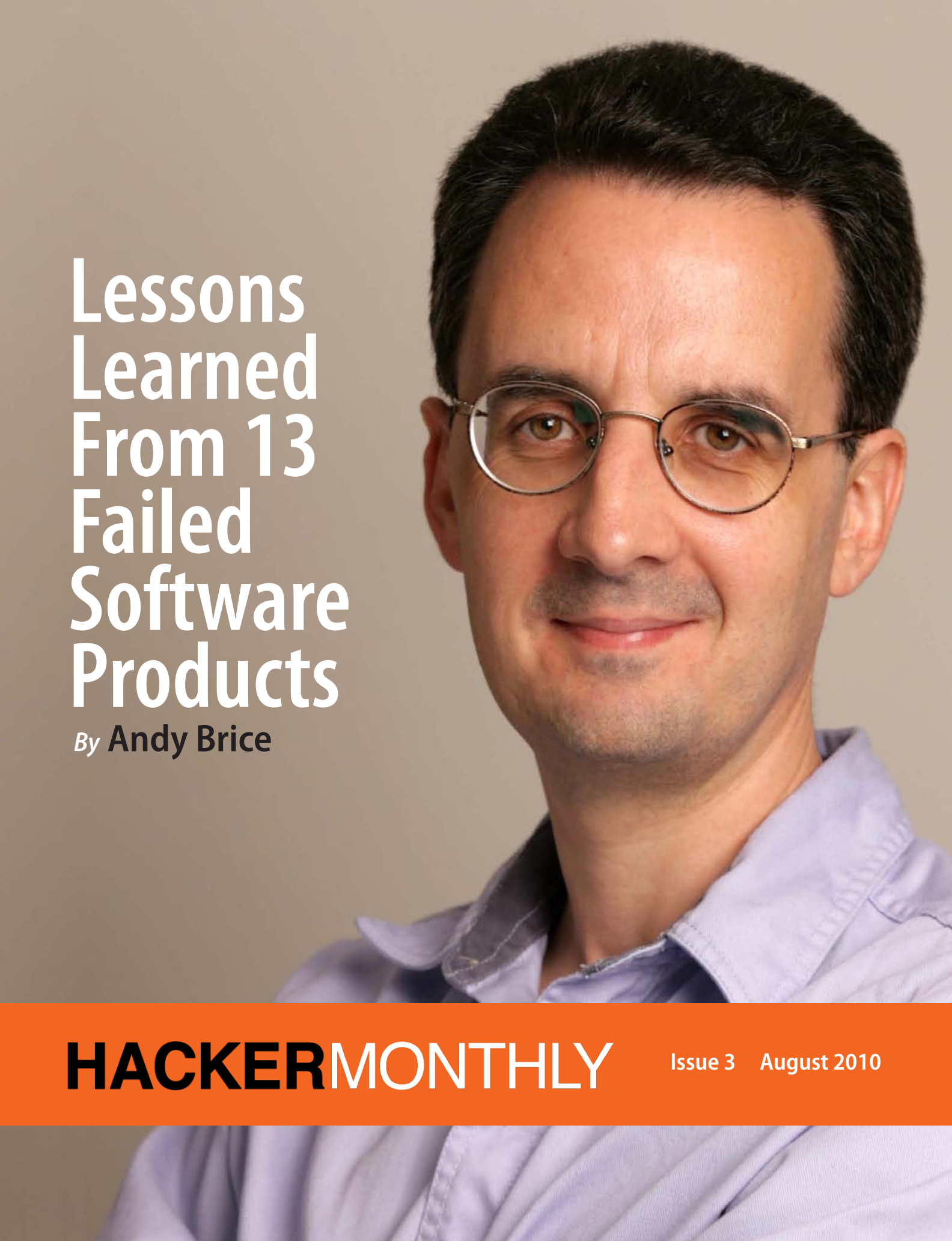


Lessons Learned From 13 Failed Software Products

By Andy Brice

HACKERMONTHLY Issue 3 August 2010

LET'S GET

SOCIAL

MailChimp 5.2

Curator
Lim Cheng Soon

Contributors
Andy Brice
Jason L. Baptiste
Jason Schuller
Hillel Cooperman
Scott Edward Walker
Xavier Shay
Nikos Moraitakis
Dave Pell
Matt Might
Brian Carper
Alan Skorkin
Daniel Spiewak
John D. Cook

Proofreader
Ricky de Laveaga

Illustrators
Jaime G. Wong

Printer
MagCloud

Advertising
ads@hackermonthly.com

Contact
cheng.soon@hackermonthly.com

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

Curator's Note

In every new issue, I try something different. This issue, I have included more technical articles, based on suggestions from our programmer-heavy readership. I have also taken a big risk by including longer (up to 10-page) articles, bumping our total pages to a whopping 56 pages — a 16-page increase from the usual issue. Another new experiment is the 'Tech Jobs' section (huge props to Zach Epstein, who suggested it) where companies can post programming and other technical-related jobs. You might notice most of the URLs in this issues are shortened under *hn.my*. It is Hacker Monthly's own URL shortening service.

Design-wise, the font is slightly (1pt) smaller in this issue with a wider leading. I've combined the use of left-justified and left-aligned paragraphs instead of choosing one style for the entire issue. This issue also marks the first time a real person (Andy Brice, who wrote and edited the fantastic featured article) has made it onto the front cover.

Hacker Monthly is slowly taking shape, one issue at a time. I need your feedback the most at this stage in particular. Reach me directly at *cheng.soon@hackermonthly.com*. ■

— Lim Cheng Soon

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*.

On The Cover: Andy Brice
Photo: *Andrew Fosker (http://www.secondsleft.co.uk/)*

Contents

FEATURES

STARTUP

SPECIAL

PROGRAMMING

Illustration: Jaime G. Wong (*http://retrazos.pe/*)

Lessons Learned From 13 Failed Software Products

By ANDY BRICE

> "No physician is really good before he has killed one or two patients." — Proverb

Software entrepreneur culture is full of stories of the products that succeeded. But what about the products that failed? We rarely hear much about them. This can lead to a very skewed perspective on what works and what doesn't (survivor bias). But I believe that failure can teach us as much as success. So I asked other software entrepreneurs to share their stories of failure in the hope that we might save others from making the same mistakes. To my surprise I got excellent 12 responses, which I include below along with one of my own. It is a small sample and biased by self-selection, but I think it contains a lot of useful insights. It is an unashamedly a long post, as I didn't want to lose any of these insights by editing it down.

CASE #1

DRAMA

DRAMA (Design RAtionale MAnagement) was a commercialization of a University prototype for recording the decision-making process during the design of complex and long-lived artefacts, for example nuclear reactors and chemical plants. By recording it in a structured database this information would still be available long after the original engineers had forgotten it, retired or been run over by buses. This information was believed to be incredibly valuable to later maintainers of the system, engineers creating similar designs and industry regulators. The development was part funded by 4 big process engineering companies.

Why it was judged a commercial failure: Everyone told us what a great idea it was, but no-one bought it. despite some early funding from some big process engineering companies, none of them put it into use properly and we never sold any licences to anyone else.

What went wrong:

- Lack of support from the people who would actually have to use it. There are lots of social factors that work against engineers wanting to record their design rationale, including:
  - » The person taking the time to record the rationale probably isn't the person getting the benefit from it.
  - » Extra work for people who are already under a lot of time pressure.
  - » It might make it easier for others to question decisions and hold companies and engineers accountable for mistakes.
  - » Engineers may see giving away this knowledge as undermining their job security.
- Problems integrating with the other software tools that engineers spend most of their time in (e.g. CAD packages). This would probably be easier with modern web-based technology.
- It is difficult to capture the subtleties of the design process in a structured form.
- A bad hire. If you hire the wrong person, you should face up to it and get rid of them. Rather than keep moving them around in a vain attempt to find something they are good at.
- We took a phased approach, starting with a single-user proof of concept and then creating a client-server version. In hindsight it should have been obvious that not enough people were actively using the single-user system and we should have killed it then.

Time/money invested: At least 3 man years of work went into this product, with me doing most of it. Thankfully I was a salaried employee. But the lack of success of this product contributed to the demise of the part of the company I was in.

Current product status: The product is long dead.

Any regrets? It was a fairly painful experience. I would rather have spent all that money, time and energy on something that someone actually used. But at least I learnt some expensive lessons without using my own money.

Lessons learned:

- Creating a new market is difficult and risky.
- Changing people's working habits is hard.
- Social factors can make or break a product. The end-users didn't see anything in it for them.
- If the end-users don't like a product, they will find a way not to use it, even if their bosses appear to be enthusiastic about it.
- Talk is cheap. Lots of people telling you how great your product is doesn't mean much. You only really find out if your product is commercially viable when you start asking people to buy it.

Contributor: Andy Brice
(http://successfulsoftware.net/)

CASE #2

CleanChief

CleanChief was to be 'The easy management solution for cleaning organisations'. Managing assets, employee schedules, ordering supplies, you name it CleanChief handled it. Essentially it was light weight accounting software for cleaning companies.

Why it was judged a commercial failure: A small number of copies were sold. No one is actively using it at present. Once I realised that it wasn't a complete product and that additional development was required I moved on to other product ideas. I had basically run out of enthusiasm for the product.

What went wrong:

- I am not an accountant.
- I have never run a cleaning company.
- I developed it for more than two years without getting feedback from real cleaning companies. I was arrogant enough to think that I knew what they wanted (or could work it out on my own). Or maybe it was that I was just where I was most happy and comfortable – writing software. Talking to real users was new and to be honest a bit scary for me.
- A successful cleaning company operator, a friend of a friend, offered to become involved for a 30% share. This was a gift from the heavens, exactly what I needed. I refused.
- In a way, even though I spent so long on the product, I gave in too soon, I was just getting feedback from real users, just getting my first batch of sales when I decided to move on.
- I developed the application in VB6 even though I knew it was outdated technology when I started the project. This meant there was no 'cool factor' when discussing it with other developers, I told myself it didn't bother me, but it probably did.

