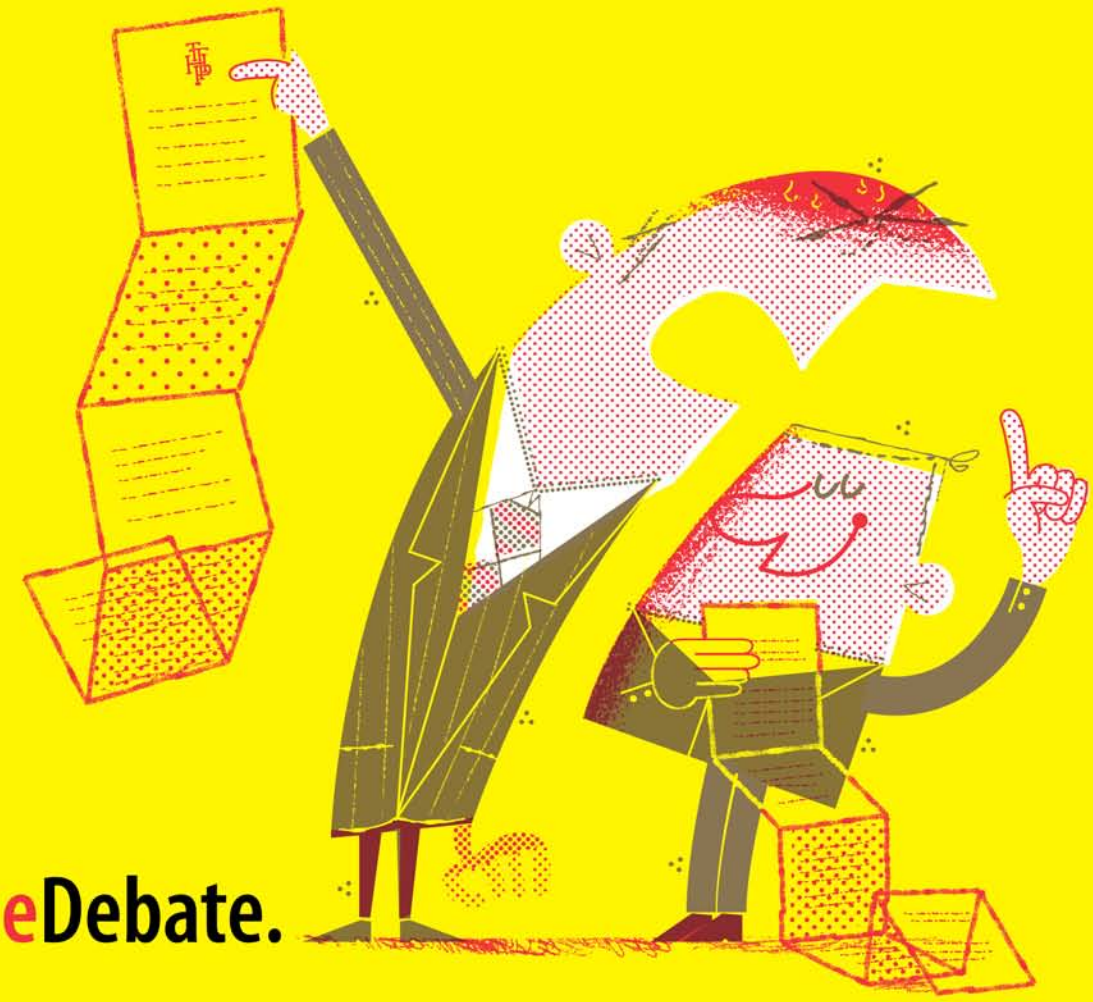


TheDebate.

HACKERMONTHLY

Special Issue
November 2010

Curator
Lim Cheng Soon

Proofreader
Ricky de Laveaga

Illustrator
Marc Aspinall

Printer
MagCloud

E-book Conversion
Fifobooks.com

Contributors

ARTICLES
Antonio Garcia-Martinez
Kelly Sutton
Robert C. Martin
Shaneal Manek
Mike Taylor
Yehuda Katz
Gary Bernhardt

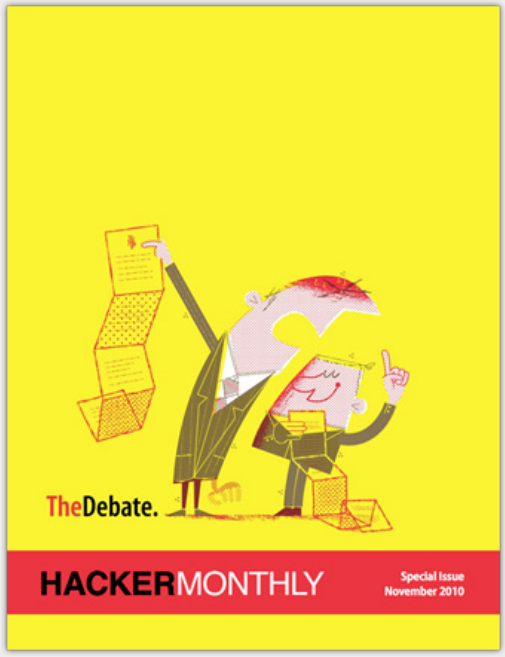
COMMENTARIES
Ben Tilly
Ardit Bajraktari
Miles Egan
Mahmud Mohamed
Peter Norvig
Matt Brubeck
Brandon Smietana
Paul Graham

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 Illustration: Marc Aspinall

Contents

New York Will Always Be A Tech Backwater

I don't care what Chris Dixon or Ron Conway or Paul Graham say

By ANTONIO GARCIA-MARTINEZ

Last Tuesday, legendary tech investor Ron Conway addressed the glop-eating masses at Y Combinator during our usual Tuesday illustrious-speaker dinner. The question was asked about the New York tech scene, and it's relative strength vis-à-vis Silicon Valley. Paul Graham took up the question with Techcrunch TV recently, as a follow-on to Conway's remarks. Chris Dixon, a respected New York-based VC, has also chimed in on the tech renaissance going on there.

They're all wrong.

New York will never be more than a tech sideshow.

Thinking the New York tech scene will ever equal Silicon Valley is as foolish as thinking San Francisco's puny theater district will one day take on Broadway. Both Silicon Valley and Broadway are unique products of the cities that spawned them, and every attempt to create a Silicon Alley/Silicon Sentier/Skolkovo/whatever in various parts of the world have failed. So far, no one's managed to do it, and New York sure as hell won't either.

The hero with a couple of faces

As Matt Mireles incisively points out in his related blog post, the mythology in New York is all wrong for startups.

Let's face it, young ambitious men have two goals in life: getting laid, and impressing other young ambitious men. You do neither in New York by saying you're starting a startup. That slinky young thing you're chatting up at Schiller's turns around to the investment banker next to her when you drop that bomb.

In the Bay Area, you drive through Atherton or Woodside and see the mansions that Netscape, Apple, and Oracle built. On the Upper East Side you see houses built thanks to the depredations of previous generations, and owned by the predators of today (probably their children).

In the Bay Area, new money is better than old. In New York, it's precisely the opposite. The mythology is all wrong.

$2495 for a 500 sq. ft. one bedroom apartment.

There, that's how much my first apartment in New York cost (in 2005).

Living in New York, you hemorrhage money, and don't see much in return. My career salary high-water mark is still working as a quant on Goldman's credit desk, and I lived worse, from a quality-of-life perspective, than I did as a Berkeley graduate student. 'Ramen' money in New York is enough to support three families, and then some, elsewhere. If YCombinator existed in New York, they'd have to dish 5x more than their already slim initial funding to keep new startups in Cheetos for three months.

Basically, startups flourish in the Bay Area the same reason the homeless do: decent weather, relatively cheap living, and no stigma attached to your lifestyle.

The cathedral and the brothel

Every yuppie I knew in New York worked as either a Wall Street guy, a lawyer, or an agent of some sort. Basically, there were all subtly screwing someone else for a living.

As an academic exile, my passport to this foreign world was my then live-in girlfriend, an embodiment of her socio-economic cohort: Bryn Mawr School for Girls, followed by Harvard, followed by med school. This was a person who could open the Sunday Styles weddings section, instantly identify a half-dozen couples, and rattle off the juicy gossip dating back to their time at Eliot House.

