'clerk' (26)
correct('wer') => 'her' (5285); expected 'were' (4290)
correct('bonas') => 'bones' (263); expected 'bonus' (3)
correct('plesent') => 'present' (330); expected 'pleasant' (97)
```

The same type of lesson holds: In "thay", changing an "a" to an "e" should count as a smaller change than changing a "y" to a "t". How much smaller? It must be a least a factor of 2.5 to overcome the prior probability advantage of "that" over "they".

Clearly we could use a better model of the cost of edits. We could use our intuition to assign lower costs for doubling letters and changing a vowel to another vowel (as compared to an arbitrary letter change), but it seems better to gather data: to get a corpus of spelling errors, and count how likely it is to make each insertion, deletion, or alteration, given the surrounding characters. We need a lot of data to do this well. If we want to look at the change of one character for another, given a window of two characters on each side, that's 26^6, which is over 300 million characters. You'd want several examples of each, on average, so we need at least a billion characters of correction data; probably safer with at least 10 billion.

Note there is a connection between the language model and the error model. The current program has such a simple error model (all edit distance 1 words before any edit distance 2 words) that it handicaps the language model: we are afraid to add obscure words to the model, because if one of those obscure words happens to be edit distance 1 from an input word, then it will be chosen, even if there is a very common word at edit distance 2. With a better error model we can be more aggressive about adding obscure words to the dictionary. Here are some examples where the presence of obscure words in the dictionary hurts us:

```
correct('wonted') => 'wonted' (2); expected 'wanted' (214)
correct('planed') => 'planed' (2); expected 'planned' (16)
correct('forth') => 'forth' (83); expected 'fourth' (79)
correct('et') => 'et' (20); expected 'set' (325)
```

3. The enumeration of possible corrections, argmax_c. Our program enumerates all corrections within edit distance 2. In the development set, only 3 words out of 270 are beyond edit distance 2, but in the final test set, there were 23 out of 400. Here they are:

```
purple perpul
curtains courtens
minutes muinets

successful sucssuful
hierarchy heiarky
profession preffeson
weighted wagted
inefficient ineffect
availability avaiblity
thermawear thermawhere
nature natior
dissension desention
unnecessarily unessasarily
disappointing dissapoiting
acquaintances aquantences
thoughts thorts
criticism citisum
immediately imidatly
necessary necasery
necessary nessasary
necessary nessisary
unnecessary unessessay
night nite
minutes muiuets
assessing accesing
necessitates nessisitates
```

We could consider extending the model by allowing a limited set of edits at edit distance 3. For example, allowing only the insertion of a vowel next to another vowel, or the replacement of a vowel for another vowel, or replacing close consonants like "c" to "s" would handle almost all these cases.

4. There's actually a fourth (and best) way to improve: change the interface to `correct` to look at more context. So far,correct only looks at one word at a time. It turns out that in many cases it is difficult to make a decision based only on a single word. This is most obvious when there is a word that appears in the dictionary, but the test set says it should be corrected to another word anyway:

```
correct('where') => 'where' (123); expected 'were' (452)
correct('latter') => 'latter' (11); expected 'later' (116)
correct('advice') => 'advice' (64); expected 'advise' (20)
```

We can't possibly know that `correct('where')` should be "were" in at least one case, but should remain "where" in other cases. But if the query had been `correct('They where going')` then it seems likely that "where" should be corrected to "were".

The context of the surrounding words can help when there are obvious errors, but two or more good candidate corrections. Consider:

```
correct('hown') => 'how' (1316); expected 'shown' (114)
correct('ther') => 'the' (81031); expected 'their' (3956)
correct('quies') => 'quiet' (119); expected 'queries' (1)
correct('nation') => 'nation' (170); expected 'nature' (171)
correct('thear') => 'their' (3956); expected 'there' (4973)
correct('carrers') => 'carriers' (7); expected 'careers' (2)
```

Why should "thear" be corrected as "there" rather than "their"? It is difficult to tell by the single word alone, but if the query were `correct('There's no there thear')` it would be clear.

To build a model that looks at multiple words at a time, we will need a lot of data. Fortunately, Google has released adatabase of word counts for all sequences up to five words long, gathered from a corpus of a *trillion* words.

I believe that a spelling corrector that scores 90% accuracy will *need* to use the context of the surrounding words to make a choice. But we'll leave that for another day...

5. We could improve our accuracy scores by improving the training data and the test data. We grabbed a million words of text and assumed they were all spelled correctly; but it is very likely that the training data contains several errors. We could try to identify and fix those. Less daunting a task is to fix the test sets. I noticed at least three cases where the test set says our program got the wrong answer, but I believe the program's answer is better than the expected answer:

```
correct('aranging') => 'arranging' (20); expected 'arrangeing' (1)
correct('sumarys') => 'summary' (17); expected 'summarys' (1)
correct('aurgument') => 'argument' (33); expected 'auguments' (1)
```

We could also decide what dialect we are trying to train for. The following three errors are due to confusion about American versus British spelling (our training data contains both):

```
correct('humor') => 'humor' (17); expected 'humour' (5)
correct('oranisation') => 'organisation' (8); expected
'organization' (43)
correct('oranised') => 'organised' (11); expected 'organized' (70)
```

6. Finally, we could improve the implementation by making it much faster, without changing the results. We could re-implement in a compiled language rather than an interpreted one. We could have a lookup table that is specialized to strings rather than Python's general-purpose dict. We could cache the results of computations so that we don't have to repeat them multiple times. One word of advice: before attempting any speed optimizations, profile carefully to see where the time is actually going. ■

Peter Norvig is Director of Research at Google Inc. He is a Fellow of the AAAI and the ACM and co-author of Artificial Intelligence: A Modern Approach, the leading textbook in the field. Previously he was head of Computational Sciences at NASA and a faculty member at USC and Berkeley.