Time/money invested: I worked on it at night and weekends for about 2 1/2 years. I paid for graphic design work,

> “Go for it, maybe you win, maybe you fail, but you will grow and get tons of useful knowledge on the way.”

purchased stock icons and images. I probably spent a couple of thousand Australian dollars in total and an awful lot of time.

Current product status: I moved on to other products that have gone much better. My newer products were released in months rather than years and I looked for real feedback from real users from day one. They are:

- QueryCell – an Excel add-in making SQL in Excel easy.
- QuizNightChief – the easy way to organise a quiz Night.
- CustomerCradle – The easiest way to record and report on where your customers come from.

I do occasionally ponder returning to CleanChief and trying to raise it from the ashes.

Any regrets? No. Looking back I learned a few lessons from a huge amount of time and work, it was a very inefficient way to learn those lessons. But when you are new to something like starting a business or creating useful software being inefficient at learning lessons is the best you can do, it’s a thousand times better than not learning lessons at all.

I learned so much more in my two and a half years of trying to develop CleanChief than I did in the two and a half years prior to that, during which time I really wanted to start a software business but didn’t take any action.

Lessons learned: Hearing or reading some piece of advice is totally different to living it. Here are some of the ideas that I always agreed were true but didn’t fully understand the implications of until I had lived them out:

- Force yourself to get out and talk to people. Ask their advice. Almost everyone will help if you ask them for feedback.
- Force yourself to cold call a few businesses in your target market.
- Create a plan of how to market your product.
- Try and use your product as much as possible as you build it.
- Get out of your comfort zone from day one.
- Do not have the mind set that the day you release version 1.0 is the finish line, it’s the starting line, so hurry up and get there.

Contributor: Sam Howley
(http://oakfocus.net/)

CASE #3
ChimSoft

ChimSoft – Software for Chimney Sweeps.

Why it was judged a commercial failure: I believe this failed for two reasons:

- Focusing on too small of a niche
- Me not being able to work full time on it.

I don’t consider it a complete failure because I sold two copies when it retailed for $2k, and maybe 10-15 more copies when I lowered the price to $200. Those sales proved that I wasn’t completely off base in thinking there was a market for the software, but the cost of customer acquisition and the size of the market were too small. Customers wanted to have a bunch of phone calls, face-to-face etc… the type of stuff you only see with much more expensive software. The problem was that for a niche this small we had to charge a lot of money to make it worthwhile for us, but the customers were small businesses where this is a major investment, so the fit was never right. The other issue was the people that did buy it were not super tech-savvy, so there was a

high cost of support that made even a $200 product not worth it.

What went wrong:

- Having all partners who were not full-time, and had equal equity. I ended up doing most of the work and this is the main reason I didn't force success is I felt I was in it alone.
- Focusing on too narrow of a niche. The plan all along was to expand for all service industries, but it was much harder to make that move than we expected.
- Not researching pricing more, we knew small businesses made major purchases for things that really helped their business, but I think it would have been better to have a cheaper product with wider appeal than an expensive product with narrow appeal.

Time/money invested: I invested maybe a year of time and $3k into the company. I did not take any huge risks on it, so there were no big negative outcomes.

Current product status: The company folded in 2007, I refocused my efforts on my existing companies (AUsedCar.com and BudgetSimple.com) and both have been doing well enough that I quit my day job.

Any regrets? I don't regret it entirely, I think I learned several valuable lessons about working with other people, small business sales, trade-shows and software development.

Lessons learned:

- Pick partners wisely. Don't try to be even-steven with equity. Use restricted stock to ensure everyone does their part.
- Know what your customers expect (24/7 phone support?) to determine if you can do this while working a day job.

Contributor: Phil Anderson
(http://www.startupdetails.com/)

CASE #4

PC Desktop Cleaner

PC Desktop Cleaner. Simple software that cleans your desktop and archives your files.

Why it was judged a commercial failure: My goal was to sell 10 units per month. I've sold less than 1 unit per month.

What went wrong:

- I think that the product concept is not useful enough. It's not a thing that people would pay for.
- The market exists (some people buy) but it's too little or difficult to reach.
- I didn't do any market research. I just got in love with the idea and did it. Later, I've learnt to use "lazy instantiation marketing" and have trashed a lot of embryo projects. :-)

Time/money invested: I think I wasted near $500 in development tools and some freelancers. Not too much.

Current product status: I'm still selling it. I've thought about other products, but not really decided yet.

Any regrets? No, it was a lot of fun and I learnt lot of things. In my "day job" I own a small firm that sells software for production scheduling. I learnt a lot about SEO and AdWords in the DesktopCleaner project that I'm now using with great results.

Lessons learned: Go for it, maybe you win, maybe you fail, but you will grow and get tons of useful knowledge on the way.

Contributor: Javier Rojas Goñi
(http://tekblues.com/)

CASE #5

Smart Diary Suite

Smart Diary Suite.

Why it was judged a commercial failure: It sells and the profits cover current investments in the product, but there is little left over on top of that.

What went wrong: If I had a chance to do anything differently:

- Take it seriously from day one.
- Never stop developing and supporting.
- Invest as much as possible in marketing early on.
- Don't stop believing in your creation.

Time/money invested: Up to this point, I have spent 13 years on Smart Diary Suite and a lot of money went into buying hardware, software, hosting, marketing, etc… All of that money came from my day job, but at this point SDS has recovered all of that back and is now making a small profit. The actual amount is hard to calculate (over the 13 year span), but we would be talking in tens of thousands of US dollars.

Current product status: For a while it may have seemed like SDS was not going to be successful, but that's probably my fault – I stopped believing for a little while. Now I am back, starting again and this time I'll make sure it doesn't fail.

Any regrets? I do not regret doing it. I regret allowing myself to stop working on it, basically bailing out on it for a while – that is my biggest mistake.

Lessons learned: If you want a successful product – believe in it and let others know that you believe in it.

Contributor: Dennis Volodomanov
(http://smartdiarysuite.blogspot.com/)

> “Develop something which you find useful, instead of second guessing others.”

CASE #6

Highlighter

Highlighter. A utility to print neatly formatted, syntax highlighted source code listings.

Why it was judged a commercial failure: I earnt a grand total of £442.52 (about $700 in todays money) in just over two years, so I guess it paid for itself if you exclude my time.

What went wrong: Since it was my first product I was very green about both marketing and product development. I suggest the following would have made things better:

- Get feedback from potential users about the product (e.g. from the ASP forums). Some parts of the program where probably too option-heavy and geeky.
- Diversify. If people didn't want to print fancy listings, maybe they would have wanted them formatted in HTML.
- Better marketing. I'm not sure this would have saved it, but all I knew in those days was uploading to shareware sites. I never even sent a press release.

  I figure it failed simply because it was a product nobody wanted. Actually, more importantly than that,, it was a product *I* didn't want to use, but it developed from a larger product I was working on, on the assumption I could earn some money on the side from part of the code. Since then I've stuck to products which I've actually wanted to use myself. There's a lot to be said for dogfooding, not just for debugging, but for knowing where the pain points are and what extra features could be added.

Time/money invested: I would guess a couple of months of evening/weekend development time. Financially there was little spent, except that I offered the option of a printed manual and CD for an extra charge. One customer took me up on the offer, so I had to get 100 manuals printed and 99 of them went in the bin.

Current product status: I moved on to another product which has sold over £50,000 and a third which has earnt even more than that. Not enough to retire on but considering I only do this part-time it must work out at a great hourly rate. There's a lot to be said for not giving up…

Any regrets? Nope. I figure every failure in life teaches you valuable lessons. Of course if I'd made a large financial investment I may feel differently, but that's one of the big advantages of software over physical product sales.

Lessons learned: Just to reiterate – develop something which you find useful, instead of second guessing others.

Contributor: Mike Sutton
(http://www.rudabet.com/)

CASE #7

R10Clean

R10Clean. A data cleaning and manipulation tool.

Why it was judged a commercial failure: In the 18 months or so it's been on the market I have sold 6. It has been £199, £99 and £19 – with no effect on sales!

What went wrong: Not sure what I did wrong? The product is maybe too techie?

Time/money invested: No effect financially as at the time I was in a strong financial position.

Current product status: I still have it for sale but do not market it at all. I have other products.

Any regrets? I don't regret it as it saved me a ton of time when I was working with legacy databases, as a commercial product it has been raved about (once!) and received a good review from the Kleper report, but has failed totally.

Lessons learned: Advice to others? Just because you need it personally, don't assume the rest of the world does too.

Contributor: Steve Cholerton
(http://lonelyhacker.net/)

> “Do not get scared of an overly populated market segment.”

CASE #8

nBinder

nBinder, packs multiple files into a stand alone executable with over 50 advanced output and file unpack options, conditional run and commands.

Why it was judged a commercial failure: It was the first product I began selling. It sold to 300+ customers in 4 years. But for about a year the sales began to go down and have finally stopped completely.

What went wrong:

- The biggest problem was that because it was a packer intended for people that wanted to pack their products (software or games) into a single package (compressed and encrypted) many have used it for creating malware by binding malware files to legit files and then distributing the output so it isn't detected by antivirus software (although it would be detected at runtime). Because of this I had lots of problems with antivirus companies that flagged files create with nBinder as malware. This was of course affecting legit users as their files would be falsely marked as malware. I used virustotal.com to see which antivirus detected it and contacted the antivirus manufacturer as soon as I detected the problem. In most cases they would remove it from their definitions. But it was an uphill battle because it would appear again in a matter of weeks. Some small AV companies didn't event bother to reply to my emails to fix the problem. Others were using heuristics to flag files create with my applications and AV developers were reluctant to whitelist files created with nBinder. You can imagine it that it was enough for an AV such as Kaspersky or Norton to pick my files as malware for a day and customers would be affected and not use my product any more, especially that it took about 3 days for AVs to remove the false positive.
- Infrequent updates. Due to lack of time I only updated the product once or twice a year and this affected the product a lot.
- No marketing. I decided that I didn't want to invest money in marketing so, except for a short AdWords campaign, I invested no money in marketing.
- My decision to develop 3 products instead of concentrating on one or two affected development time and quality. I have worked on 3 products simultaneously instead of concentrating on making a single good one. The reason I worked on 3 is because I enjoyed developing different software in different categories. I didn't start this for money but for the fun of development.