At cocktail parties with these people, the "ambitious ass-kickers" Paul Graham thinks will save the New York tech scene, the second question you're asked is inevitably what do you do? And so begins the not-so-subtle binning of you into your social echelon, more ritualistic and damning than any Japanese business card exchange ceremony:

+2 for working at Goldman Sachs
-1 for being a quant rather than a banker or trader
-1 for living on the Lower East Side
-2 for not being Ivy League
+/- 1 for being Gentile (depends on the cocktail party).

And you're socially in the red at that point. The rest of the conversation is as vacuous as interstellar space.

These people aren't builders, they're hustlers. And hustlers don't have the patience or skill to create the next Google or Facebook.

Open vs. closed source

New York's entire economy is based on monopolies of information. Wall Street banks make a mint trading because they have inside information on the market flows of the products they trade. Literary agents arbitrage scarce access to book publishers against a mass of hopeful authors. Real estate brokers (and these are brokers on rental properties, not properties for sale) routinely make a 15% commission when you sign a lease, pocketing a good two-months salary (read, upwards of $5000) for the privilege of telling you where there's an apartment free.

In New York, those monopolies go unchallenged.

In San Francisco, people don't pay two months' rent to a real estate pimp: they create Craigslist and make the pimp obsolete.

The intellectual candle-power isn't there

Harvard and MIT anchor Boston's startup scene, and have midwifed countless startups. Berkeley and Stanford were the birthplaces of everything from BSD Unix to Google.

New York has no comparable sources of intellectual firepower. NYU is an arts school. Their only world class science is the Courant Institute and its applied math program, which serves as a feeder school for Wall Street. Columbia is not a top-notch engineering school, and anyhow, it's way the hell up and gone in Harlem, and no one who isn't a student or faculty ever goes up there.

No place for Trotsky to sit down

One of the biggest shocks upon moving to New York was realizing it had no cafés. You can't have startups or revolutionary political movements without cozy cafés to dawdle, work, and plot in. Every day I step into the Red Rock Café in Mountain View, I see 2-3 startup founders I know, see about half a dozen hackers working on something on their Macs, or overhear some entrepreneur's pitch to an investor. Every day. Assuming you teleported all those people to New York tomorrow, the system would fall apart, as they'd have nowhere to meet.

Katz's pastrami is the only thing I miss

As a random but illustrative tangent, the food culture in NY vs. SF explains much of the attitude toward work and money as well.

The reality is, the food culture in New York mostly sucks. Sure, people there know how to go to Nobu and drop $300 on sushi, and every headliner chef needs to have some New York outpost, but most New Yorkers couldn't fry an egg if their lives depended on it (plus, most don't even have decent-sized kitchens).

In San Francisco just about everyone I know is an über-foodie. Over plates of home-cooked and home-grown asparagus, I've had endless, meandering conversations about heirloom tomato gardening or where on Twin Peaks to find the blackberry bushes. My ex-girlfriend keeps a backyard chicken farm, in posh Rockridge. People here go abalone diving in Bodega Bay. There's a herd of goats in a vacant lot in West Oakland I drive by, kept by an urban farming hippie. Most of the veggies I eat come from our backyard garden. Even the skeeziest convenience store in Daly City or Oakland has a drinkable collection of California wines on offer.

On the flip side, New Yorkers don't know anything about actual food. They know how to queue for two hours at the fashionable brunch spot they read about in New York magazine, and then opine haughtily about whether the hollandaise sauce on the Eggs Benedict compares to Balthazar's or not. In three years of living in New York, I never ate someone else's home-cooked food even once.

The lesson when it comes to tech is this: New Yorkers like bling. They like the establishment. They go Gucci and let you know it.

San Franciscans are more subversive: they get obsessed with creme brulée, quit their jobs, sell their obsession from a cart, tweet about where the creme brulée cart will be next (12,000 followers and counting), and create a whole new food paradigm: the socially-networked food cart.

The latter is a bootstrapped startup culture at work. It permeates everything in SF life, including the food. And it's why SF will dominate tech for the foreseeable future.

Happiness is a warm Sawzall

Another tangential but illustrative anecdote: Manhattan didn't get a Home Depot, or any sort of proper hardware store, until 2004 (!). The boys at Home Depot know their market though. The place is mostly indoor gardening supplies and little home toolkits to tighten that loose door hinge that keeps popping out. So, if it's the 18V DeWalt Sawzall that cuts through quarter-inch steel rods like a warm knife does butter that you want, well, then, like the signs on the BQE say, fuhgeddaboutit.

The Bay Area, by contrast, is a hacker's paradise. I'm fairly sure that between the big Oakland Home Depot and the geek paradise of Fry's Electronics in Palo Alto, I and a band of hardy souls could re-build all of 21st century human technological life on some barren island if need be. Good luck doing that with what you find on Fifth Avenue.

And that's precisely what's wrong with New York: it's filled with hyper-stressed, aggressive, social climbers who are actually kind of effete and helpless at the end of the day, and probably need to outsource their software development, because they're not, like, technical and all that. Except there's one problem….there aren't that many hackers in New York, and the few there are (I know because I used to be one of them) won't leave their $300,000 jobs on Wall Street to work on your hopelessly risky idea.

Which brings us to the other reason why New York will never be Silicon Valley…

Greed is God

While that odd mélange of Las Vegas, the Mafia and the Marines that we call Wall Street has taken a bit of a beating of late, rise again it will. And when it does, the tech scene in New York will evaporate like a puddle of water in the desert.

Time for the full disclosure: I spent three years on Goldman Sachs' credit trading desk as a pricing quant, which is what brought me to New York. The job paid well. The hours and stress, no worse than a startup's. The social vindication of what I was doing, absolute and immediate.

When the credit markets started looking dicey at the beginning of 2008 and I told my GS deskmates that I was moving to California to join a startup, they looked at me as if I had just proposed shaving my head and joining a Buddhist monastery in Burma. It was complete and total incomprehension. And these were the quants, most of them Ph.D.'s, the geekiest Wall Street gets. Most of the sales and trading guys probably couldn't find California on a map.

None of those Goldman Sachs quants, most of whom were precisely in the Spolskyian "smart/get things done" category that you'd want in a startup, really knew about or understood the startup scene, and how you could get just as wealthy with a startup, having lots more fun along the way, than warming a seat on a Wall Street trading floor. You, potential employer, will have to sell that person not just on your startup, but on startups in general. And that is a hard sale indeed. You'll only do it if that quant has lost his seat on the trading floor. That's been the case for some recently (including your faithful correspondent), but last I heard, Goldman is hiring again. So best of luck to you, aspiring New York entrepreneur.

Money talks, but bullshit walks

Since I suspect this post may get a flame or two from some diehard New Yorkers, I'll lay down this gauntlet in the face of regional jingoism:

I promise to wear one of those ridiculous "I <heart> NY" shirts you buy for $3 from the Nigerians in Times Square for an entire month if the total amount of New York-based startup funding, as reported in Crunchbase, exceeds that of Bay Area-based startups in any financial quarter during the next five years.

So…bring it, New York. 'Cause I say the hippies from California will continue to eat your lox. ■

Antonio is CEO and co-founder of AdGrok, a VC-backed startup founded with the express purpose of helping small businesses do online marketing. Before AdGrok, Antonio was a research scientist at a Bay Area digital marketing agency, writing bidding algorithms for online advertising exchanges. Prior to marketing, he was a pricing quant on Goldman Sachs' credit and equity trading floors, modeling credit-default swaps and various other weapons of financial destruction. He arrived on Wall Street as a young, naive physics Ph.D. student, and regrets the day he ever read Liar's Poker. When not grokking, he tries to play squash and sail his 26-foot sloop Moksha.

Reprinted with permission of the original author.
First appeared in *http://hn.my/nybackwater/*.

Commentary

By BEN TILLY

The point I consider most important is only alluded to. In NY people try to maintain monopolies of information. One place that applies to is a very restrictive work for hire doctrine. If you have a job in NYC, by default everything you create belongs to your employer. Even if it was done at home on your equipment. In California that particular right is one that remains yours and cannot be lost (unless what you do relates to your employer's business).