SLED DRIVER
FLYING THE WORLD'S FASTEST JET
BRIAN SHUL
CENTENNIAL OF FLIGHT LIMITED EDITION
No. 1
1903
2003
FROM FIRST TO FASTEST
SLED DRIVER

Sled Driver Giveaway Challenge Winner

Thank you to all who participated. We've picked Phil Jepsen (randomly among the correct entries) as the winner of the Sled Driver Giveaway Challenge. Here is his winning answer:

```
echo '1903, 2003, "Centennial Of
Flight"' | awk '{split($0, a, "")};
{for (i=1;i<=length(a);i++) {if
(a[i] ~/^[0-9]+$/) {s+=a[i]} else
if (a[i] ~/^[a-zA-Z][a-z[A-Z]*$/)
{c+=length(a[i])}}}; END {print "Sum
of 1+9+0+3 and 2+0+0+3 =", s}; {print
"Length of letters in \"Centennial Of
Flight\" =", c};'
```

Congratulation, Phil!

Five Principles for Choosing and Using Typefaces

By DAN MAYER

For many beginners, the task of picking fonts is a mystifying process. There seem to be endless choices—from normal, conventional-looking fonts to novelty candy cane fonts and bunny fonts—with no way of understanding the options, only never-ending lists of categories and recommendations. Selecting the right typeface is a mixture of firm rules and loose intuition, and takes years of experience to develop a feeling for. Here are five guidelines for picking and using fonts that I've developed in the course of using and teaching typography.

❶ Dress For The Occasion

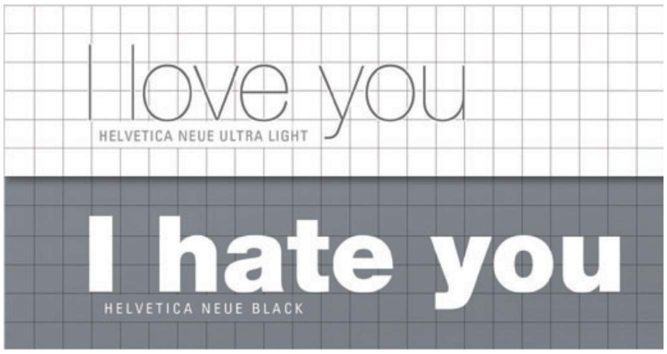
Many of my beginning students go about picking a font as though they were searching for new music to listen to: they assess the personality of each face and look for something unique and distinctive that expresses their particular aesthetic taste, perspective and personal history. This approach is problematic, because it places too much importance on individuality.

The most appropriate analogy for picking type.

For better or for worse, picking a typeface is more like getting dressed in the morning. Just as with clothing, there's a distinction between typefaces that are expressive and stylish versus those that are useful and appropriate to many situations, and our job is to try to find the right balance for the occasion. While appropriateness isn't a sexy concept, it's the acid test that should guide our choice of font.

My "favorite" piece of clothing is probably an outlandish pair of 70s flare bellbottoms that I bought at a thrift store, but the reality is that these don't make it out of my closet very often outside of Halloween. Every designer has a few favorite fonts like this — expressive personal favorites that we hold onto and wait for the perfect festive occasion to use. More often, I find myself putting on the same old pair of Levis morning after morning. It's not that I like these better than my cherished flares, exactly… I just seem to wind up wearing them most of the time.

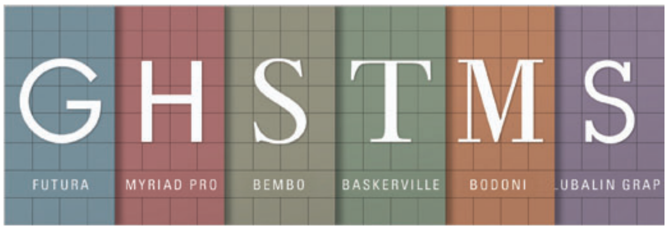
Every designer has a few workhorse typefaces that are like comfortable jeans: they go with everything, they seem to adapt to their surroundings and become more relaxed or more formal as the occasion calls for, and they just seem to come out of the closet day after day. Usually, these are faces that have a number of weights (Light, Regular, Bold, etc) and/or cuts (Italic, Condensed, etc). My particular safety blankets are: Myriad,Gotham, DIN, Akzidenz Grotesk and Interstate among the sans; Mercury, Electra and Perpetua among the serif faces.



A large type family like Helvetica Neue can be used to express a range of voices and emotions. Versatile and comfortable to work with, these faces are like a favorite pair of jeans for designers.

❷ Know Your Families: Grouping Fonts

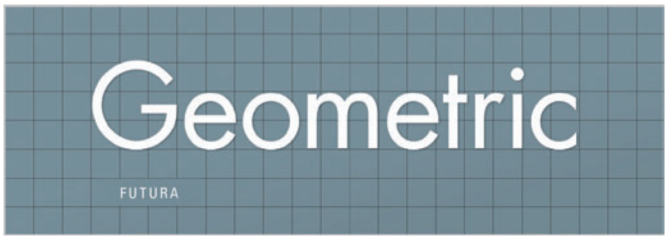
The clothing analogy gives us a good idea of what kind of closet we need to put together. The next challenge is to develop some kind of structure by which we can mentally categorize the different typefaces we run across.



Typefaces can be divided and subdivided into dozens of categories (Scotch Modern, anybody?), but we only really need to keep track of five groups to establish a working understanding of the majority of type being used in the present-day landscape.

The following list is not meant as a comprehensive classification of each and every category of type but rather as a manageable shorthand overview of key groups. Let's look at two major groups without serifs (serifs being the little feet at the ends of the letterforms), two with serifs, and one outlier (with big, boxey feet).

❶ Geometric Sans

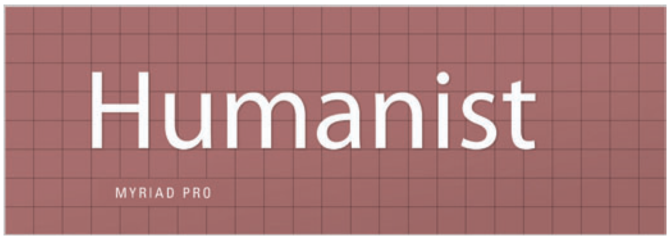


I'm actually combining three different groups here (Geometric, Realist and Grotesk), but there is enough in common between these groups that we can think of them as one entity for now. Geometric Sans-Serifs are those faces that are based on **strict geometric forms**. The individual letter forms of a Geometric Sans often have strokes that are all the same width and frequently evidence a kind of "less is more" minimalism in their design.