Time/money invested: I invested almost no money (except for hosting costs). Time invested I can't really say exactly, but not too much as I only worked on nBinder in short bursts like 6 hours a day for a week or so before releases.

Current product status: Still for sale. My other products are:

- nCleaner – a free system cleaner that has gone quite well (over 2 million downloads).
- nMacro – an automation tool that has seen some limited success (bought by over 100 customers in a year or so).

Any regrets? It's not a total failure as I did make some money out of it with no investment, so I don't regret starting it, but it could have been much better.

Lessons learned: Words of advice for others trying to make money from software development:

- Study the market and the current trends very well.
- Before deciding to take on large competition make sure you have something better (at least from one point of view) than the competition (for example you might not have the same features but you have a better GUI and general presentation).
- Do not get scared of an overly populated market segment. For example with nBinder I picked a segment with very little competition but also few possible users and the results were not so great (I didn't have many users). With nCleaner I went head-to-head with lots of already established products but also the market is very big. Although nCleaner is free it has had the most success because there are so many potential users (anyone with a PC actually), so it had over 2 millions downloads and I still receive lots of mails regarding it, even if the last update was in 2007. So it is possible to have success in a market with lots of competition with no investment but it's hard to reach the level of more established products.

Contributor: Boghiu Andrei
(http://www.nkprods.com/)

CASE #9

Net-Herald

Net-Herald – a monitoring application for water supply companies. It was a complex client server application that would receive monitoring data from specialized hardware and store that data inside a SQL database. The client displays that data in different graphs, provides printable reports or sends alarm messages via SMS if a monitored value is not within its specified limits.

I developed Net-Herald as a perfect fit for that specialized hardware that is provided by a local manufacturer. That way, so I hoped, I could profit from their sales leads and would find a smoother way into these water supply companies. The downside of course, was that my software would only work with their hardware.

Why it was judged a commercial failure: I sold a first license fairly soon after I had a sellable product, although it took the customer nearly a year until they finally bought. But since then I sold only one more license within the last 4 years or so.

What went wrong:

- I didn't do my own marketing and the hardware guys weren't really concerned with selling my software.
- Water management companies have a terribly long sales cycle. Other vendors monitoring applications usually cost tens of thousands and are geared toward large suppliers. Whenever a supplier buys into such a product he is unlikely to change within the next decade or more. I tried to position my software towards small suppliers but even then most of them were already locked into another vendor's solution.
- My software only worked with a specific hardware. That narrowed the market down substantially.
- In the end the software became too complex for one poor mortal to maintain. Because the software didn't produce any substantial income I had to stop adding new features which would make it attractive for more prospective clients.
- This kind of software is not sold over the Internet. Rather it needs very active sales people that nurture clients over a rather long period of time.
- All these facts indicate that software like this should not be developed by a one man show.

Time/money invested: The development time for the first sellable version was maybe about 9 months. I didn't have a job income at that time, but got funding due to government support for small start-up businesses. So I didn't drain our family's personal finances. But I did of course invest a great deal of time and sweat.

Current product status: Now, I have drawn a line and stopped active development of Net-Herald. I still do some custom extensions for my first clients. But I no longer market the software. I have instead focused on my consulting services. I also develop and sell my cross-platform drag-and-drop product Simidude.

Any regrets? I didn't succeed yet selling my own software (which is still my goal) but I do not regret doing it. I developed Net-Herald using (Java) technologies that now give me leverage at my consulting gigs. All in all it was a heavy ride. But it was fun and I would do it again.

Lessons learned:

- My biggest mistake was the lack of market analysis. I trusted the word of the hardware manufacturer without verification.
- I have written more about the above and some other failures on my blog.

Contributor: Torsten Uhlmann
(http://www.agynamix.de/)

CASE #10

HabitShaper

HabitShaper – set and track daily targets for your goals (weight loss, quit smoking, jogging, writing, etc…).

Why it was judged a commercial failure: I sold a few copies, but not enough to make back the time I invested in it and my conversion numbers and traffic are below average.

What went wrong:

- Did not do enough pre-production research (talking to customers, etc).
- Did not do a large enough beta to make up for lack of initial research.
- Ignored gut-feeling that my product is better suited to being web-based and multi-platform (incl. mobile).
- Did EVERYTHING myself (logo, web design, video, software, AdWords, etc).

Time/money invested: I worked on it two years, part-time, while doing Masters/PhD in Physics. It had no impact on my finances (very little money invested) or circumstances.

Current product status: I am relaunching as a web-based product this summer.

Any regrets? Not in the least! I learned about as much from making HabitShaper as I have from my MSc thesis and PhD work.

Lessons learned:

- Most important: PAPER prototypes, minimum viable product, and iterate.
- Don't be afraid to launch early.
- Launch a little bigger than you'd expect (it's harder to find those initial customers than you think).
- Don't be afraid to change directions, especially early on.
- Doing things yourself is a great learning experience, but if you want to get your product out to customers as fast as possible, don't be afraid to invest money and outsource your weaknesses.

Contributor: Adriano Ferrari
(http://blog.habitshaper.com/)

> “Launch a little bigger than you’d expect.”

CASE #11

BPL

BPL – Batch Programming Language Interpreter.

Why it was judged a commercial failure: I sold about 10 copies.

What went wrong:

- I didn’t really do enough research to find out if the target market was in existence. I was hoping that network admins and support staff members would find it easier to use than batch files and less complicated than any of the free scripting language options available. So, I just rushed to get the MVP (Minimum Viable Product) out the door.
- I never did provide a compiler that would build a stand-alone EXE. I think that might have met with more success.
- I didn’t do much as far as advertising the existence of the product.

Time/money invested: I only spent a few weeks coding and documenting it in my spare time. Support issues sometimes took a whole evening, but nothing major. It did not have any impact on my finances as I had invested nothing but my time.

Current product status: I will still address support issues with this product for registered users, but I don’t actively sell it. I’ve open-sourced the program and it still really isn’t seeing heavy use.

I was more successful with other products. I have a few retired products that saw some good bulk-purchase deals (command-line DUN HangUp, command-line scheduler) and I still sell the following (for Windows):

- MailSend – Command-line SMTP mailer.
- MailGrab – Command-line POP3 reader.
- CMD2EXE – Packages up a batch file into an EXE.
- ScreenKap – Command-line screen capture.

All of the above still bring in a modest passive income.

Any regrets? Not at all. “Nothing ventured,…”

Lessons learned: Had I not attempted to bring the BPL product to life, I might still be sitting here wondering “what if?” I think it was very beneficial for me to invest the time to try out this idea.

Contributor: Jim Lawless
(http://www.mailsend-online.com/blog/)

CASE #12

Anonymous

A time tracker.

Why it was judged a commercial failure: Because it is not my primary income. I have about 150 customers in one year.

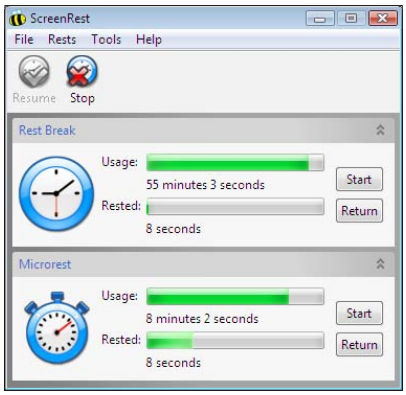
What went wrong:

- No marketing.
- No real thought into features.
- I don’t spend any time on it.

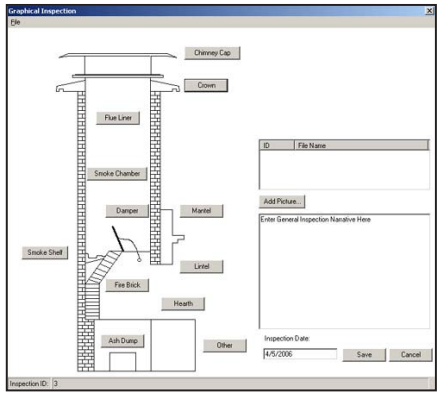
In my defense, the reason I do not spend much time on it is that the market became saturated with ‘me toos’ right after I released, which was quite expected. In fact, as I was looking for users, I got an email from a competitor suggesting that I don’t enter the market because they are working on the same thing! I don’t know what I would do differently. Maybe spend more time on it? I think the law of diminishing returns applies quite early in this space so I am not sure.

Time/money invested: Since inception (Nov 2008), I’ve spent close to 250 hours total. Total cash outlay was something like $500.