The result is that in California it is common for people to have a day job and work on the side on a startup. You almost never see the side startup in NY. Furthermore if you do see it, those startups can be squashed in an instant when the employer finds out about it. I've seen it happen. Not pretty. (The same thing can happen, and does from time to time, with open source projects. However it happens less often because there are fewer cases where someone wants to leave their day job for their open source project than cases where they want to leave their day job for their startup.)

That nasty little dynamic is a constant hidden drag on NY. Nobody knows how many good ideas got squashed and never saw the light of day…

Choosing New York over San Francisco

By KELLY SUTTON

I moved to New York two months ago to start full-time at blip.tv. I have also been fortunate enough to live in San Francisco and New York before (along with Seattle, Los Angeles, Düsseldorf and Berlin). Given that there's a constant surplus of work to be done and not enough engineers to do it, it's realistic to assume that I could have worked in any city I wanted in just about any industry (as long as I was designing software or pounding code in some capacity). Given that the Bay Area is so gung-ho on everything silicon, why did I choose New York City over Silicon Valley?

In short, New York City is more interesting and – I believe – better suited for the startup and a young guy's lifestyle. While I was not one of the first employees at blip.tv, the startup lifestyle seems to be pretty standard up until about 40 people. So when Antonio railed on my chosen city for 1,000+ words, it's difficult to not have a small existential crisis. Did I make the right choice? What if he's right?

San Francisco is fun, don't get me wrong. Compared to New York, it's boring. I have trouble even thinking about living in an apartment in Mountain View, Cupertino or Palo Alto. Those places are socially dead. Getting out into the real world is an important thing to do once in awhile. Talking tech over microbrews with other startup founders doesn't count. Most of the products and companies coming out of the Bay these days are… a little too optimistic.

Every once in awhile, you see a Google emerge from the Valley. But for every Google emerging from the Valley, there are ten thousand equally ambitious startups that fail. Some of them fail catastrophically. Bay Area startups are much bigger gambles than many NYC startups. Given the lifestyle in the City, products are much closer to the pavement and are a solution to a real-world problem from Day 1. Not some social network plaything.

I've got plenty of normal friends in New York. When I'm giddy about an idea, pitching it to someone that doesn't spend 8+ hours per day on a computer can bring me back down to earth. The social component of a real city with museums, clubs, venues, pubs, bars and barcades is important. Is housing more expensive? Yes. Am I paying for a car? No. Am I saving a decent amount of money by not worrying about car payments, gas, insurance and maintenance? Very much so. New York really isn't that much more expensive that the Valley, SF or LA if you compare them correctly.

I've never experienced the ibanker elitism that Antonio spends so much time lamenting over. That's Wall Street. That's part of New York, but that is not New York in its entirety. I've picked up that much in the two months I've been here. Many people are genuinely interested in you when you say that you work for a startup. And besides, if you're expecting to have some sort of social status because you're a professional gambler startup employee, I'm guessing you're in it for the wrong reasons. But what do I know? I'm only 23. ■

Kelly Sutton is a software engineer at blip.tv. Currently living in New York, Kelly has lived in Seattle, San Francisco, Berlin and Los Angeles. In his spare time he runs the sites HackCollege and Cult of Less.

Reprinted with permission of the original author.
First appeared in *http://hn.my/nyoversf/*.

Commentary

By ARDIT BAJRAKTARI

I think one major argument that he didn't mention is that NYC beats SF hands down in the "getting laid" department.

If you are young and your idea of having a good time is going out, partying until 4am, and getting laid often, then you should stay in NYC.

If you a super hipster, social outcast and/or into weird stuff, then come to SF. Your friends are already here.

BREADPIG SEARCHED THE GALAXIES FOR THE AWESOMEST SAUCE

...and found this 12 oz bottle of hot garlicky goodness. Unicorn tested, TIE-fighter Darwin approved.

Breadpig donates all of its non-sustainable profits to organizations and individuals doing great things for the world. We spend the rest of our time discovering and promoting fascinating technology, hacks, and ideas from all over the world that inspire and impress us. Find out more at www.breadpig.com.

GET AWESOMESAUCE AT STORE.XKCD.COM/BREADPIG OR FROM OUR FRIENDS AT THINKGEEK.

Why Clojure?

By ROBERT C. MARTIN

I HAVE RECENTLY BECOME quite an enthusiast for the language Clojure. But why? Why would someone who has spent the last 30 years programming in C, C++, Java, C#, and Ruby suddenly become enamored with a language that has roots that go back to 1957, i.e. Lisp?

During my first few decades as a professional programmer, I never learned Lisp. I had heard of it, of course; though mostly in derisive terms. People sneered about it with names like "Lots of InSignificant Parentheses." So my view was not particularly favorable.

A few years ago, someone suggested that I learn Lisp by reading a book entitled: "The Structure and Interpretation of Computer Programs." So I went to Amazon and ordered a copy from the used books section. It arrived a week or so later, and then sat on my "to read" stack for a couple of years.

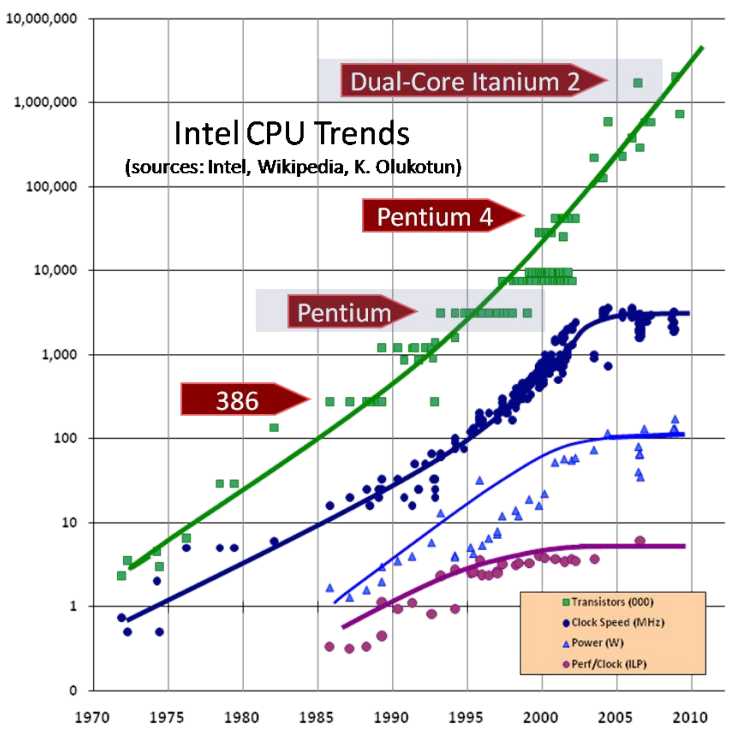
I started reading it about two years ago; and it changed everything I had previously felt and believed about Lisp. It also changed a great deal of what I felt and believed about programming in general. In short, the book was startling.

SICP is a literary masterpiece. It's not often that you can say that a technical book is a page-turner, but that's just what I found SICP to be. The book moves from topic to topic with rare ease and clarity, but more importantly it moves with purpose and mission. As you read it, you can feel the authors slowly building a tension towards a climax. The chapters fly by as you read about data structures, algorithms, message passing, first-class procedures, and so much else. Each concept leads inevitably to the next. Each chapter adds to the ever building tension. By time you are half-way through the book, the sense that something important is about to change becomes palpable.

And then something important changes! Something you had not anticipated. Something you should have guessed, but did not. On page 216 they introduce a concept so familiar that most programming books start with it. On page 216 they prove to you that you've had some wrong ideas about programming all along. On page two hundred and sixteen, after talking about algorithms, data structures, recursion, iteration, trees, high-order procedures, scoping, local variables, data abstraction, closures, message-passing, and a plethora of other topics – after all that, they introduce assignment!

And with that elegant coup-de-grace (which is not the last in this book!), they vanquish the concept that programming is about manipulating state. With that one stroke, they force you to look back on all you had done in the previous pages in a new and enlightened way – a functional way.

> “Moving functional programs to massively parallel system will be easier than moving non-functional programs.”