At their best, Geometric Sans are clear, objective, modern, universal; at their worst, cold, impersonal, boring. A classic Geometric Sans is like a beautifully designed airport: it's impressive, modern and useful, but we have to think twice about whether or not we'd like to live there.

Examples of Geometric/Realist/Grotesk Sans: Helvetica, Univers, Futura, Avant Garde, Akzidenz Grotesk, Franklin Gothic, Gotham.

❷ Humanist Sans

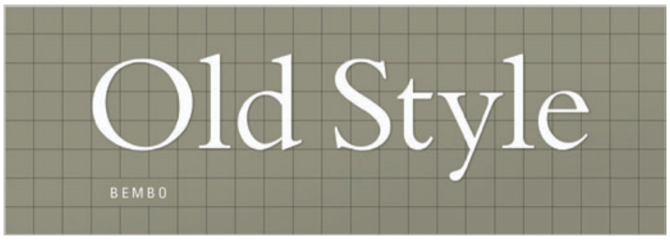


These are Sans faces that are derived from **handwriting** — as clean and modern as some of them may look, they still retain something inescapably human at their root. Compare the 't' in the image above to the 't' in 'Geometric' and note how much more detail and idiosyncrasy the Humanist 't' has.

This is the essence of the Humanist Sans: whereas Geometric Sans are typically designed to be as simple as possible, the letter forms of a Humanist font generally have more detail, less consistency, and frequently involve thinner and thicker stoke weights — after all they come from our handwriting, which is something individuated. At their best, Humanist Sans manage to have it both ways: modern yet human, clear yet empathetic. At their worst, they seem wishy-washy and fake, the hand servants of corporate insincerity.

Examples of Humanist Sans: Gill Sans, Frutiger, Myriad, Optima, Verdana.

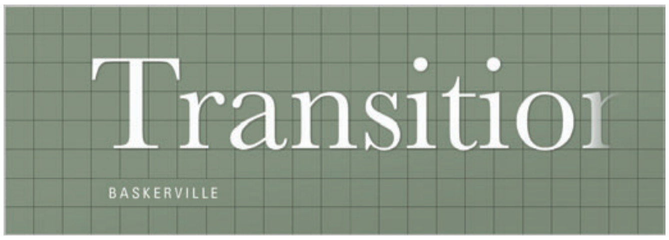
❸ Old Style

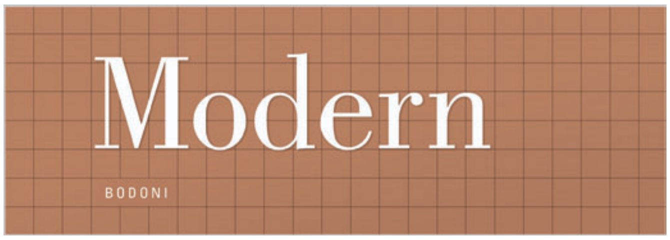


Also referred to as 'Venetian', these are our **oldest typefaces,** the result of centuries of incremental development of our calligraphic forms. Old Style faces are marked by little contrast between thick and thin (as the technical restrictions of the time didn't allow for it), and the curved letter forms tend to tilt to the left (just as calligraphy tilts). Old Style faces at their best are classic, traditional, readable and at their worst are… well, classic and traditional.

Examples of Old Style: Jenson, Bembo, Palatino, and — especially — Garamond, which was considered so perfect at the time of its creation that no one really tried much to improve on it for a century and a half.

❹ Transitional and Modern



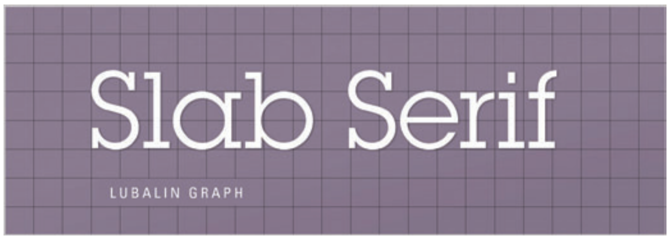


An outgrowth of Enlightenment thinking, Transitional (mid 18th Century) and Modern (late 18th century, not to be confused with mid 20th century modernism) typefaces emerged as type designers experimented with making their letterforms **more geometric, sharp and virtuosic** than the unassuming faces of the Old Style period. Transitional faces marked a modest advancement in this direction — although Baskerville, a quintessential Transitional typeface, appeared so sharp to onlookers that people believed it could hurt one's vision to look at it.

In carving Modernist punches, type designers indulged in a kind of virtuosic demonstration of contrasting thick and thin strokes — much of the development was spurred by a competition between two rival designers who cut similar faces, Bodoni and Didot. At their best, transitional and modern faces seem strong, stylish, dynamic. At their worst, they seem neither here nor there — too conspicuous and baroque to be classic, too stodgy to be truly modern.

Examples of transitional typefaces: Times New Roman, Baskerville.
Examples of Modern Serifs: Bodoni, Didot.

❺ Slab Serifs



Also known as 'Egyptian' (don't ask), the Slab Serif is a wild card that has come strongly back into vogue in recent years. Slab Serifs usually have strokes like those of sans faces (that is, simple forms with relatively little contrast between thick and thin) but with solid, rectangular shoes stuck on the end. Slab Serifs are an outlier in the sense that they convey **very specific — and yet often quite contradictory — associations**: sometimes the thinker, sometimes the tough guy; sometimes the bully, sometimes the nerd; sometimes the urban sophisticate, sometimes the cowboy.