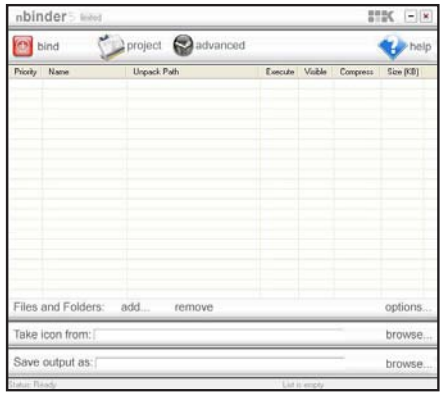
Current product status: I never tried to make it succeed, to be honest. It was



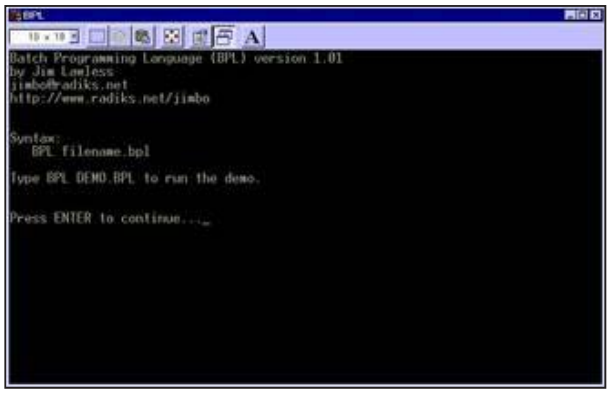
01



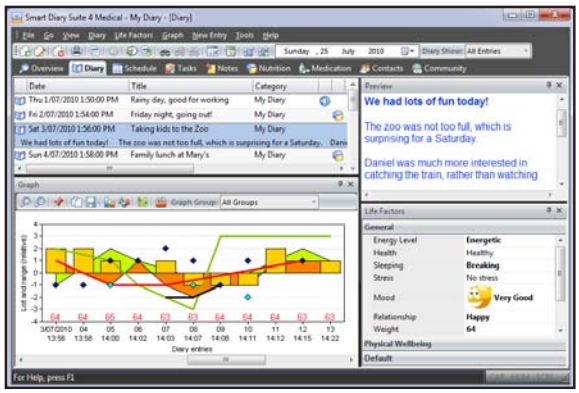
02



03



04



05

only a learning experience for me. What I probably need now is to go all in. Quite frankly, if I double the sales for this product, I can quit all consulting work. But I really do not think it is a good idea to work on this app full time as it is too simple.

Any regrets? Definitely not.

Lessons learned:

- Do it!
- Solve a problem people know they have.
- Don't invest too much time and money at the beginning.
- Don't be wedded to a particular idea.
- Don't only listen to your customers. Listen to yourself. After all, you created the idea which attracted the customers.
- Never promise a feature for a sale. I've never done it but the pressure is really great. My stock response is always: "While such a feature may be available in the future, I recommend that you only use current features when deciding on your purchase."
- Do use Google to your advantage.

Contributor: Anonymous

CASE #13

ScreenRest

ScreenRest - a consumer software product that reminds users to take regular rest breaks while using their computer.

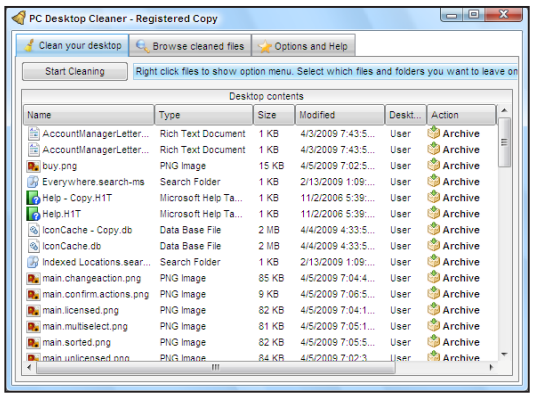
Why it was judged a commercial failure: ScreenRest failed commercially because we built a product without having a clearly defined market. This was compounded by it offering prevention, not a solution. ScreenRest continues to regularly sell a small number of licences but not in sufficient quantity to justify further enhancements. The conversion rates are good, but there are simply not enough visitors to the website.

What went wrong:

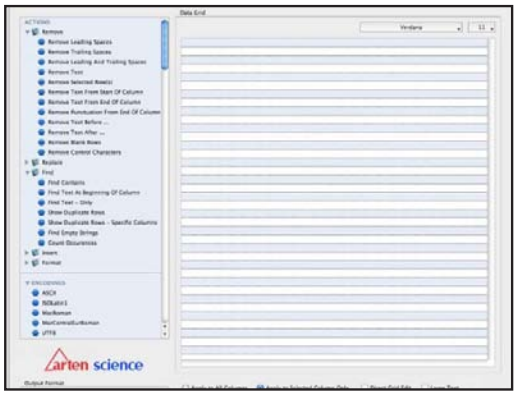
- Not doing market research first.
- Creating a prevention rather than solution product – people generally wait until they have a problem and then look for a solution.
- Creating a product with medical associations – the SEO and PPC competition for related keywords is prohibitive for a product with a low purchase price.

Time/money invested: At least £2000 was spent on the project, including software licences and additional hardware. The product and website were created over roughly 12 months by myself and my wife Lindsay, some during spare time, then part-time and finally full-time so it is difficult to determine the total number of hours. Working part-time and then full-time on ScreenRest caused a significant impact on our finances. Although right from the beginning we saw this as in investment for building a business.

Current product status: Once the product was complete and we started learning SEO it became all too apparent that organic search traffic for related keywords was going to be insufficient. Research into PPC then revealed that the price point was too low to support purchasing medical terms. Planned features for ScreenRest have been put on hold and no further marketing is planned. We continue to support new and existing ScreenRest customers and plan to do so for the foreseeable future. Rather than create another software product we chose to use what we had learned about



06



07

08

1. ScreenRest
2. ChimSoft
3. nBinder
4. BPL
5. Smart Diary Suite
6. PC Desktop Cleaner
7. R10Clean
8. HabitShaper

marketing, copywriting and SEO to create a series of websites targeting a range of topics (often known as niche sites). The most successful of these sites we are expanding in value and functionality to fill gaps not serviced by the competition.

Any regrets? No. ScreenRest succeeded in every way intended, other than commercially. Creating it was a rewarding learning exercise that started us down a path to finding the intersection of our skills, experience and market opportunities.

Lessons learned:

- Start with market research – creating a high-quality product you believe in is not enough on its own.
- Make sure you can identify a specific target market, that you can reach that market and that it is large enough to support your financial goals.

Contributor: Derek Pollard *(http://www.kimotaprime.com/)*

Conclusion

Analysing the above (admittedly small and self-selected sample) it is clear that by far the commonest causes of failure were:

- lack of market research
- lack of marketing

With the benefit of 20/20 hindsight it seems blindingly obvious that we should:

- spend a few days researching if a product is commercially viable before we spend months or years creating it
- put considerable effort into letting people know about the products we create

Yet, by my count, a whopping 6 out of 13 of us admitted to failing to do each of these adequately. Probably we were too busy obsessing over the features and technical issues so beloved of developers, which actually contributed to far fewer failures.

It is also noticeable that, despite the failure of these products, there are few regrets. Important lessons were learned and no-one lost their house. Many of us have gone on to develop successful products and the others will be in a much stronger position if they do decide to try again.

A big thank you to everyone who ate a large slice of humble pie and submitted the above. I hope we can prevent other budding software entrepreneurs making the same mistakes. Even if you don't succeed, you will learn a lot. ■

Andy Brice is a UK-based software developer with over twenty years of professional experience. He runs a one-man software product company at *www.perfecttableplan.com*, blogs at *www.successfulsoftware.net* and provides a one-day consulting package to other small software companies interested in improving their marketing and usability.

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

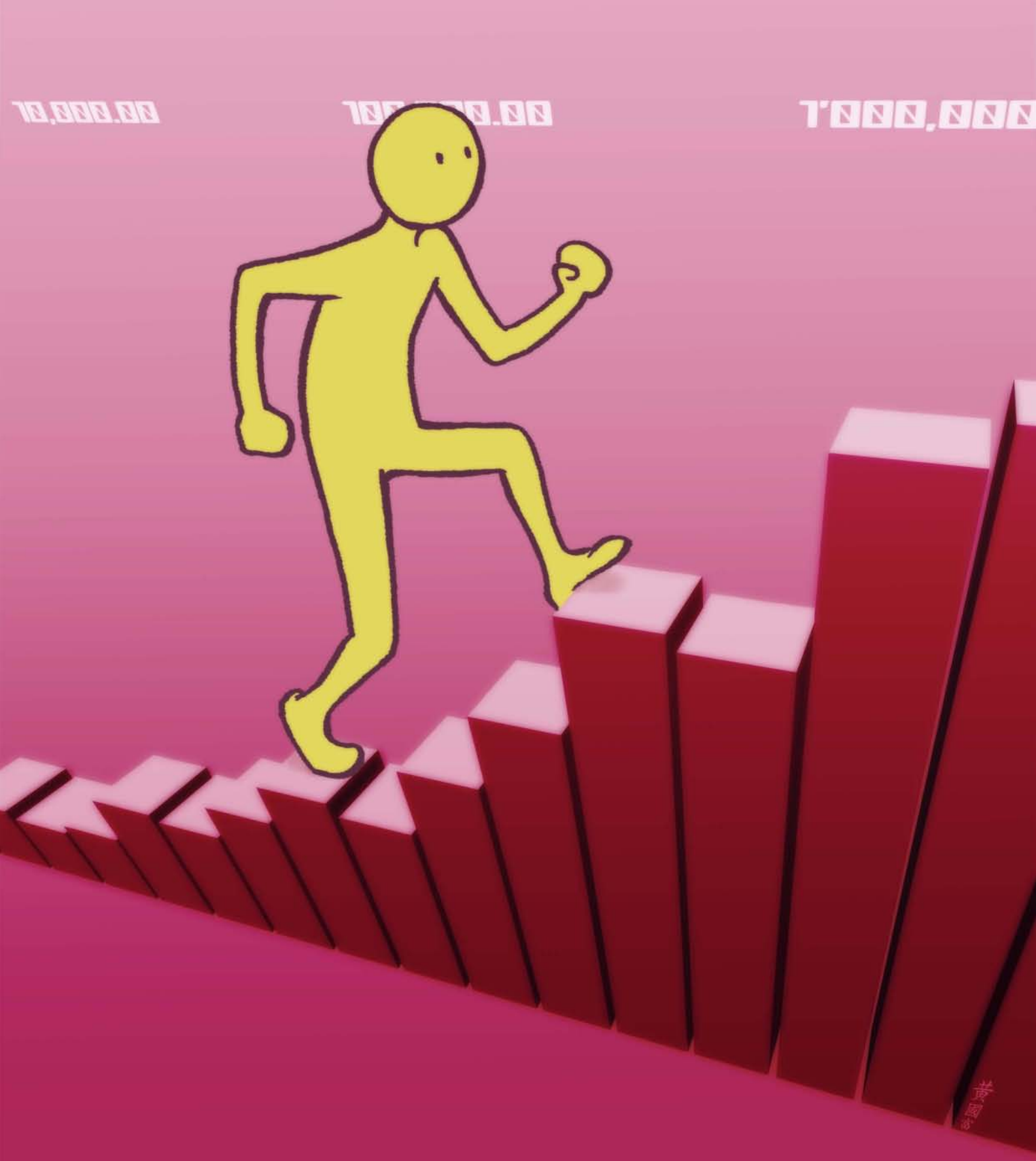


Illustration by Jaime G. Wong (*http://retrazos.pe/*)