Moore’s Law

Why is functional programming important? Because Moore’s law has started to falter. Not the part of the law that predicts that the number of transistors on a chip doubles every two years. Thankfully, that part of the law seems to still be in effect. The part that faltered is the part that says the speed of computers doubles every two years.

What this means is that our computers can still get faster, but only if we put multiple CPUs on a chip. This is why we’ve seen all these multi-core processors showing up. And that means that programs that need greater speed will have to be able to take advantage of the multiple cores.

If you’ve ever written multi-threaded code, the thought of eight, sixteen, thirty-two, or even more processors running your program should fill you with dread. Writing multi-threaded code correctly is hard! But why is it so hard? Because it is hard to manage the state of variables when more than one CPU has access to them.

And this is where functional programming comes in. Functional programming, of the kind shown in SICP, is a way to write code that does not manage the state of variables, and could therefore be partitioned to run in parallel on as many processors as you like – at least in theory. In practice it might not be quite that trivial; but one thing is certain. Moving functional programs to massively parallel system will be easier than moving non-functional programs.

Why Clojure?

So why is Clojure the best option for a functional language? After all, there are lots of functional languages out there. Some are old, like Haskell, and Erlang. Some are new like Scala and F#. Why is Clojure the language that has everybody so fired up? Here are just a few reasons.

- Clojure is Lisp. And Lisp is a functional, simple, well-known, elegant language. The syntax is almost laughably terse. This is in contrast to languages like F# and Scala which have a complexity and "quirkiness" reminiscent of C++.
- Clojure is Java. Clojure sits on top of the Java stack, and has the ability to inter-operate with Java with extreme ease. Java programs can call Clojure, and Clojure can call Java. You can write Clojure code that derives from Java classes and overrides Java methods. In short, if you can do it in Java, you can do it in Clojure. What's more there is a Clojure port for the CLR! So Clojure may be the only functional language that inter-operates well with both major VMs.
- Clojure implements Software Transactional Memory which means that any time a Clojure programmer want's to change the state of a variable, they must do so using the same kind of transaction management as they would use for a database. This enforces the functional paradigm to a degree that few other functional languages do. The STM facilities of Clojure are elegant and simple, just like the rest of the language. They do not intrude where they aren't needed, and they are simple to employ where state must be changed.
- Clojure is fast. Data structures in functional languages are immutable. For example, you can't add an item to a list, instead you create a copy of the list with the new item added. This copying could obviously slow things down a lot. Clojure manages complex immutable data structures using a sharing technique that eliminates the need to make deep copies of those structures. This means that Clojure runs very fast.
- Clojure is supported. There are tutorials and blogs. There are IDE plugins. And there are mailing lists and user groups. If you program in Clojure, you won't be alone.

Conclusion

The last few decades have seen us migrate from procedures to objects. Now the physical constraints of our hardware is driving us to make a similar kind of paradigm shift towards functional languages. The next few years will see us experiment with many different projects using those languages as we try to figure out which functional languages are best. I fully expect Clojure to be ranked very highly when the results of those experiments come in. ■

Robert C. Martin (Uncle Bob) is an international software consultant guiding companies and programmers to raise the bar of software professionalism and craftsmanship.

Reprinted with permission of the original author.
First appeared in *http://hn.my/whyclojure/*.

Commentary

By MILES EGAN

You know, I really want to like Clojure, but so far I'm finding it a bit of a slog, for the following reasons:

1. stack traces are very hard to decipher
2. ubiquitous laziness can lead to some really subtle bugs
3. the syntax can be extremely obtuse, more so than other lisps
4. the basic ADT is just a keyed map, which provides a lot of flexibility but also can make data modeling very opaque, even compared to other dynamic languages
5. immutable and recursive as a default takes a lot of getting used to. things I could dash out in minutes in ruby take me hours sometimes
6. other languages have caught up enough that Lisp macros are not as game changing as they might have been ten years ago

I embarked on a project recently to explore machine learning algorithms, implementing them in both Scala and Clojure at the same time. I expected this to be an easy win for Clojure but, to my surprise, I'm much more productive in Scala, my code has fewer bugs, and I can refactor it much more aggressively. I realize that this isn't an either/or thing and it may be entirely my shortcomings at play here, but my instinct is that although Clojure is a superb lisp, it's still a lisp, and it's going to be a niche that most programmers won't choose to occupy.

Scalable PHP Hosting, made easy with the CatN Platform.

Instantly deploy your apps on our super cluster. No code changes necessary. SSH Access, Cron jobs, Git & SVN support all as standard.

from £5 per month

www.catn.com

Why I Chose Common Lisp over Python, Ruby, and Clojure

By SHANEAL MANEK

A FEW MONTHS AGO, two co-founders (Stu Wall and John Buchanan) and I (Shaneal Manek) started working on a startup called Postabon.

The idea behind Postabon is simple: we wanted to create a platform where users could find and share 'deals' at brick and mortar stores (be they sales, coupons, happy hours, specials, etc). For example, if I'm out near a mall and need a pair of jeans I can pull out my phone and see which store near me is having the best sale on pants right now.

I just wanted to talk about a few of the high level technical decisions that I've made – in the hopes that it could help other people starting new projects out (and that I can get some feedback and learn something myself). This post is going to be pretty tightly focused on the language I chose. I have a few other posts in mind on topics such as the database (BerkelyDB) and overall architecture that I'm planning to write up in the next week or two..

Language

The way I saw it, was that as the sole programmer I needed a language that was concise, powerful, and that let me work quickly. This set of requirements, in my mind, eliminated Java (and I don't know any C# …), which left me considering Python, Ruby, Clojure, and Common Lisp. I thought Haskell and Erlang were promising – but I'm just too inexperienced with them to commit to a large project (and, for better or worse, they aren't really known as great languages for web applications).

Python

I am fairly experienced with Python, there are lots of great libraries/frameworks for anything I would want to do, and it would be easy to bring other programmers on-board later. However there were a few negatives that, in aggregate, were enough to get me to move on.

First, and most importantly, the Python 2 to 3 conversion really scared me. Most libraries I wanted to use were still Python 2 only – which meant I would have had to write Postabon's back end in Python 2. But it makes no sense to me to write a large app, that I may have to maintain for years, in a language that has effectively received a death sentence. Python 2 is fine now – but in the coming months and years new libraries, features, and performance improvements are only going to be introduced in Python 3, and I didn't want to get left behind or forced to take on an expensive and time consuming port in the future.

Second, I know it's a bit cliche, but I don't like the Global Interpreter Lock, which makes it basically impossible to write multi-threaded apps that work on multiple CPUs. Of course, writing a multi-process app would be a reasonable work-around, but it is a bit of an annoyance.

Finally, Guido's disdain for functional programming makes it clear that I would be a second class citizen in Python-land. As a few small examples see:

- The fact that he proposed removing map, reduce, filter, and lambda from Python 3.
- His refusal to include tail call optimizations in the language, despite the minimal down side.
- The completely broken implementation of closures.

My mind just works functionally, and I don't want to be forced to fight the language I'm using at every turn.

> **“As the sole programmer I needed a language that was concise, powerful, and that let me work quickly.”**

Ruby

I've played with Ruby (mostly in the context of Rails) some – although I'm nowhere near as proficient with it as I am with Python, Lisp, etc. Ruby has a lot of the same strengths as Python, with fewer weaknesses. My criticisms about Python's GIL apply to it too – but again simply using processes is an acceptable work-around.

The biggest reason I chose not to go this route is that the Ruby community is just moving too fast for me right now. Some major component of the development/production stack of choice seems to be changing every 6 months (e.g., I've seen the webserver go from FastCGI/Apache to Mongrel to Phusion to Unicorn). I couldn't even easily figure out which version (1.8 or 1.9?) to use – or even which implementation (Ruby MRI, Ruby EE, JRuby, etc). Most of the articles I found online are a few months old and I am told they are no longer accurate.

Also, much of the Ruby community is built around Rails, and I'm a bit wary of using 'heavy weight' frameworks like Rails (or Django) on large custom projects. In my experience they make the first 90% of what I'm trying to do be really easy – but then make the last 10% a living hell since I need to modify something the framework never intended me to control. I probably could have written a "bare-bones" implementation of the site's back-end in Rails in a week instead of two weeks, but I would rather "waste" that one week up front to have more flexibility later.