They can convey a sense of authority, in the case of heavy versions like Rockwell, but they can also be quite friendly, as in the recent favorite Archer. Many slab serifs seem to express an urban character (such as Rockwell, Courier and Lubalin), but when applied in a different context (especially Clarendon) they strongly recall the American Frontier and the kind of rural, vernacular signage that appears in photos from this period. Slab Serifs are hard to generalize about as a group, but their distinctive blocky serifs function something like a pair of horn-rimmed glasses: they add a distinctive wrinkle to anything, but can easily become overly conspicuous in the wrong surroundings.

Examples of Slab Serifs: Clarendon, Rockwell, Courier, Lubalin Graph, Archer.

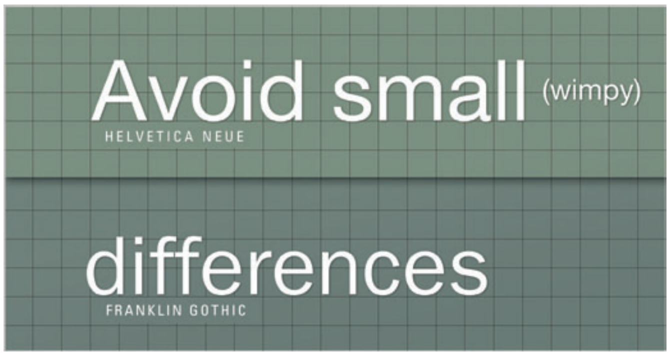
❸ Don't Be a Wimp: The Principle of Decisive Contrast

So, now that we know our families and some classic examples of each, we need to decide how to mix and match and — most importantly — whether to mix and match at all. Most of the time, one typeface will do, especially if it's one of our workhorses with many different weights that work together. If we reach a point where we want to add a second face to the mix, it's always good to observe this simple rule: **keep it exactly the same, or change it a lot** — avoid wimpy, incremental variations.

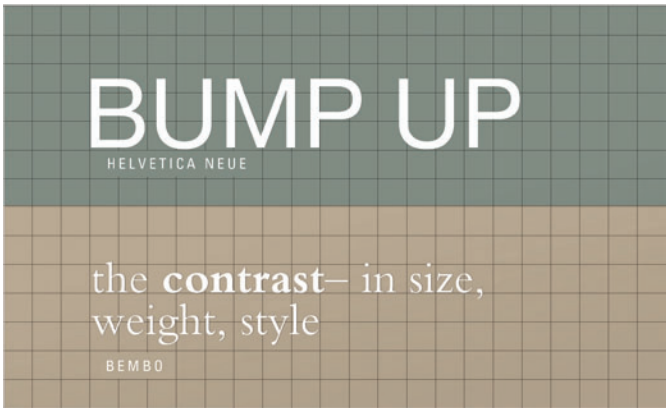
This is a general principle of design, and its official name is *correspondence and contrast*. The best way to view this rule in action is to take all the random coins you collected in your last trip through Europe and dump them out on a table together. If you put two identical coins next to each other, they look good together because they match (*correspondence*). On the other hand, if we put a dime next to one of those big copper coins we picked up somewhere in Central Europe, this also looks interesting because of the contrast between the two — they look sufficiently different.

What doesn't work so well is when put our dime next to a coin from another country that's almost the same size and color but slightly different. This creates an uneasy visual relationship because it poses a question, even if we barely register it in on a conscious level — our mind asks the question of whether these two are the same or not, and that process of asking and wondering distracts us from simply viewing.

When we combine multiple typefaces on a design, we want them to coexist comfortably — we don't want to distract the viewer with the question, are these the same or not? We can start by avoiding two different faces from within one of the five categories that we listed above all together — two geometric sans, say Franklin and Helvetica. While not exactly alike, these two are also not sufficiently different and therefore put our layout in that dreaded neither-here-nor-there place.

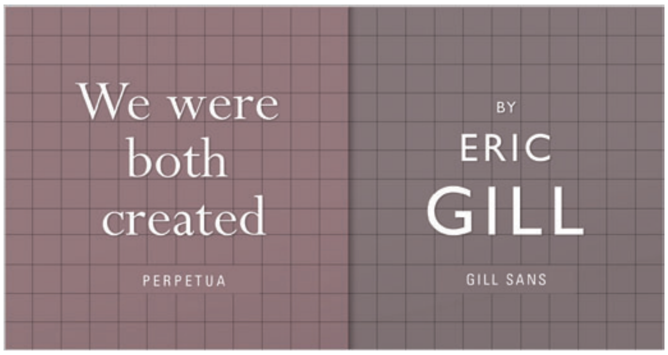


If we are going to throw another font into the pot along with Helvetica, much better if we use something like Bembo, a classic Old Style face. Centuries apart in age and light years apart in terms of inspiration, Helvetica and Bembo have enough contrast to comfortably share a page:



Unfortunately, it's not as simple as just picking fonts that are very, very different — placing our candy cane font next to, say, Garamond or Caslon does not guarantee us typographic harmony. Often, as in the above example of Helvetica and Bembo, there's no real explanation for why two faces complement each other — they just do.

But if we want some principle to guide our selection, it should be this: often, two typefaces work well together if they have one thing in common but are otherwise greatly different. This shared common aspect can be visual (similar x-height or stroke weight) or it can be chronological. Typefaces from the same period of time have a greater likelihood of working well together… and if they are by the same designer, all the better.

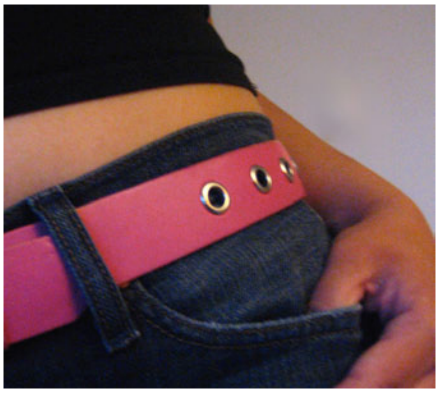


❹ A Little Can Go a Long Way

'Enough with all these conventional-looking fonts and rules!' you say. 'I need something for my rave flyer! And my Thai restaurant menu! And my Christmas cards!' What you're pointing out here is that all the faces I've discussed so far are 'body typefaces', meaning you could conceivably set a whole menu or newspaper with any of them; in the clothing analogy presented in part one, these are our everyday Levis. What of our Halloween flares?