How To Become A Millionaire In Three Years

By JASON L. BAPTISTE

> "I move forward the only direction
> Cant be scared to fail in search of perfection"
>
> — Jay-Z, On To The Next One

I'm going to go ahead and replace 3 years with a "short time frame". Some things to focus on:

Market opportunity

A million dollars is not a lot in the grand scheme of things, but it certainly is a lot if the market opportunity is not large enough. Even if you put Bill Gates and Steve Jobs as founders in a new venture with a total market size of 10 million, there is no way they could become too wealthy without completely changing the business (i.e. failing).

Inequality of information

Find a place where you know something that many undervalue. Having this inequality of information can give you your first piece of leverage.

Leverage skills you know

You can go into new fields such as say Finance, but make sure you're leveraging something you already know such as technology and/or product. Someone wanted to start a documentary with me. I said that would be fun, but it would be my first documentary regardless of what happened. There was a glass ceiling due to that. If I do something leveraging a skill I know, I'm already ahead of the game.

Look in obscure places

We're often fascinated with the shiny things in the internet industry. Many overlook the obscure and unsexy. Don't make that mistake. If your goal has primarily monetary motivations, look at the unsexy. One example would be email newsletters, which I've profiled before.

Surround yourself with smart people

Smart people that are successful usually got there by doing the same and have an innate desire to help the people surrounding them achieve the same success. It's the ecosystem that's currently happening with the PayPal mafia and can be traced all the way back to Fairchild Semiconductor.

Charge for something

Building a consumer property dependent upon advertising has easily made many millionaires, but it isn't the surest path. It takes a lot of time and scale, which due to cash-flow issues will require large outside investment probably before you are a millionaire. Build something that you can charge for. That's how business has worked for thousands of years prior to the 1990s. Make something, charge for it, repeat it. DHH explained this really well at Startup School 08.

Information products are valuable

E-Books, screencasts, and anything that can teach others to be good at something is a very lucrative business. Look at guys like PeepCode... they're killing it. There are also things like Parrot Secrets that make 400k a year. Bonus points if the information helps a person make money (directly or indirectly) or improves their self image. FYI, this doesn't mean sell snake oil ebooks. That may get you a somewhere in the 5 figures, but word will spread that your shit smells.

Your primary metric shouldn't be dollars

If you are going after a big enough market and charging a reasonable amount, you can hit a million dollars. Focus on growth, customer acquisition costs, lifetime value of the customer, and churn.

Get as many distribution channels as possible

There is some weird sense that if you build something they will just come. That a few "like" + retweet buttons and emails to editor@techcrunch.com will make your traffic explode + grow consistently. It fucking won't. Get as many distribution channels as possible. Each one by itself may not be large, but if you have many it starts to add up. It also diversifies your risk. If you're a 100% SEO play, you're playing a dangerous dangerous game. You're fully dependent upon someone else's rules. If Google bans you, you will be done. You could easily replace the SEO example with: App store, Facebook, etc.

Go with your gut and do not care about fameballing

Go with what your gut says, regardless of how it might look to the rest of the world. Too often we (I) get lost in caring about what people think. It usually leads to a wrong decision. Don't worry about becoming internet famous or appearing on teh maj0r blogz. Fame is fleeting in the traditional sense. Become famous with your customers. They're the ones that truly matter. What they think matters and they will ultimately put their money where their mouth is.

> "If you do focus on a dollar amount, focus on the first $10,000."

Be an unrelenting machine

Brick walls are there to show you how bad you want something. Commit to your goals and do not waver from them a one bit regardless of what else is there. I took this approach to losing weight and fitness. I have not missed a single 5k run in over a year. (I profiled this in my article "Hacking Calories" if you're interested). It did not matter if I had not slept for two days, traveling across the country, or whatever else. If your goal is to become a millionaire, you need to be an unrelenting machine that does not let emotions make you give up/stop. You either get it done with 100% commitment or you don't. Be a machine.

If it's a mass market "trend" that's all over the news, it's too late

This means the barriers to entry are usually too high at this point to have the greatest possible chance of success. Sure you could still make a lot of money in something like the app store or the Facebook platform, but the chances are significantly less than they were in the summer of 08 or spring of 2007. You can always revisit past trends though. Peter Cooper and I clarified some of the semantics about what is a trend over here.

If you do focus on a dollar amount, focus on the first $10,000

This usually means you've found some repeatable process/minimal traction, i.e. if you're selling a $100 product, you've already encountered 100 people who have paid you. From here you can scale up. It's also a lot easier to take in when you're looking at numbers. Making 1 million seems hard, but making $10,000 doesn't seem so hard, right?

Be a master of information

Many think it might be wasteful that I spent so much time on newsyc or read so many tech information sites. It's not, it's what gives me an edge. I feel engulfed.

Get out and be social

Even if you're an introvert, being around people will give you energy. I'm at my worst when I'm isolated from people and at my best when I've at least spent some time with close friends (usually who I don't know from business.)

Make waves, don't ride them

There was a famous talk Jawed Karim gave from YouTube. He described the factors that made YouTube take off in terms of secondary/enabling technologies. I think they included (1- broadband in the home 2- emergence of flash, so no codecs required 3- proliferation of digital cameras 4- cheap hosting 5- one click upload 6- ability to share embed). Find those small pieces and put them together to make the wave. That's what YouTube did imho. The other guys really just rode the wave they created (which is okay).

Say NO way more than you say YES

I bet almost every web entrepreneur has encountered this: You demo your product/explain what you're doing and someone suggests that you do "X feature/idea". X is a really good idea and maybe even fits in with what you're doing, but it would take you SO FAR off the path you're on. If you implemented X it would take a ton of time and morph what you're doing. It's also really really hard to say no when it comes from someone well respected like a VC or famous entrepreneur. I mean how the fuck could they be wrong? Hell, they might even write me a check if I do what they say!!!!! Don't fall for that trap. Instead write the feedback down somewhere as one single data point to consider

amongst others. If that same piece of feedback keeps coming up AND it fits within the guidelines of your vision, then you should consider it more seriously. Weight suggestions from paying customers a bit more, since their vote is weighted by dollars.

Be so good they can't ignore you

I first heard this quote from Marc Andreessen, but he stole it from Steve Martin. Just be so good with what you do that you can't be ignored. You can surely get away with a boring product with no soul, but being so good you can't ignore is much more powerful.

Always keep your door/inbox open

You never know who is going to walk through your door + contact you. Serendipity is a beautiful thing. At one point Bill Gates was just a random college kid calling an Albuquerque computer company.

Give yourself every opportunity you can

I use this as a reason why starting a company in silicon valley when it comes to tech is a good idea. You can succeed anywhere in the world, but you certainly have a better chance in the valley. You should give yourself every opportunity possible, especially as an entrepreneur where every advantage counts.

Give yourself credit

This is the thing I do the least of and I'm trying to work on it. What may seem simple + not that revolutionary to anyone ahead of the curve can usually be pure wizardry to the general public, whom is often your customer. Give yourself more credit.

Look for the accessory ecosystem

iPod/iPhone/iPad case manufacturers are making a fortune. Armormount is also making a killing by making flat panel wall mounts. WooThemes makes millions of dollars a year (and growing) selling Wordpress themes. There are tons of other areas here, but these are the ones that come to mind first. If there's a huge new product/shift, there's usually money to be made in the accessory ecosystem.

> "If this was easy then everyone would be a millionaire."

Stick with it

Don't give up too fast. Being broke and not making any money sucks + can often make you think nothing will ever work. Don't quit when you're down. If this was easy then everyone would be a millionaire and being a millionaire wouldn't be anything special. Certainly learn from your mistakes + pivot, but don't quit just because it didn't work right away.

Make the illiquid, liquid

I realized this after talking to a friend who helps trade illiquid real estate securities. A bank may have hundreds of millions of assets, but they're actually worth substantially less if they cannot be moved. If you can help people make something that is illiquid, liquid they will pay you a great deal of money. Giving you a 20-30% cut is worth it, when the opposite is making no money at all.

Productize a service

If you can make what might normally be considered a service into a scaleable, repeatable, and efficient process that makes it seem like a product you can make a good amount of money. In some ways, I feel this is what Michael Dell did with DELL in the early days. Putting together a computer is essentially a service, but he put together a streamlined method of doing things that it really turned it into a product. On a much smaller scale, PSD2XHTML services did this. It's a service, but the end result + what you pay for really feels like a product.

Look for something that is required or subsidized by law

Motorists are required to have insurance, public companies have to go through sarbox laws, doctors get tens of thousands of dollars for EHR systems, etc. Look for something that is required by law and capitalize on that. Usually things that are required and/or subsidized by law are mind numbing with complexities. Find a way to simplify that process.