For example, I ended up writing my own completely stateless session handling, building a fairly smart geospatial cache (in-memory R* trees that asynchronously persist to disk using B-Trees), and using a key-value store and raw b-trees for persistence (instead of a relational database). These (and a lot of other non-standard decisions I've made) are possible within Rails, but I think they would have cost me more time and energy than Rails would have saved up front – especially in light of my lack of experience with Rails.

In principle, I could see Ruby being the right choice for someone who was more experience with it upfront, is adequately plugged into the community and willing and able to switch out components of their stack. But, personally, I prefer a bit more stability in things.

Clojure

There are a lot of great things about Clojure: it runs on the JVM so I get all the Java libraries and that great JVM performance, it's functional from the ground up, and it even has macros.

However, 6 months ago Clojure hadn't even had it's 1.0 release (and the language was constantly changing). When I tried to download it the Slime integration was completely broken and I had to manually search through the SVN repos of several key components to find a relatively recent working set of tools that worked together.

My feeling is that things are better now (Clojure is 2 years old!) and if I were making this decision again today, I would give much more serious consideration to Clojure.

Common Lisp

Finally, that brings me to Common Lisp. I have plenty of experience writing web apps in Lisp, so the high barrier to entry wasn't a deterrent in my case. Although, make no mistake, that learning curve for writing a good CL web app is steep enough that I would warn most programmers to shy away from Lisp for writing a production app on a tight schedule.

I like that the language stabilized 15 years ago, that the kinks have been worked out, and that it has stood the test of time. I know the traditional complaint is the dearth of libraries, and there obviously aren't as many options as they may be for other languages. While I have been fortunate to find several great options for everything I've needed to do so far (JSON/XML parsing, HTTP servers and clients, ORMs, etc) – more obscure libraries like Thrift and OpenID support may be an issue in the future. The lack of libraries is, without a doubt, the biggest disadvantage of CL and one of the reasons Clojure is so appealing to me. I can usually just write my own foreign function interface into a C library – but that's really time consuming compared to downloading an egg/gem/jar.

The ability to use dynamic typing for most of my code but optionally give the compiler type-hints and get all the performance of a statically typed language for critical portions is a killer features that I still haven't found elsewhere.

On balance, I think that Common Lisp was the best choice for me given my background and the needs of this project. ■

Shaneal is a Harvard dropout with a long time interest in functional programming. After a stint at Charles River Analytics, an AI/ML focused defense contractor, he wrote Postabon in Common Lisp and raised a Series A round lead by Spark Capital. He recently left that position to join Allston Trading, where he uses functional programming in the world of high frequency trading.

Reprinted with permission of the original author.
First appeared in *http://hn.my/commonlisp/*.

Commentary

By MAHMUD MOHAMED

Did you really choose Lisp over alternatives? Before learning CL I was a fairly decent, C, C++ and Perl programmer. Did assembly, Pascal, TCL and Awk. Up to that point, I always had to pause a for a minute when starting a new project/script, think about its scope, and choose a language based on the necessary performance, development speed, expressiveness, available libraries, etc. (and whether whoever was going to read the code afterward knew the language; C was often a natural choice for code shared with others on Unix, C++ for MFC/COM, Perl for sysadmin stuff, and TCL and Awk for my own tools.)

I learned Lisp in over a month, to spite someone (I dared a notorious troll I would write an AI bot of his choice if he stopped spamming us, youthful bravado for sure, and I lost the bet) While researching "AI" I came across Winston and Horn's "Common Lisp," then the hyperspec, then a few more books over the course of a month. I sat down with SICP and did the exercises on my break, while I was in school and waiting tables.

After I learned it however, specially with CLOS, there was no contest. Three months after buying Sonya Keene's CLOS book it was fair to say I forgot all other programming languages. There were no more "projects"; I no longer had to sketch out designs on paper or do "requirement analysis" (something I was told in school was necessary for all software.) For once, the great ideas in my head were a new emacs buffer away. I could write code faster than I would in Perl, Awk or TCL, it ran as fast as C++, and it was more expressive than the English in my head. I could type "commands" into a shell get a dialog embedded in my window, a few more commands and it would move to the upper right corner, I could change its name property and add text to it, then I could fold that dialog box into a menu-item named "Help" in the menu bar and call that dialog box "About." Amazing.

I went on hacking like this for about year when I realized I was doing the "wrong thing." You see, I had been using CMUCL with its built-in editor and writing GUI applications in Motif (it was 2001 and Motif wasn't open source yet, so I got the hang of Lesstif and learned its quirks.) Right around this time, Linux GUIs were maturing and people were being

snobs about their Enlightenment themes and dissing each other over their choice of Window Manager. So I was peer-pressured into learning DHTML and Web Design. I read comp.lang.lisp and those too were snobbish condescending idiots who flamed everyone, especially competent programmers whose work I admired (including Scott McKay and Robert Fahlman (the very people who gave me my CMUCL.))

It was really hard to be a Lisper for a while, especially a young impressionable one who read cll uncritically; news of corporate giants coming with new tools and programming languages to enslave humanity were abound. First C++, then Java, then XML, and finally .NET. You literally had to pick your battles and choose a corporate sponsor or you would have no future in computing! (you think I am kidding?) cll is all doom and gloom, and of course, there are the obligatory stabs at Lisp vendors by Open Source proponents, and stabs at Open Source from people alleging it's killing our beloved vendors. Every once in a while there was news of a Lisp dialect that's going to kill Common Lisp (Smalltalk, Dylan, and the ancient religions of Mesopotamia.)

Fuck, that was painful.

All the while I was following this 4-year long intellectual funeral, becoming ever more "hardcore" and learning mathematics, there was a small group of "Yobos" silently kicking ass and churning out great software. CMUCL got forked to SBCL, added unicode support and threads, not to mention easy building, SLIME was a new Emacs mode better than anything before and since, Cliki was launched, C-L.net, and the #lisp IRC channel was born and hit puberty overnight. Perfect ecosystem.

Today, Lisp is nothing like what it was 8,7,6, even 2 years ago. It's not just "good" in the well-explored text book fashion; no, it's _good shit_. Get work done good. Think, hack, ship, bill for it good. 2-3 products per month good. You still have to know where things are, who is working on what, what's maintained and what's obsoleted by what. Sure. But there is absolutely no lack of libraries.

Special Commentary

By PETER NORVIG

I CAME TO PYTHON not because I thought it was a better/acceptable/pragmatic Lisp, but because it was better pseudocode. Several students claimed that they had a hard time mapping from the pseudocode in my AI textbook to the Lisp code that Russell and I had online. So I looked for the language that was most like our pseudocode, and found that Python was the best match. Then I had to teach myself enough Python to implement the examples from the textbook. I found that Python was very nice for certain types of small problems, and had the libraries I needed to integrate with lots of other stuff, at Google and elsewhere on the net.

I think Lisp still has an edge for larger projects and for applications where the speed of the compiled code is important. But Python has the edge (with a large number of students) when the main goal is communication, not programming per se.

In terms of programming-in-the-large, at Google and elsewhere, I think that language choice is not as important as all the other choices: if you have the right overall architecture, the right team of programmers, the right development process that allows for rapid development with continuous improvement, then many languages will work for you; if you don't have those things you're in trouble regardless of your language choice.

Still Hatin' on Git

Now with Added Actual Reasons!

By MIKE TAYLOR

Monday's little diatribe on git seemed to stir up quite a bit of strong opinion, both agreeing with me and disagreeing. As is often the case, they two camps seem to be split about 50-50, which makes me happy. I means I can be confident that I'm not talking complete arse-gravy, but I have a good chance of actually learning something.

For anyone who wasn't around on Monday, the substance of my post was "git is bad because I don't understand it." Or to paint myself in a slightly less bad light, "git is bad for me because it makes assumptions about how I work that don't match how I actually work." Or, to summarise the summary, "git is the work of Sauron Gorthaur, the Abhorred, servant of Morgoth Bauglir, the Dark Lord that was called Melkor, destroyer and despiser, the corrupt Ainu and corrupter of Arda."

I'll admit that yesterday's post was more a howl of anguish than a reasoned argument (although I still like the Harrier analogy). Having now calmed down a little, I thought it might be worth explaining myself a bit more, and addressing some of the comments, both here and at Hacker News.