Periodically, there's a need for a font that oozes with personality, whether that personality is warehouse party, Pad Thai or Santa Claus. And this need brings us into the vast wilderness of Display typefaces, which includes everything from Comic Sans to our candy-cane and bunny fonts. 'Display' is just another way of saying '**do not exceed recommended dosage**': applied sparingly to headlines, a display font can add a well-needed dash of flavor to a design, but it can quickly wear out its welcome if used too widely.

Time for another clothing analogy:



Betsey's outfit works because the pink belts acts as an accent and is offset by the down-to-earthiness of blue jeans. But if we get carried away and slather Betsey entirely in pink, she might wind up looking something like this:

Let's call this the Pink Belt Principle of Type: display faces with lots of personality are best used in small doses. If we apply our cool display type to every bit of text in our design, the aesthetic appeal of the type is quickly spent and — worse yet — our design becomes very hard to read.

❺ Rule Number Five Is 'There Are No Rules'

Really. Look hard enough and you will find a dazzling-looking menu set entirely in a hard-to-read display font. Or of two different Geometric Sans faces living happily together on a page (in fact, just this week I wound up trying this on a project and was surprised to find that it hit the spot). There are only conventions, no ironclad rules about how to use type, just as there are no rules about how we should dress in the morning. It's worth trying everything just to see what happens — even wearing your Halloween flares to your court date.

In Conclusion

Hopefully, these five principles will have given you some guidelines for how to select, apply and mix type — and, indeed, whether to mix it at all. In the end, picking typefaces requires a combination of understanding and intuition, and — as with any skill — demands practice. With all the different fonts we have access to nowadays, it's easy to forget that there's nothing like a classic typeface used well by somebody who knows how to use it.

Some of the best type advice I ever received came early on from my first typography teacher: pick one typeface you like and use it over and over for months to the exclusion of all others. While this kind of exercise can feel constraining at times, it can also serve as a useful reminder that the quantity of available choices in the internet age is no substitute for quality. ■

Dan Mayer's interest in graphic design began when he was five years old and visited a printing press on a 1979 episode of Sesame Street, and has been expanding ever since. A native of the US, he currently resides in Prague, where he works freelance and teaches courses in design history and theory at Prague College. His work and more examples of his writing can be found at *www.danmayer.com*.

Reprinted with permission of the original author.
First appeared in Smashing Magazine — an online magazine for professional designers and web developers [*hn.my/typefaces*].

Photo credit (in order of appearance): Samuuraijohnny [www.flickr.com/photos/samuraislice/], Betsssssy [www.flickr.com/photos/betsssssy/], Philip Leroyer [www.flickr.com/people/philippeleroyer/]

#badass

Rypple

Try it with your team: www.rypple.com/hacker

Dead-End Jobs: Are You Suffering From Stockholm Syndrome?

By CHAD FOWLER

HAVE YOU HEARD of Stockholm Syndrome? It's a name given to the condition wherein hostages develop positive feelings toward their captors despite being held in negative, unfavorable and even life-threatening conditions. Victims of Stockholm Syndrome will even inexplicably stay with their captors even when given the chance at freedom.

Hopefully nobody reading this is literally being held hostage right now. If you are, good luck!

For the rest of you, why might I suggest that you are suffering from Stockholm Syndrome? Because employment relationships can manifest themselves in this very way.

In the article, Love and Stockholm Syndrome: The Mystery of Loving an Abuser, Dr. Joseph Carver says that the following four situations serve as a foundation for the development of Stockholm Syndrome:

- *The presence of a perceived threat to one's physical or psychological survival and the belief that the abuser would carry out the threat.*
- *The presence of a perceived small kindness from the abuser to the victim*
- *Isolation from perspectives other than those of the abuser*
- *The perceived inability to escape the situation*

Looking back at my own career (and those of other extremely intelligent people I've met who are stagnating in oppressive companies) I have recognized that many (me, too, occasionally) have felt "stuck" for no obvious reason. Some people seem just plain crazy when you look at their skill sets, ability, and the low quality of work or environment they're willing to put up with.

So I contacted Joseph Carver to ask his opinion. Could this be Stockholm Syndrome? He agreed. In e-mail, he said "SS is most likely to develop when the employee feels trapped, perhaps by a high salary, fear of losing a career, or fear of humiliation." So let's look at his four conditions:

Perceived threat

Getting fired, being humiliated, not being a "top 20%" employee, not getting a raise. Employers wield a lot of perceived power over employees, especially for those in very traditional corporate jobs. The employer must be willing to carry out the threat to meet this condition.

Small kindness

Got a Christmas bonus once when you really needed it? Make a competitive salary? Great benefits? Get to work on a technology you don't think you'd be able to work on elsewhere? There ya go.

Isolation from other perspectives

Again, a big corporate environment is ripe for this kind of isolation. If you work for BigCo, you learn to do things The BigCo way. The company's organizational structure becomes a blueprint for your career progression. You start to lose sight of what industry pay and incentives look like since you have a homogeneous population to compare with. Unfortunately, from what I've seen even the best run companies create this kind of isolation of perspective and group-think.

Charismatic leaders are particularly capable of creating a culture vacuum around a cult of personality.

Perceived inability to escape

According to the Bureau of Labor statistics, American adults spend by far more time working than any other activity. That's a lot of your waking time being trapped in a routine. In a Stockholm Syndrome situation, the captor chips away at the self-esteem of the captive. So for most of our waking hours, those of us trapped in dead end jobs like these are exposed to environments which systematically destroy our self-confidence. Not only that, a persistent fear and feeling of failure makes it harder to actually explore the options for leaving the bad situation. The instinctive self-preservation reaction in this kind of situation is to work harder to try to avoid the perceived threat coming to fruition.

So, what if this describes your job? You owe it to yourself to find a way out. Hopefully recognizing the signs will show you that the real situation is far less grim than you might believe and that you have control over how you choose to spend the majority of your adult life.