Make sure you're robbing a bank

When Willie Sutton was asked why he robbed banks, he said because that's where the money is (Thanks to edw519 for this quote). Make sure whatever you're going after is where the money actually is, i.e. a customer that will pay you. Consumer markets are tough, especially with web based products. People expect everything to be free. Businesses are usually your best bet.

Don't be emotional

Emotions can let you make stupid decisions. It can make you not walk away because you're attached to something. Most importantly it will lead to indecision and a loss of confidence. Put your emotions into your product or save them for your lover, family, friends,etc.

Don't leave things up to chance

People feel that things will just work out due to carpe diem. They usually don't. People can be unreliable, deals can fall through, and shit will always happen. Prepare for multiple scenarios and contingencies. You can mitigate this by working with smart AND reliable people.

> “Reward yourself a little bit, but live as frugally as possible.”

Raise revenue, not funding

Everyone is always so damn fixated on getting funded because it's the cool thing to do. Focus on getting people to pay you at first and then scale things outwards with funding IF and WHEN you need it. If your goal is to make a million dollars in three years, funding probably isn't the way to go. VCs won't let you take a salary of ~300k per year. Selling a company in < 3 years is a crapshoot. The lifespan of an investment is usually about 7 years from what I've read.

Don't get comfortable

You will probably get comfortable somewhere around 200k, maybe less or more, but it will certainly be before 1 million dollars. If you get comfortable you start getting off balance and having the hunger to move forward. Reward yourself a little bit, but live as frugally as possible. I have friends who have made some okay money, but blow it all away on stupid shit because they got comfortable.

Look for those who are comfortable

Who is comfortable in a certain industry? Go in and knock them off their hammock so they spill their mojitos on themselves. This can also be considered stagnation. Industries often mature and people get comfortable keeping the status quo. Stagnation is the mid-life crisis for a former trend. This is usually a good point to come in with something.

Don't skimp on the important things

When it comes to things that need to be reliable such as infrastructure, delivery, or even your own personal tech equipment - don't skimp out. These are the tools that ensure reliability and your product being delivered. You can skimp on the office space, the desks, coach airfare, budget motel in mountain view, etc.

Companies spend just as much or more on services as they do on software

Paying for the ERP, CRM, or custom built system is just the first step. Then there's the maintenance, training, and service contracts.

Keep the momentum going

I've had projects where things were moving a million miles an hour, then BOOM, they just lost a lot of momentum. That is the worst possible thing you can have happen. Keep moving the ball everyday.

Listen (or read the transcriptions of) to every Mixergy interview you can

Most of my audience will probably know about Mixergy, but I can't let a single reader leave without making sure they know it exists. It is by far the most practical resource on the Internet if your goal is to do well. Andrew has interviewed entrepreneurs from all walks of life and levels of success. Most of them had real business models and bootstrapped. Most importantly, he finds out what specifically led to their success.

Last, but not least: Learn how to filter

I just wrote upwards of 2,200 words. Some of the points are even contradictory. Start adding in other sources of information and you will feel like you're being pulled in a five million directions. You will then become indecisive. Take in information and then filter the good bits while synthesizing them to be a part of your overall plan. What works for person A doesn't always work for person B. ■

Jason L. Baptiste, is currently the co-founder of Cloudomatic, which provides an easy to integrate affiliate engine specifically made for SaaS and web app developers. He is also on the board of the MIT Enterprise Forum of South Florida. You can learn more about Jason at *jasonlbaptiste.com*.

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

Why I Quit A Six Figure Job

By XAVIER SHAY

I HAD THE BEST job in the world. My immediate team of ten people were all world class, and everyday I was able to work on hard and interesting challenges with them. Hours were flexible—many of us worked seven to four (by choice!)—and there was virtually never any overtime. It wasn't unheard of to have our end of week review down at the pub. I was paid a six figure salary.

After six months, I quit.

I need to be working for a reason. Salaried work isn't necessarily a bad thing, but the benefits it provided me weren't benefits I actually wanted.

The Up

A job is easy money. This is the obvious one. The easiest, most comfortable way to get money is to work for someone else. I currently have enough assets to sustain me for about two years, and there's nothing that I want to buy, so I don't need any more money.

A job is low risk. Related to the last point but worth stating separately. You get a paycheck every month, whether or not the company makes money. This risk is shouldered by the owners of the company, and that's why they stand to gain (or lose) a lot more. I can provide my own safety net at the moment, I don't need someone else to do it for me.

A job fills the time. This isn't relevant to me, but I've heard a few times "Won't you get bored without a job?" This is so far outside my conceptual space I didn't even think of it. If you are worried about being bored without a job, first try cutting TV out of your life and see how you find ways to fill that space. A job is a TV that takes up even more time.

A job allows you to work on large challenges. The type and scale of problems you are able to work on in IT at a big company are totally different to those you have the opportunity to attack flying solo. It was a fantastic experience working on these projects, but I'm no longer feeling inspired by them.

A job allows you to work with smart people. This is actually the primary reason I accepted the job. The opportunity to work in such a high calibre team in such an environment was one that doesn't come up often outside of salaried positions. There are many other smart people I will get to work with outside of a job, but I will have to work harder to make that happen.

They're pretty fantastic benefits, and I don't regret the last six months in the slightest. None of them are particularly relevant to me any more though, and when matched with the downsides the balance is no longer positive.

The Down

A job is working on someone else's schedule. I was expected to be productive for eight hours in the middle of the day, five out of seven days a week. This doesn't match my natural rhythm. Some days I can work for fourteen hours, others I just need a day off. If I work in the mornings only, I don't need a weekend. I'm really keen to explore different modes of working to find what is most productive for me.

A job means you have to show up. Forty hours of every week were sold to someone else. That's a huge opportunity cost. I couldn't put everything on hold for a few days to chase a new idea. I couldn't use the burst of energy I get often when I had a great idea late at night, because I had to be up early the next day.

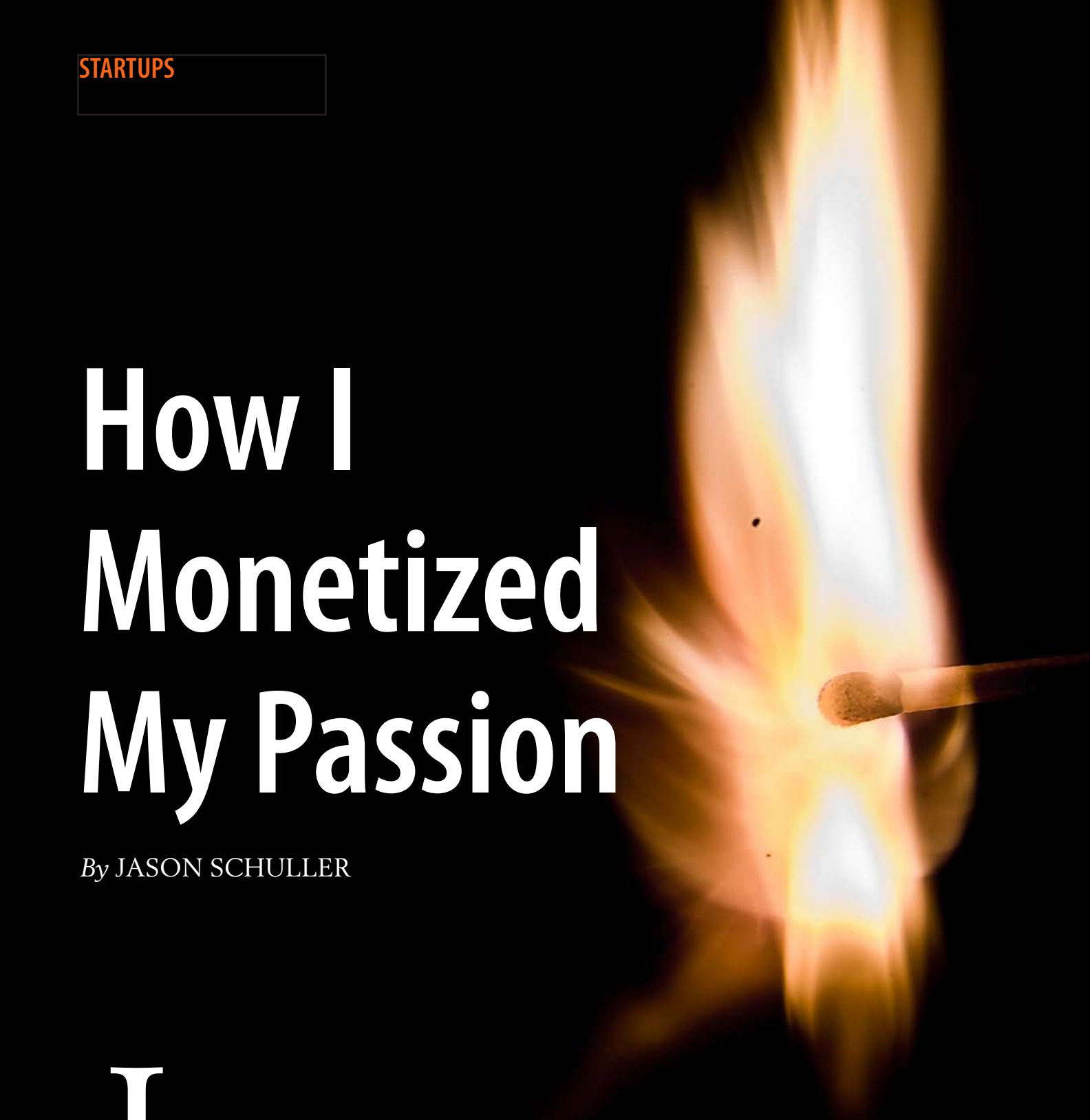
A job is working on someone else's dream. This isn't necessarily bad — helping people achieve their dreams is fantastic — but those dreams didn't align with my own.