Explain yourself, Taylor!

First, I was a bit shocked at the number of people (mostly at HN) who seemed to think that my whole problem with git is the need to specify -a when doing a git commit of all changed files. Folks, that was what is known as an example of how its model isn't a good fit for how a lot of us work. There are many more of these — for example, the fact that if you run git tag and subsequently push your repo, the tag doesn't get pushed.

Here is a more serious problem that I run into all the time (including once this very day):

- I make a one-line change to a single file.
- I commit my change.
- I `git push`, only to be told "! [rejected] master -> master (non-fast forward)" (This is git's cuddly way of telling you to do a pull first.)
- I git pull, only to be told "CONFLICT (content): Merge conflict in filename. Automatic merge failed; fix conflicts and then commit the result."

So far, so good — someone else edited the same region of the same file as I did (among their other edits): of course its a conflict, there's nothing git could do differently here but notify me and ask me to fix it. So I edit the file, fix the trivial conflict, and `git commit filename`.

Nuh-uh. "fatal: cannot do a partial commit during a merge."

Well, darn. So, OK, no problem, I already fixed the conflict, so now I'll just `git merge` again to get it to make its world consistent, right? Nope: "fatal: You have not concluded your merge. (MERGE_HEAD exists)" Well, duh! I was telling you to merge, you stupid git. You're the branches-and-merges-are-easy version-control system around here.

All right, so I will just `git pull` again, and this time the merge will work OK. Gotta work, yes? No. "You are in the middle of a conflicted merge." Well I knew that! That's why I am trying to resolve it. In fact, that's why I have resolved it! All I am asking you to do is accept my resolution. Please? Is that so much to ask?

But wait — it's worse than that! Not only can I not commit the file that had the conflict: I can't commit any other file. My whole repo is stuffed until I satisfy the hungry god.

> **“You start out believing what you’re told, that you can just use clone, pull, add, commit and push, and ignore the other 139 git commands. But you can’t.”**

But wait — it’s worse than that! `git status` shows that there are many, many modified files even though I know full well that I only edited the one line of the one file. Because all the other changes that my colleague made have been splunged across my tree and suddenly, what the heck, they’re my responsibility!

The solution turns out to be that I have to use `git commit -a`, i.e. commit all my changes in one go. But, dammit, git, that’s not what I wanted to do! If I like to commit on a file-by-file basis, what business is it of yours to forbid me? And: much, much worse: my `commit -a` re-commits all the changes my buddy had already made and commited! Seriously, git: what the hell?

Something is rotten.

A handy household hint: how to abandon your changes when dealing with a conflicted merge

Of course, in the merge-conflict scenario above, you may sometimes see that your friend’s changes are correct and leave yours irrelevant, so that you just want to throw your own changes away and use the version you pulled. Should be pretty simple, huh? Well, according to the top-voted answer to this question on Stack Overflow, the correct thing to do is:

```
git reset –hard HEAD
git fetch origin
git reset –hard origin
```

Talk about intuitive.

Here’s another one that I hate.

I needed to get back an older version of a binary file, `foobar.doc`, so I could compare it with the current version and see what had changed. (`git diff` is no use in this situation, because it works on text: I needed to get hold of the earlier revision so I could pull it into OpenOffice, which knows how to compare documents.)

The command that does this is `git show`, which writes the old version on standard output so you can redirect it into the file of your choice. In general, the command is `git show revision:pathToFile`. revision can be `HEAD^` to mean “the one before the current one.” But when I did `git show HEAD^:foobar.doc`, I got back a more than usually incomprehensible error message.

```
fatal: ambiguous argument
'HEAD^:foobar.doc': unknown revision
or path not in the working tree.
Use '–' to separate paths from
revisions
```

It turns out that this is because the file in question isn’t at the top level of the git module: when I said pathToFile earlier, I really meant it — you have to give the whole path relative to the root of the module. (The bit of the error message about using ‘–’ turns out to be complete red herring.) So I have to use `git show HEAD^:dino/epub/foobar.doc`, even though I am already in the directory dino/epub.

You can’t tell me that’s right.

What makes it much, much worse

I just know that someone — probably several someones — are going to reply to this article saying: "you are mistaken; git is correct." These people, most of them kindly and gently, will talk me through my misconceptions about what a version is, what a commit is, how it affects the index, what a merge means, why it has to be this way and why I am sadly mistaken in thinking it should be otherwise. If we were discussing this in a pub rather than over the Internet, they would probably find a scrap of paper and draw a nice state-transition diagram for me, showing how the various change-sets propagate between the various checkouts, branches, indexes and repositories. Nine times out of the ten, this will be done with patience and tact, with a side of burning evangelistic fervour.

Here is my rebuttal:

I. Do. Not. Care.

This is what I meant last time about git not degrading gracefully. It's great that it handles multiple local and remote branches and merges and all the other stuff, but you can't Just Not Know about that stuff. You start out believing what you're told, that you can just use `clone`, `pull`, `add`, `commit` and `push`, and ignore the other 139 git commands. But you can't. You have to keep learning more of them, and learning new and baroque ways of invoking them; and, more importantly, learning more of the concepts. Any day now, I expect to learn that before git moves files into the index, it first keeps them in a top-secret pre-index stash-cache area.

Who is the user around here?

Is it terribly old-fashioned of me to believe that when a user uses a tool, he should be the one who determines how it's used?

The bottom line for me with git is that I am sick of being pushed around. It swans about as though it owns the place. It make arbitrary demands. It tells me what to do. It's as though ext2fs insisted on particular file-naming conventions, or vi mandated a specific indentation regime for your C code.

Unless of course …

Unless git is a hammer and I am trying to use it as a screwdriver. Or perhaps more appositely, it's a bandsaw and I'm trying to use it as a bread-knife. Or indeed, it's a Harrier and I'm trying to use it as a bicycle.

Which I suspect is the case, and why I think the move back to CVS/Subversion might be the way to go.

The conclusion of the matter

One of the more thought-provoking comments on the last article was this one from teh:

> *I disagree with "Git's just version control. I resent the idea of investing a month of evenings and weekends just to be able to check my freakin' files in."*
>
> *Version control is not "just" version control, it's a first class tool for every programmer, up there with recursion and all that jazz. A programmers work is transforming code from one state to another. Git treats these transformations as first class objects, allowing you to rewrite or reorder them, have alternative transformation branches, send them around etc.*

I still, frankly, resent the idea of spending the amount of time that I know will be necessary to become a git wizard. But I am increasingly reconciled to the idea that it will be time invested rather than time wasted.

I don't intend to be graceful about this — I plan to mutter and groan and whine incessantly — but I have a horrible feeling the the outcome of this article and its predecessor is that I'm going to end up Deeply Learning git. I don't want to — I hate the idea of ending up as one of the Git Advocates that I was complaining about earlier — but I think I'm going to have to. And if I do it, I'm going to do it properly, which means *sigh* another book, and probably another Long Overdue Serious Attempt At series.