I'm writing this for the many people I've met (and the countless I haven't) who are senselessly stuck in bad job situations. Please stop wasting your precious time. ■

Chad Fowler is an internationally known software developer, trainer, manager, speaker, and musician. He loves to program computers and, as part of his role as CTO of InfoEther, Inc., spends much of his time solving hard problems for customers in the Ruby language. He is co-organizer of RubyConf and RailsConf and author or co-author of a number of popular software books, including the recently released "The Passionate Programmer: Creating a Remarkable Career in Software Development".

Commentary

By SANDBOX (SandB0x)

DEAD END JOBS destroy transferable technical skills. I've witnessed miserable scenes. Many people are stuck maintaining large pieces of poorly written software. They forget how to actually program because their work involves very little development, and becomes all about knowing how the specific piece of software works (and the company's admin procedures) so that they can fight fires and make minor changes.

Escape involves gathering the confidence and the determination for self study, so that applying for another job is even a viable option. Skunk-works type projects at work are strongly recommended.

By JAKE VOYTKO (jakevoytko)

I JUST LEFT A job that afflicts Stockholm Syndrome on employees. Thankfully as a systems programmer in a research firm, I avoided the worst of it, but most people weren't so lucky. Here are some management behaviors that were effective:

Never thank people for finishing something on time, on budget, and to the project specifications. Instead, heap attention on those who finish in an all-nighter that ends in the final demo. Heroic measures are sometimes necessary and deserve reward, but being part of the process is a big red flag.

Never set expectations or milestones, just expect the project to be finished on the due date. This had an interesting effect on the work pace. Due to Parkinson's Law, individual workers finish days before the deadline, but that's not enough time to test integration. Major problems are discovered late, and everyone works ridiculous hours to fix it. Thank everyone for making it work at the last minute, rinse and repeat!

Tell employees to work weekends and nights for projects that could be unnecessary. Make these individual efforts to maximize the time one person wastes. When burning the candle on both ends, it's satisfying when you're done and the work was needed. After all, you took on the impossible, and here it is! But when days or weeks of your life are thrown away with a laugh, you would find another job if you actually had the time.

The Perceived Threats were the Bad Things that would happen if our demos failed. Funding lines would dry up, the company would be in trouble, etc. So everyone pitched together to keep the system going. Everyone became so focused that they stopped realizing that it could be done another way.

The Day MAME Saved My Ass

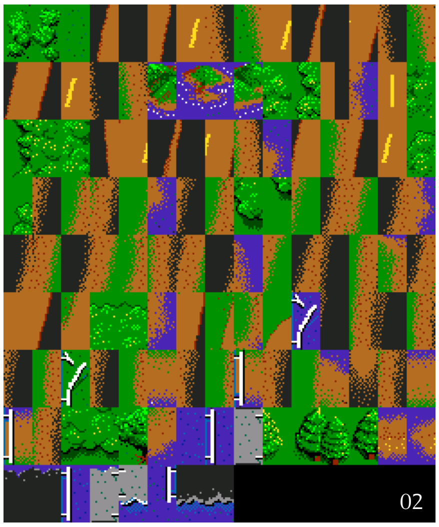
By MARK FELDMAN

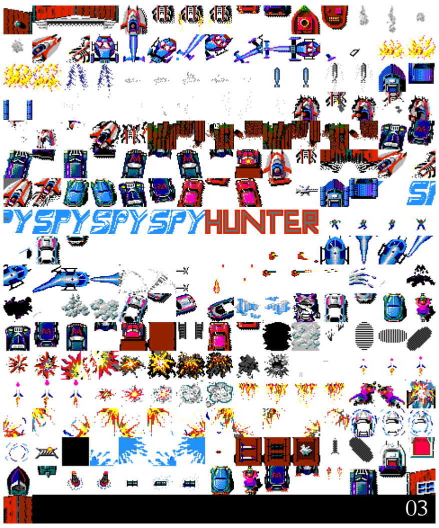
Ask any game developer and they'll tell you that publishers are the scum of the earth. It's never a question of "if" the publisher screws you, it's "when". During my 15 years as a developer, I have seen publishers pull every dirty trick imaginable, from telling the dev team of a certain AAA title to remove all the black kids from the game ("it hurts sales in Germany") to informing a small studio that they were only going to pay half what they owed for work already completed, and then only if the studio signs a legal waiver first (knowing full well that because of late payments the studio would be out of business long before it reached court). This story is not about publishers, but it is about the kinds of situations that publishers create, and the lengths that we developers are often forced to go to in order to clean up the messes they leave us with.

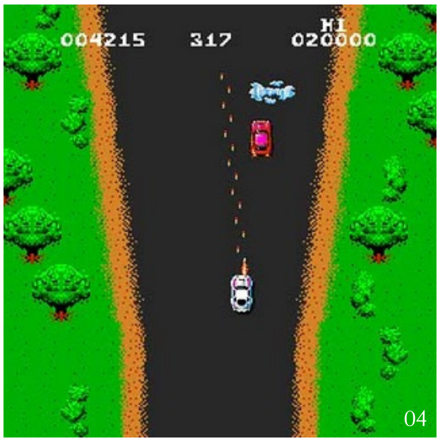
The project in question was a nightmare from day one. There were multiple studios involved, multiple contracts had been broken, and several teams had simply walked away and washed their hands of the whole thing. We had signed a contract to do some mobile ports of some old Midway arcade titles for a publisher who was subcontracting us on behalf of a much larger publisher. The contract with our publisher stated that they would obtain the original source code from Midway, hand it over to us at the start of the project, and we would deliver working alpha, beta, and release versions on certain dates.

We signed the contract and immediately got to work on our titles. As it turned out, there were many problems with this contract, but one of the more serious ones was that the publisher was missing the source code and assets for one title in particular: SpyHunter.