A job is selling your time. When I'm working by the hour, there is an economic incentive to take longer to complete a task, but a professional one to be efficient. I don't want competing motivations. Why are the hours spent on a task even relevant? I want to sell value rather than my time.

So what am I going to do? For a large part, I don't know, and that's kind of the point. I have a few projects I'm working on — A tour of the US and this blog being two major ones — but now the biggest benefit is that I'm free to say yes. Yes to projects, yes to schemes, yes to travel, yes to "let's stay up on a weeknight and watch B-grade sci-fi." I don't want the best job in the world, I want the best life. ■

Since quiting his job, Xavier Shay has taken up beat boxing, performed as a life-sized giraffe, become a dance teacher, organized a tech training tour through the US, and started a blog with his brother Jared about personal development and being more awesome at *www.two-shay.com*.

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



How I Monetized My Passion

By JASON SCHULLER

Just over 2 years ago, I was sitting in what seemed like an ever-shrinking cubicle at a major Seattle based company making updates to websites for upper management. I suppose I made a decent (average) living, and my job was secure, but at the same time I felt that I was under-challenged, under-utilized and ready for a major change in my life. Needless to say, I thought about quitting my job more than once per day (sound familiar?), but to what end?

There were countless times during my years at that company when I tried to make a difference by presenting alternative ways to implement and manage their internal network of websites. In the end, all of these ideas were either dismissed or put off on the basis that these "alternatives" were unknown, untested and unsupported open source technologies such as Joomla, WordPress, etc. I specifically remember one manager telling me that my ideas sounded amazing, but a little "wild and crazy" for the company.

> “I had a passion for web design and I was determined to turn that passion into a career.”

Making the Decision to Change

It was pretty much at that point that I realized if I didn't at least try and do something different, I would wake up one day (years later) in that same ever-shrinking cubicle working the same dead end job. I was ready to step up to the plate.

As much as I wanted to, I knew that I couldn't just walk into work one day and quit on the spot. I had responsibilities – a wife, a house and bills to pay. With that in mind, I researched and then approached my manager about taking a 2 month “sabbatical”…a trial run if you will. Basically a leave without pay, but at least I would still have a job if I needed one at the end of those two months. Management signed off on it (no questions asked) and off I went into the unknown.

Getting Your Feet Wet

My plan was simple… I had a passion for web design and I was determined to turn that passion into a career. The first step was to get my name out there, and I figured the best way to do that was to start a blog and begin writing about web design, development and other related topics. Please keep in mind that my skills as a designer/developer were all self taught to that point (still are actually), so I really was not all that confident about doing this – but I had to try.

WordPress seemed to be a trendy topic at the time, so in January of 2008 I launched a site called WPelements.com and started blogging about WordPress. Really, all I was doing was writing about things that I myself was learning at the time as I played around with the platform. I spent hours on end every day working with WordPress, reverse-engineering themes and tweaking code. WPelements.com gained some traction right after I released my first free theme called “Massive News” which was downloaded a few hundred times within the first week. Right after Massive News, I released my first WordPress plugin called the “Featured Content Gallery” which was also an instant success. I remember thinking that this was my ticket into something new, a step toward that “big change” in my life that I was searching for. Soon after I released Massive News, the emails started rolling in from people looking for custom WordPress development services which is how I started my (short) freelance career.

By the time my two month sabbatical was up, I was confident enough to walk back into work and put in my two weeks notice. Actually, what I said was: “I am prepared to give you two weeks, but if you can let me go in a week that would be great because I'm really really busy.”

Trial and Error

During the first two months of 2008, I was able to build enough of a name for myself where I could sustain my income by doing freelance WordPress design and development work. However, it was about that same time that I realized that I still was not quite happy with what I was doing for a living. Basically, it was the exact same thing I was doing at my previous job, just with clients instead of managers. Back to square one. Something needed to change once again just two months into my entrepreneurial career.

The WordPress community was growing, and I took note of a trend which was on the rise… “Commercial WordPress Themes.” Brian Gardner pioneered (or at least was one of the first to pioneer) the idea of selling commercial WordPress themes in August of 2007 with a theme called “Revolution” (now StudioPress.com). Shortly after Brian, a few others popped up selling their own themes as well including Adii with his “Premium News Theme” (now WooThemes.com). To say the least, the idea of creating a theme and selling it as a commercial product perked my interests. I remember emailing both Brian and Adii about their businesses looking for tips on how to get started. Surprisingly, both of them already knew about me and what I was doing with WPelements.com, and gave me the inspiration to try selling some themes of my own. Let me say that Brian and Adii are some real stand-up guys who I am happy to consider my friends even though we are each others competition.

> “Money is not a bad thing, but it really should not be your means to happiness.”

Then Success

I had created a site called TrailerFlick.com in December of 2007 which never became popular, but there was always interest in the site design by random users who just happened upon it one way or another. I created TrailerFlick.com to provide an alternative method of viewing movie trailers, and the design was simple… just a grid of movie posters that when clicked would display the trailer in a pop-up window. This was actually the first live website I had ever built entirely on WordPress. In February of 2008 I had a client that found TrailerFlick.com and wanted a WordPress theme based on the design for his own movie production studio. I spent a week tailoring the theme for this client who in the end never paid up. I decided that this would be a good candidate for my first commercial theme, so I cleaned up the code, called it “Video Flick” and threw it on WPelements.com for just $5.00 per download. I just want to take a second to thank that client for never paying his tab.

The interest in Video Flick blew me away, and I immediately knew that had something on my hands that I could build into a real business. One theme at $5.00 per download was definitely not enough to make a living, but it was good extra cash to throw on top of the freelance work I had at the time. As the months rolled by, I released two more video-centric WordPress themes (TV Elements followed by Video Elements) and started charging $25 a piece. By June of 2008 I knew there would be no looking back and that I had a substantial business on my hands. Not many theme developers (if any) were creating video-centric WordPress themes at that particular time, and I think that releasing a commercial solution for video was the key to growing my business as fast as I did.

At that time, I was still doing freelance work and blogging about WordPress on WPelements.com, but I finally decided that neither blogging or freelance work were really what I wanted to do which is why I designed a simple theme store and moved all my commercial themes over to Press75.com separating my theme business from my freelance business and blog. By August of 2008, I had doubled my income on Press75.com with only 4 themes at $50 a piece. This allowed me to completely close the doors on freelance work to focus 100% of my efforts on commercial WordPress themes.

Nearly one year and about a dozen themes later, Press75.com continues to grow and I just launched a second site called ThemeGarden.com in hopes to expand beyond my own personal brand. Needless to say, I could not be happier with what I do for a living. I get to design and create WordPress themes that are used by thousands of people around the world, and the best part is that the only one telling me what to do and how to do it – is me.

Let Your Passion Drive You

I really don’t consider myself any sort of talent when it comes to writing, but if you have stuck with this story to this point, a few more minutes aren’t going to kill you. I didn’t write this for recognition, or to brag about what I consider my own personal success. Honestly, I’m sure most of you don’t even know who I am or what I do, nor do you probably care. The whole point of writing this article was to share with you that life is in fact, what you make of it. If you want to change, there really is nothing stopping you from doing so. That is not to say that change is at all easy, or something that will happen overnight. It took me more than 10 years of working at a company, developing skills, experimenting with different ideas and just growing up before I found the confidence to really go after my passion.

Also, it is my strong belief that if making money is your only goal in life, you will probably spend the rest of your life chasing that goal and never end up where you want to be. I realize that the title of this article is “How I Monetized My Passion”, but what I really mean by that is money can sometimes become a by-product of chasing your passions. Money is not a bad thing, but it really should not be your means to happiness. When I left my day job two years ago, money was never my end game, and I hope it’s not yours when/if you decide to make a major change in your life. Let your interests and your passions be the driving force behind change in your life – I did. ■

Jason Schuller is a digital creative professional living and working in Seattle Washington. He primarily designs and builds themes and plugins for the popular WordPress publishing platform which can be found on *Press75.com*.

What Kind Of Girl Do You Think I Am?

By NIKOS MORAITAKIS

Don't try this at home:

> ***Guy:*** *"If I gave you a million dollars, would you sleep with me?"*
> ***Girl:*** *"A million dollars is a lot of money, and you don't look that bad, so I guess I would consider it"*
> ***Guy:*** *"Ok, since I don't have a million dollars, would you sleep with me for $100?"*
> ***Girl:*** *(outraged) "What kind of girl do you think I am?"*
> **Guy:** *"We've already established the answer to that question. Now we're just negotiating the price"*

Without the million-dollar question, moral choice is easy for this girl. Yes, she won't sleep with someone for $100 and that doesn't take a lot of thinking. She's just not that kind of girl. The million-dollar question is interesting because it forces her to really decide what kind of girl she is.

If a principle is subject to revision past a price point, then this is no longer a question of principle, but a question of finding the price point. Surely not $100, but possibly less that a million.

How many business decisions have you taken without the privilege of considering the million-dollar option, and its moral consequences? If you're like most people, it's happened to you several times and you weren't even aware of it.

I think of this little anecdote every time a cost/benefit question comes up at work. It's not uncommon in business to be faced with the opportunity to do something that is against your standards and professional practices, but would bring in some extra revenue. In such circumstances, it's typical for the people involved to debate whether it's "worth" doing X or not. And, inevitably, the "worthiness" of the action is measured in dollars. Sure, no big company will put its reputation or professional standards at stake for $10,000 (if they're at all serious about their business) but for an amount that has an actual impact on its bottom line, morality gets fuzzier.

So here's a handy tool that may, in some cases, bring more clarity to your decision-making:

1. Would it be easier/harder to decide if the amounts at stake were dramatically lower/higher?
2. If so, then what kind of girl are you?