As Xiong Chiamiov wrote in a comment:

> *I use it because the benefits outweigh all of the things that you mentioned.*

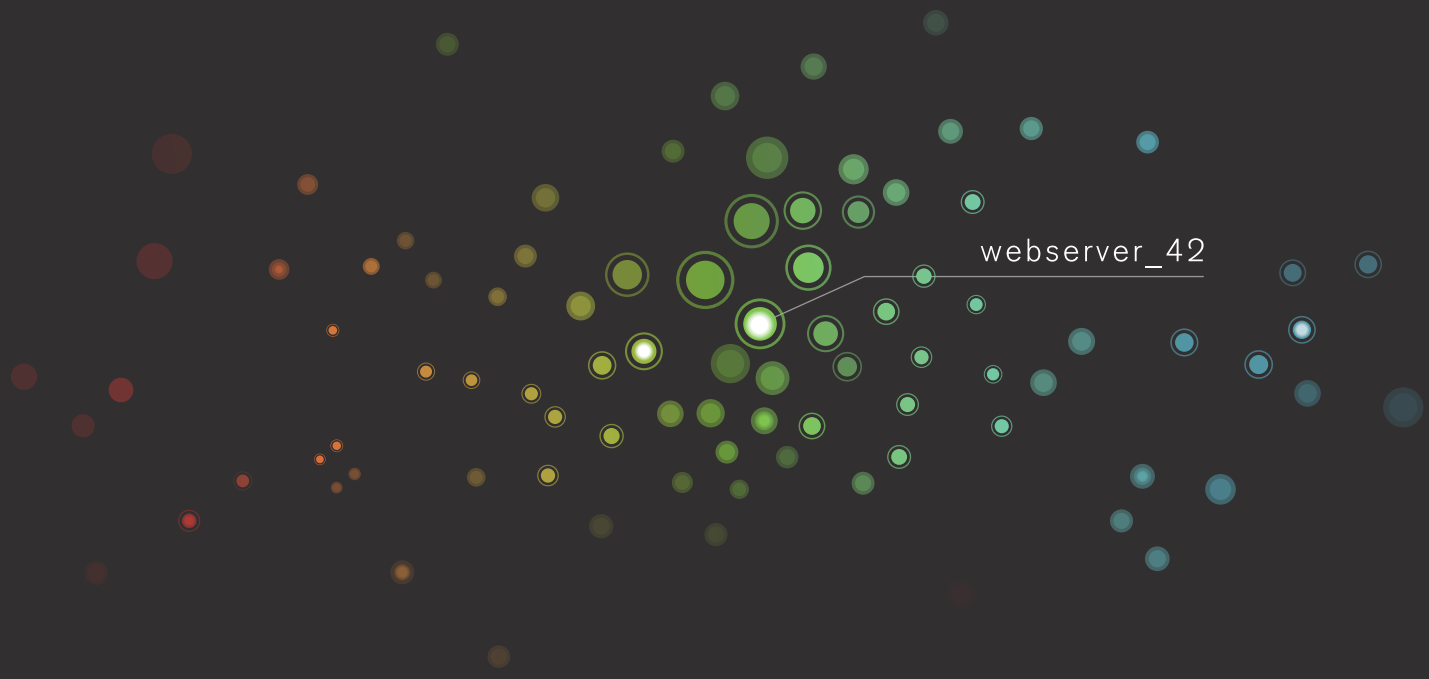
Dammit all, he's right, isn't he? ■

Mike Taylor is a computer programmer by day and a dinosaur palaeontologist by night, twin obsessions reflected in his two blogs, http://reprog.wordpress.com/ and http://svpow.wordpress.com/. He started programming in 1980, on a Commodore PET 2001 and a Video Genie, and has hardly stopped since.

Reprinted with permission of the original author.
First appeared in *http://hn.my/hatingit/*.

These are your servers

These are your servers on Cloudkick



Any questions?

cloudkick.com
415.779.5425

support for 8 clouds + dedicated hardware

cloudkick

the best way to manage the cloud

My Common Git Workflow

By YEHUDA KATZ

A RECENT POST THAT was highly ranked on Hacker News complained about common git workflows causing him serious pain. While I won't get into the merit of his user experience complaints, I do want to talk about his specific use-case and how I personally work with it in git.

Best I can tell, Mike Taylor (the guy in the post) either tried to figure out a standard git workflow on his own, or he followed poor instructions that tried to bootstrap someone from an svn background while intentionally leaving out important information. In any event, I'll step through my personal workflow for his scenario, contrasting with subversion as I go.

Cloning the Repository

The very first step when working with a repository is to clone it. In subversion, this is accomplished via `svn checkout svn://url/to/repo/trunk`. This retrieves the most recent revision of the trunk branch of the repository.

In git, this is accomplished via `git clone git://url/to/repo` (the http protocol is also possible). This retrieves the entire repository, including other branches and tags.

Making the Change

In both git and subversion, you make the change using a normal text editor.

After Making the Change

In git, you make a local commit, marking the difference between the most recent pulled version (master) and the changes you made. In subversion, the normal workflow does not involve making a change, but apparently some people make manual diffs in order to have a local copy of the changes before updating from the remote. Here's an example comment from the Hacker News post:

> *I'll tell you what happens when I use svn and there's been an upstream change: I never update my local tree with local modifications. Instead, I extract all my local changes into a diff, then I update my local tree, and then I merge my diff back into the updated tree and commit.*
>
> *When I need three-way merging, which isn't often – usually patch can resync simple things like line offsets – it's handled by a file comparison tool. I have a simple script which handles this.*

My personal process for making the commit in git almost always involves the gitx GUI, which lets me see the changes for each individual file, select the files (or chunks in the files) to commit, and then commit the whole thing. I sometimes break up the changes into several granular commits, if appropriate.

Updating from the remote

Now that we have our local changes, the next step is to update from the remote. In subversion, you would run svn up. Here, subversion will apply a merge strategy to attempt to merge the remote changes with the local ones that you made. If a merge was unsuccessful, subversion will tell you that a conflict has occurred. If you did not manually save off a diff file, there is no way to get back to the status from before you made the change.

In git, you would run `git pull`. By default, git applies the "recursive" strategy, which tries to merge your current files with the remote files at the most recent revision. As with subversion, this can result in a conflict. You can also pass the `--rebase` flag to pull, which is how I usually work. This tells git to stash away your commits, pull the remote changes,

Operation	git	svn
Clone a repository	`git clone git://github.com/rails/rails.git`	`svn checkout http://dev.rubyonrails.org/svn/rails/trunk`
Preparing changes	`git commit (using gitx)`	nothing or create a manual diff
Update from the remote	`git pull --rebase`	`svn up`
Resolving conflicts without –rebase	`git add` then `git commit`	N/A
Yikes! Rolling back	`git reset –hard`	`svn up -rOLD` then apply diff (only if you manually made a diff first)
Pushing	`git push`	`svn commit`

Workflow comparison between git and subversion.

and then reapply your changes on top one at a time.

If you use `--rebase`, you may get a conflict for each of your local commits, which is usually easier to handle than a bunch of conflicts all at once.

I definitely recommend using `--rebase` which also provides instructions for dealing with conflicts as they arise.

In either case, in my experience, git's merging capabilities are more advanced than subversion's. This will result in many fewer cases where conflicts occur.

Resolving Conflicts

From here on, I am assuming you followed my advice and used `git pull --rebase`.

If a conflict has occurred, you will find that if you run git status, all of the non-conflicting files are already listed as "staged," while the conflicting files are listed outside the staging area. This means that the non-conflicting files are already considered "added" to the current commit.

To resolve the conflicts, fix up the files listed outside the staging area and `git add` them. Again, I normally use gitx to move the resolved files into the staging area.

Once you have resolved the conflict, run `git rebase --continue`. This tells git to use the fixed up changes you just made instead of the original commit it was trying to put on top of the changes you got from the remote.

In subversion, if you got a conflict, subversion will create three files for you: `file.mine`, `file.rOLD`, and `file.rNEW`. You are responsible for fixing up the conflicts and getting back a working file. Once you are done, you run svn resolved.

***NOTE:** If you had not used* `git pull --rebase`*, but instead did raw* `git pull`*, you would fix up the files, add the files using* `git add` *or gitx, and the run* `git commit` *to seal the deal*

Yikes! Something went wrong!

In git, if something goes wrong, you just run `git reset --hard`, which will bring you back to your last local commit.

In subversion, it's not always possible unless you manually stored off a diff before you started.

Pushing

Now that you're in sync with the remote server, you push your changes. In git, you run `git push`. In subversion, you run `svn commit`.

One Glossed-Over Difference

Subversion allows you to commit changes even if you haven't `svn uped` and there have been changes to the remote, as long as there are no conflicts between your local files and the remote files.

Git never allows you to push changes to the remote if there have been remote changes. I personally prefer the git behavior, but I could see why someone might prefer the subversion behavior. However, I glossed over this difference because every subversion reference I've found advises running `svn up` before a commit, and I personally always did that in my years using subversion.

Note that I am not attempting to provide an exhaustive guide to git here; there are many more git features that are quite useful. Additionally, I personally do a lot of local branching, and prefer to be able to think about git in terms of cheap branches, but the original poster explicitly said that he'd rather not. As a result, I didn't address that here.