In the weeks that followed, the publisher assured us that they were around somewhere, and everything would be taken care of in due course. One week before alpha we heard back; they didn't have them and wouldn't be able to get them in time. SpyHunter had been included in the contract, because the suits at the top assumed they could deliver given the fact that they had already released a

01


02


03



04

1. SpyHunter.
2. Background tiles.
3. Sprites images.
4. Gameplay screenshot.

port to one of the major consoles. When we looked closer into this we discovered that the "port" team had actually written an emulator for that console (based on MAME btw) and simply used it to run a hacked version of the original ROM image, which they'd downloaded from a warez site.

Oh.

Running an emulator wasn't an option on a low-end mobile, but the publisher insisted that we had signed a contract and that with or without assets we were expected to deliver the alpha version on time, which at this point was a week later. If we didn't, not only would they withhold all payments due for work we'd already done, but they would also cancel all other projects which would have inevitably meant letting all our staff go. I had one week to somehow create a convincing version of SpyHunter without any of the source code or game assets. The way I saw it, there were four problems that needed to be solved: the AI, the graphics assets, the sound assets, and the map data.

AI wasn't too much of a problem; by simply looking at the game I was able to hack together some code in a few hours to roughly copy the behavior of the various objects in the game. It wasn't 100% perfect, but the publisher was more concerned with the visual look of the game. I don't think they ever played the original version much, so we got away with that one pretty easy.

Graphics were a bit more of a problem as there was no way our artist could have created replacement art assets in a few days. This was where MAME first came to the rescue. The version we had downloaded to play the game had a sprite page viewer that displayed all the tiles used by the game. They changed occasionally as the game progressed, but with some trial and error (and judicious use of the Prnt Scrn button) I managed to get a complete set of graphics for the port. They needed to be scaled down for our target platform, and this required some Photoshop magic to make sure that the tiles didn't bleed into each other, but the end result was perfect.

Originally, I thought that sound was going to be a real problem. Our target platform couldn't play music, but we still needed a way of getting the effects. MAME didn't have a sound effect exporter, so it looked like I might have to resort to trial and error poking around the ROM. Fortunately, it never came to that; a quick web search returned the page of a SpyHunter fan who had extracted the sound effects himself and put up WAV files on his web page. These weren't recordings, these were rips. Raw, ADPCM uncompressed WAV files, perfect digital reconstructions of the original assets. A quick check with our publisher's legal department confirmed that yes we were fully entitled to use these in our port. Personally, I wanted to give the guy a game credit, or at least send him a download code for a free copy of our game or something once it came out in return for all the work he saved us. The publisher refused. Their legal team was already writing up a cease-and-desist letter ordering him to remove the assets from his fan page.

And that left the map data. The publisher insisted that the map be a perfect replica of the original with all the right tiles in all the right places. Given enough time, I could have reverse engineered the ROM and reconstructed it that way, but with only a day or two left I had to come up with something fast. The idea I came up with was so utterly absurd that even I was surprised it worked so well. The first thing I did was write an application that searched for the MAME window and then sat in the background taking a constant stream of screenshots which it saved to disk. I then had the best game player in the office play SpyHunter for about 10 minutes using MAME's built-in cheats so that he never died. A second utility then post-processed these files and stitched them together using a simple pixel-match algorithm to create a continuous image. A third utility then scanned over this image and again applied a simple pixel-match against the tiles in the sprite page that we had exported earlier. What I was left with was a partially reconstructed version of the original SpyHunter map array. I say partially because many of the tiles were missing; if a car or explosion effect, etc was present on screen then the pixel-match algorithm would fail, and the tiles in that part of the map would be left blank. Most of the action happened on the road where it was easy to fill in missing tiles. For the rest, I had our guy play the MAME version a few more times, merging the data sets for each pass. When all was said and done there were only about 20 or so tiles missing from the entire game. At which point it was trivial to go through and add them manually using the MAME screen-shots as reference. The end-result was an 18x1538 array of tile indices representing a perfect reconstruction of the original map data.

Publishers would have people believe that MAME and the emulation scene is the root of all evil, that it promotes piracy and ultimately hurts the poor, starving developers slaving away on the game. Not only is this claim patently false, it ignores the fact that many developers use things like MAME, mod chips, and homebrew development utilities to help us overcome the day-to-day frustrations caused by the people behind the real problems in our industry. ■

Mark Feldman is a game industry veteran who has worked for studios both big and small all over the world. He currently works for a medium-sized developer that has successfully made the switch to self-publishing, thus removing itself from publisher dependence.

Commentary

By SUKOTTO (Sukotto)

Tip for next time: Make deliverables relative to the time you get the initial assets you were promised.

BigCo promises you the original source code + game assets.

Instead of promising Alpha/Beta/Release from the signing of the contract, promise X/Y/Z business days from delivery of those critical assets.

(Make sure you have a time-out clause in there too. "Assets will be delivered by BigCo within 60 days of the signing of this contract, or we will not develop that title and BigCo will pay a penalty of $X.")

Be careful to define what "assets" mean to you. Assume what you write will be read by a lowest-bidder consultant with active disincentives against showing initiative, and who can barely read (not stupid though).

PAYMO
timetracker

Paymo - Time Tracking & Invoicing

Wise tracking pays more!

www.paymo.biz

Get two months of free service by tweeting:

"I just learned about @Paymo time tracking & invoicing via @hackermonthly"

Winning Isn't Normal

By JASON SHEN

GROWING UP, I spent my summers holed up in the gym, training gymnastics for up to six hours a day. When I wasn't in the gym, I was doing math problems or practicing Chinese characters. Or preparing for SAT's (did 10 full tests one summer). Or reading personal development books, like 7 Habits for Highly Effective People, and writing personal mission statements. That wasn't normal.

In high school, there were days where I'd:

- Wake up at 7 am
- Go to school 'til 3 pm
- Stay after class to work with my high school gymnastics team till 4:30pm
- Drive to my "real" gymnastics practice at 5pm
- Do serious and intense training until 9-9:30pm
- Get home at 10pm
- Shower, eat dinner, and START doing my homework at 11pm.

That wasn't normal.