I also don't believe that thinking of git in terms of subversion is a good idea. That said, the point of this post (and the point of the original poster) is that there are a set of high-level version control operations that you'd expect git to be able to handle in simple cases without a lot of fuss. ■

Yehuda Katz (wycats) is a programmer at Engine Yard, with a background in accounting, journalism, and twelve other things that would surprise you. He grew up in New York, and now lives in sunny California. He's been working with Rails since before 1.0, is a Core Team Member on the jQuery project, and was the lead developer on the Merb project. In the time not spent working on open source, he writes about open source, and talks about open source. Go figure.

Reprinted with permission of the original author. First appeared in *http://hn.my/gitworkflow/*.

Why I Switched to Git From Mercurial

By GARY BERNHARDT

I used Mercurial for three years, but started switching to Git about a year ago. I now grudgingly recommend Git to anyone who intends to be a full-time programmer. Git's interface is bad in many ways, which is the main complaint about it, and it's a legitimate one. It's just an interface, though, and this is a tool you're going to use all day, every day, in a wide variety of situations.

Here are all of the ways that Mercurial has harmed me, or that I've seen it harm others, and the ways in which Git does good where Mercurial does evil:

❶ Mercurial is bad at handling large amounts of data. A friend accidentally committed a couple GB of data into a Mercurial repository. It became completely broken, to the point where most commands would die because they ran out of memory. Git has no problem with large data. It's awesome to be able to put, say, an entire home directory or ports install under version control without fear. (I recently put a multi-gigabyte MacPorts install under version control with Git without even thinking about it.)

❷ Mercurial's repository model is clunky and stays hidden in the background (this is a bad thing; don't let anyone tell you otherwise). If you have a Mercurial repository whose size is dominated by a single, 20 MB directory, and you then rename that directory, your repository just doubled to 40 MB. This has limited my ability to manage real-life Mercurial repositories. Git's repository model is so good that I only hesitate slightly when calling it perfect. It allows me to think about what's going on in the repository with an ease that I never had with Mercurial, despite using it much more than Git.

❸ Mercurial is not safe. Both systems ship with many commands that change history, but Git's data model is such that even a "delete" isn't really a delete. Destructive commands just create new nodes in the history graph, then adjust the branch to point at them. Whenever this happens, the old branch HEAD is still accessible using the reflog. That's awesome, and it alone would bring me to Git.

Mercurial's answer to this is weak: destructive commands shove a bundle file into a subdirectory of the Mercurial repository; you have to manually manipulate it if you want to get the data back. Except some of the destructive commands don't dump the bundle files, which has made me lose actual data in the past. Even for the commands that do dump the files, keeping track of them, and which applies where, becomes difficult fast.

tl;dr:

- Mercurial made my repositories huge for no reason.
- Mercurial broke when my friend put lots of data in it.
- Mercurial lost my data when I did a destructive command.
- In a year of Git, it's never done anything nearly as bad.

I'm sorry for recommending software with a confusing interface. But you'll be spending a lot of time with it; it's worth getting over the initial hurdle of confusion.

...until something better comes along, of course. ■

Gary Bernhardt is a creator and destroyer of software compelled to understand both sides of heated software debates: Vim and Emacs; Python and Ruby; Git and Mercurial. He blogs about software at extracheese.org.

Reprinted with permission of the original author.
First appeared in *http://hn.my/gitmer/*.

Commentary

By MATT BRUBECK

I've used Git for three years, at two different companies and various personal and community projects. I sync my home directory between hosts using git, I have administered multi-user git repos using gitosis, and I have been a GitHub user since early 2008.

I started using Mercurial two months ago when I joined the Mozilla corporation, and now use it every day on one of the biggest and best-supported installations in the world. I also started using BitBucket for some of my Mozilla-related personal projects.

It might be that I don't fully "get" Mercurial yet, but I still find myself frequently missing Git. My first impression is that Git has a simple flexible model that supports a complex front-end, while Mercurial has a simple extensible front-end that ends up creating a somewhat more complicated model.

Any time I need to step outside the standard commit-merge-push workflow, I find the higher level of abstraction between Mercurial's user commands and its database makes it harder to understand what I'm doing, and sometimes prevents me from doing what I want. Things like rewriting history (e.g. for rebasing), or combining changes from multiple remote repos with different sets of branches, are still possible in Mercurial - but they usually require plugins to do well, there are more different incompatible ways to do them, and there are more opportunities to mess up your repo.

More concretely, I find that MQ is the best way to do many tasks in Mercurial that I would do in git with plain old branches and commits, and for many of these uses MQ is both more complicated and less flexible than the equivalent git commands. (But there are other uses where MQ is really better than the alternatives.) I also find that it's much more annoying to manage short-lived throwaway or topic branches in Mercurial, so much that I simply avoid using them much of the time.

But perhaps I'm just brainwashed (or brain-damaged) from too many years with Git. :)

What were the key decisions that Larry Page and Sergey Brin made in the early days of Google?

By BRANDON SMIETANA

Larry Page's brother Carl Page had experience with venture capitalists, having sold eGroups to Yahoo for $432 million. Because of Carl Page's experience with venture capitalists Page and Brin were extremely unwilling to cede control over their company to investors.

Learning the lessons of Carl Page, Page and Brin delayed venture capital financing until they were almost profitable, resulting in a higher valuation and less loss of control over their company; Google's VC round valuation was extremely high by historical standards and this reflected their bargaining position at the time. They played two prominent separate venture capitalist firms against each other to minimize their loss of control and to maximize valuation and double the number of social connections the company had access to. Page and Brin issued themselves special Class-B shares which held 10 votes per share compared to the 1 vote per share of the Class-A and common stock. This effectively eliminated the possibility of investor take-over of the company by shareholder vote, as each founder had more votes than all the outstanding shares of Class-A and common stock. Sergey and Larry were also extremely careful about choosing their board members and put an emphasis on retaining control of the board.

The one concession Page and Brin made was an agreement to bring on an outside CEO. However they delayed doing this for years and antagonized all prospective CEO candidates, merely meeting with them to placate their VC investors. Under pressure from the VC, Page and Brin took on Eric Schmidt as CEO, but only after examining a large number of candidates. Page and Brin put an unusual amount of time into CEO selection and chose to delay taking on a CEO until they found one who was compatible with and would not adversely affect their company culture. Eric Schmidt was probably a particularly good fit with Google's culture because of his tenure at Sun.

Retaining control over the company proved to be crucial to Google's success. Most of the early company revenues came from enterprise search, and the investors and Eric Schmidt pressured the company to drop consumer facing search and to focus on the enterprise market. However Page and Brin disregarded this advice as they anticipated the growth of a market for online advertisement. At the time this decision was being made, the New York Times was quoting experts as saying "No one will ever make $250 million dollars a year from online advertising." It has also been said that Page and Brin disregarded pressure from investors to copy Yahoo! and diversify Google into a portal site, deciding instead to focus on the core search and advertising market.

Another key decision Google made was to keep their financials absolutely secret. Page and Brin learned something from the Internet Explorer and Netscape browser wars and had a healthy fear of Microsoft coming in and crushing them. They therefore kept their financials absolutely secret, did no marketing, did not evangelize their product and hid the fact that they were making money at all. Not even employees in the company had access to the financial information due to the fear that knowledge about the profitability of search could spur competition with Microsoft.

Another key decision in the company's history was achieving early profitability, and going for an IPO, using the funds to acquire and build an advertising network. Google would not have been as successful as it is today if it had chosen to be acquired instead of achieving profitability and doing an IPO. It has been suggested that Google's hand was forced in the decision to IPO rather than undergoing acquisition because it was unable to find a company interested in purchasing them.

Commentary

By PAUL GRAHAM

Managing VCs well was not what made Google successful – or what makes any startup successful. The real key decisions were things like realizing search itself was important, at a time when all the other search engines thought it was unsexy, and were trying to get people to start calling them "portals" instead; designing the architecture to work on large numbers of unreliable, cheap computers; understanding how important speed was; making the site uncluttered; deciding to hire only very smart people; etc. That's what made the company valuable, and if it hadn't been valuable it wouldn't have mattered how well they'd avoided dilution.

Tell us what you think

Let us know what you liked, and what we need to work on.
Please share your thoughts so we can improve the coming issues.

hackermonthly.com/feedback/