Sometimes I wish I had a more normal life growing up. I wanted desperately to watch more TV shows, play video games, and perhaps even get a girlfriend somehow. I wanted to fit in, you know, like a normal kid.

But then I realized that there were other things about my life that weren't normal. Making the Jr National Team, being named Boston Globe Gymnast of the Year 3 times in a row, getting a 1580 out of 1600 on my first (and only) taking of the SATs, and being selected as the graduation speaker for a 2000+ student high school. None of that was normal either.

If you want to win or succeed in something — you've got to be willing go against the grain. The truth is winners do what losers won't. World champion climber Patxi Usobiaga goes months without a single off day. What kinds of unreasonable, abnormal, and irrational things are YOU doing to ensure that you get results that blow people away?

> *Remember one thing: winning isn't normal. That doesn't mean there is anything wrong with winning. It just isn't the norm. It is highly unusual, so it requires unusual action. In order to win, you must do extraordinary things.*

(Excerpted from Winning Isn't Normal by Dr. Keith Bell)

You look at people who are extremely successful and I can almost guarantee there is at least something very weird or different about them. They have attitudes, habits, ideas and tendencies that are very abnormal. And that makes total sense. Because winning isn't normal. ■

Jason Shen is Customer Scout at isocket and a tech startup in the SF Bay Area. He did his undergrad and masters at Stanford where he was captain of the NCAA championship-winning men's gymnastics team. Jason loves new projects, sharing insights, and going on adventures.

Reprinted with permission of the original author.
First appeared in *hn.my/winning*.

Commentary

By ALLEN FREEMAN (knieveltech)

Reminds me of a singular experience I had several years ago. I'd been driving pizza delivery and in general coasting through life after dropping out of college. I had wanted to be a programmer when I grew up, but a personality conflict between myself and the head of the CS department at school convinced me to hit the bricks.

Anyway, I was doing a delivery to a local apartment complex on a windy day and on my way back to the car this little pink piece of paper blew across the parking lot and fetched up next to my shoe. I picked it up and inspected it. It was printed on both sides; one side had some kind of spam advertisement for carpet cleaning or real estate or something, but the other side caught my eye. All it said was "If you want something you've never had, you have to do something you've never done".

While it would make a great narrative to say I went back and quit my dead end job immediately and dedicated myself to getting my shit together, it was another year before I started putting my life in order. But ever since that day, whenever I felt like I was in a rut or whatever, I'd remember that little piece of paper.

Eventually, I got my shit together, educated myself on a couple of programming languages, and joined the workforce as a developer, married, and bought a house. While I don't think that piece of paper is solely responsible for my successes, the idea "If you want something you've never had, you have to do something you've never done" has been kicking around in my head ever since. And I'm sure it's colored at least some of the choices I've made since that day.

By DREW HAVEN (Periodic)

I like the point that "winning" requires abnormal behavior, but I don't like the phrasing as "winning". Winning implies a competition, when many worth-while things in life aren't a competition and often simply passing a personal bar will suffice.

It is a good point that doing extraordinary things requires extraordinary devotion, but the focus on overworking and beating an opposition bothers me.

I'd like to think that accomplishing extraordinary things requires an extraordinary goal and extraordinary passion. You need to have a goal to direct your energy towards, and you need the passion to pursue that goal even when it feels like it might be out of reach.

I think the real message of this article is just that accomplishing extraordinary things requires a lot of hard work, persistence, and patience. I think it misses the love and passion that is required to do that long-term.

HACKER JOBS

Full Stack Rails Developer

Howcast Media *(http://www.howcast.com)*
San Francisco
Your first project will be a semantic content syndication product that we are scaling to serve millions of page views per day. The project leverages bleeding-edge technologies to deliver relevant content to partner sites. Performance is king, so if you don't know it already, you'll be learning optimization in JavaScript, Ruby, and Tokyo Tyrant.
To Apply: All applications receive a response.
http://captainrecruiter.com/jobs/71/applications/new

Amazing Software Engineer Generalist

Rapleaf *(http://www.rapleaf.com)*
San Francisco
Rapleaf is seeking amazing engineers to help us reach our ambitious vision of creating a more personalized world. We're a technology company at heart — we solve complex, data-intensive problems with innovative solutions.
Check out our API and play around with your own application. We'd love to hear your feedback: *rapleaf.com/developers*
To Apply: To learn more, please visit us at *rapleaf.com/careers*

Senior Developer

youDevise, Ltd. *(https://dev.youdevise.com)*
London, England
60-person agile financial software company in London committed to learning and quality (dojos, TDD, continuous integration, exploratory testing). Under 10 revenue-affecting production bugs last year. Release every 2 weeks. Mainly Java, also Groovy, Scala; no prior knowledge of any language needed.
To Apply: Send CV to *jobs@youdevise.com*.

Front-end and Back-end Engineers

Meetup *(http://www.meetup.com)*
New York
Meetup thinks the world is a better place when groups of people meetup locally, in person, around a common interest. We're reinventing how this is done, but we can't do it alone! We value iterating/launching quickly, pragmatism, and long walks on the beach.
To Apply: *http://meetup.com/jobs*

Dream. Design. Print.

MagCloud, the revolutionary new self-publishing web service by HP, is changing the way ideas, stories, and images find their way into peoples' hands in a printed magazine format.

HP MagCloud capitalizes on the digital revolution, creating a web-based marketplace where traditional media companies, upstart magazine publishers, students, photographers, designers, and businesses can affordably turn their targeted content into print and digital magazine formats.

Simply upload a PDF of your content, set your selling price, and HP MagCloud takes care of the rest—processing payments, printing magazines on demand, and shipping orders to locations around the world. All magazine formatted publications are printed to order using HP Indigo technology, so they not only look fantastic but there's no waste or overruns, reducing the impact on the environment.

Become part of the future of magazine publishing today at www.magcloud.com.

25% Off the First Issue You Publish

Enter promo code **HACKER** when you set your magazine price during the publishing process.

Coupon code valid through February 28, 2011.
Please contact promo@magcloud.com with any questions.

MAGCLOUD