Legislators and moral philosophers have invented fancy terms for the two possible answers to the "what kind of girl are you"question.

Consequentialist reasoning suggests that morality is context-sensitive, in the sense that the results of an action have a real effect on our judgment of its morality. Torture is morally reprehensible, but what if torturing one man can give you information to save the lives of thousands? In other words, the potential outcomes of an action, i.e. what's at stake, need to be considered in moral judgments.

The girl of our story is a consequentialist kind of girl. (if it bothers you that the difference between the options is just a bit more money, consider a case where instead of a million dollars she was offered the formula for a drug that will cure cancer for all humanity – and consider whether she should feel equally or less "dirty" for sleeping with the guy to get it as with the $100)

Categorical reasoning on the other hand suggests that the morality of action is independent of results. The morality of exchanging sex for money is the same regardless of the amount. We can debate if it's good or bad, moral or immoral, but we would not be prepared to revise our moral judgment based on what's at stake. In other words the girl should happily take the $100 assuming that this is a fair value for the time she will spend on the activity.

The philosophical debate is in no way decided (it wouldn't be a philosophical debate otherwise) and that is really why making decisions generally sucks. In each case, we have to first decide whether we are a consequentialist or a categorical girl, whether we will always be that kind of girl, or whether we will take some decisions categorically (and set up "commandments", "golden rules" and "honour codes" to defend from consequentialist creep) while allow for consequentialist thinking in others (and use methods such as "cost-benefit analysis" and "risk valuation" to determine our moral inflection point).

I find it helpful, before I consider a dilemma, to at least debate whether I'm in that girl's situation, and what kind of girl I'm going to be for this particular question. Imagining different stakes for the available options is perhaps one of the easiest ways to abstract any business decision to a level where the consequentialist/categorical dichotomy is obvious. ■

Nikos Moraitakis is vice president of business development at Upstream. Follow him on Twitter at @moraitakis.

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

How I Almost Ignored Our Single Best Source For Customer Feedback

By HILLEL COOPERMAN

Back in mid-2009 when we were building A Story Before Bed a children's books online service for its eventual launch in the fall of 2009 we had a talk about how to support our eventual customers. I remember reading a blog post (which I can't find now) about how putting an 800 number on your website made people much more willing to give you their credit card numbers. We decided that having free 1-800 tech support for our site was going to be a differentiator for us. It's not often you find a consumer website these days that provides that level of support. Typically if there even is a phone number it's buried under layers and layers of FAQs, knowledge bases, and e-mail forms. It often seems like companies will do anything possible to avoid actually speaking to a customer. I've experienced this many times as a customer and I know how it makes me feel. Like crap. And yet, as a business owner, I read all this reluctance as an indicator of how costly and time consuming it is to provide person-to-person customer support. I was nervous.

At first I suggested that the 1-800 number would ring my cell phone. This wasn't some altruistic desire to connect with customers, but me being cheap. My partner Walter laughed at me. He pointed out this would not be a good use of my time as we would no doubt be inundated by calls, and I had lots of other stuff to do. I was a little embarrassed, but he's annoyingly right almost all of the time. I spent months looking for firms to which I could outsource our phone support. I finally found one in the Philippines. Our operator was very nice. She was dedicated to our product. And I could chat with her over IM, even when she was on call (to which I could listen in on). Her attitude was just wonderful, but there was no way she could know the product the way I did. She also couldn't know how much we cared about making our customers happy. One day I discussed with her when to give a refund. I told her we had a no questions asked 7 day refund policy. She asked what to do if the person wanted a refund on day 8? I told her to go ahead and give it anyway. There were a lot of situations like this that had to be spelled out. To the letter.

We launched, and she handled calls. She definitely did her best, and it was great to know that our customers had someone they could rely on. And while we didn't have a huge number of customers, we woefully overestimated how many support calls they would generate. It's not that our product was perfect. It definitely wasn't. It's just that while the 800 number may have made people feel comfortable using the site, for the most part, they didn't use it. My rough calculations show about one support call out of every 100 registered users, if that. After a couple of months of paying an incredible amount of money to handle the handful of calls we decided to bail and go back to the original plan. We set up a new 800 number that rings straight to my cell phone. Caller ID lets me distinguish between my mom calling and a customer needing help. And now, every few days, I get a phone call from a customer who has a question about our service.

“When customers call, their questions and feedback are essentially a free focus group.”

When I used to work at a large software company, I couldn't imagine many jobs worse than being a tech support person. Perhaps it was my own interaction with support folks stuck supporting products they almost never had control over, and often didn't have enough expertise in. Or maybe it was all the effort that companies make to avoid being on the phone with customers in the context of support that made me assume it's something to be avoided. It turns out that answering our support calls has been an incredibly productive experience as well as potentially a profit center. When customers call, not only am I in a great position to help them as I understand the product inside and out, but their questions and feedback are essentially a free focus group. We always have a list of improvements we need to make to the product, but sometimes prioritizing can be a crapshoot. Vocal customers tell me quickly which work items need to move to the top of the list. I can only imagine how many customers of ours experience the same frustration as these callers but don't bother picking up the phone. I think of our support callers as unelected representatives of our customer population. Each of them represents a non-trivial number of users who (understandably) didn't have the time to call us.

Not only do I get great information that I can empathize with from these customers, but recently I've started finding out how effective our marketing is – "Do you mind me asking where you heard about A Story Before Bed?" and turning each support call into a gentle sales call – "Did you know about our subscription offer? It could save you a lot of money." I realize these things may be obvious to many of you reading this post, but even if I understood them intellectually, I didn't *really* understand them, at an emotional level. It's still early, but it looks like answering calls may not only not be a drag on the bottom line, but a boost.

And while the frequency of calls is on the rise as our site gets more popular, for now, handling the calls isn't just 'not a problem' it's something I look forward to. It makes me understand why Craig's (a.k.a. Craigslist Craig) main job is customer support. From my perspective, there's no better way to understand what my customers are thinking. Analytics can tell me what they're doing, but not why. When the calls are frequent enough to impact my other responsibilities, I honestly wonder which of my tasks I'll delegate. More and more I think that someone else might be flying to New York to sign up new publishers, and I'll stay focused on answering calls and e-mails.

A Story Before Bed. This is Hillel. How may I help you? :) ■

Hillel Cooperman is one of the founders of Jackson Fish Market, a bootstrap startup from Seattle. They have shipped several consumer experiences for the web and mobile devices including A Story Before Bed (*www.astorybeforebed.com*) that lets parents and grandparents record a video of themselves reading a digitized children's book, and lets kids play it back in the web browser or on the iPad or iPhone synchronized to the pages of the book.

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

What Are The Biggest Legal Mistakes That Startups Make?

By SCOTT EDWARD WALKER

My buddy and I are coding up a new site and we will be ready to launch the beta in about a month. We have a couple of angel investors who are interested, and we don't want to screw anything up. What are the biggest mistakes that you've seen guys like us make?

Here are ten quick ones (in no particular order):

1 **IP Ownership.** Some entrepreneurs make the mistake of creating IP for their new venture while they are still working for someone else. They then quit and launch their startup, not realizing that the IP is actually owned by their prior employer. This is a tricky issue, and you should carefully review all employment-related agreements to determine if there are any provisions that may inhibit your new venture, including IP ownership. I discuss this issue in detail in paragraphs 2 and 4 of my blog post regarding formation issues (part 2).

2 **Choice of Entity.** Some entrepreneurs make the mistake of forming the wrong entity. Investors generally invest only in corporations – not LLC's or partnerships. You should thus form a corporation – and consult with an accountant as to whether you should make an S corporation election (and then convert to a C corporation down the road). I discuss the issue of choice of entity in detail in my blog post "Choice of Entity for Entrepreneurs."

3 **Place of Incorporation.** Some entrepreneurs make the mistake of incorporating the company in the wrong state. You should incorporate in Delaware – that's what investors generally require. You should then qualify the company to do business in California and/or any other State in which it is "doing business." I discuss this issue in paragraph 1 of my blog post regarding formation issues (part 1).

4 **Vesting Restrictions.** Some startups make the mistake of issuing stock to co-founders without imposing vesting restrictions. Then, one of the founders ends-up leaving in a few months and keeps all of his or her equity. You should make sure you and your co-founder execute a restricted stock purchase agreement with reasonable vesting schedules (typically four years) upon the issuance of the company's stock. I discuss this issue in detail in my blog post "Founder Vesting: Five Tips for Entrepreneurs."

5 **Securities Law Issues.** Some startups make the mistake of not complying with applicable securities laws; for example, they issues shares to "friends and family" who are not "accredited investors" without proper disclosure documents; or they retain a consultant who is not a registered "broker-dealer" to sell company stock for a commission. You should be very careful when issuing any kind of securities; non-compliance could cause severe consequences, including

a right of rescission for the securityholders (i.e., the right to get their money back, plus interest), injunctive relief, fines and penalties, and possible criminal prosecution. I discuss these issues in detail in paragraphs 2 and 4, respectively, of my blog post "Five Common Mistakes Entrepreneurs Make in Raising Capital."

6 **Splitting Equity.** Some startups make the mistake of splitting equity equally between or among the co-founders. The splitting of equity is a significant business decision which must be negotiated between or among the co-founders based upon their respective contributions to date and their expectations going forward. Simply dividing the shares equally may sound fair on its face, but it's usually not the correct decision. I discuss this issue in detail (and the various factors to consider) in my blog post "Ask the Attorney – Splitting Equity."

7 **Employment Issues.** Some startups make the mistake of not addressing employment-related issues with respect to new hires. For example, if an employee is hired by a startup, he or she generally should be required to execute two documents: (i) an offer letter and (ii) a confidentiality and IP/invention assignment agreement. The offer letter will set forth all of the employee's respective rights and obligations, including position, compensation (including stock options and/or other incentive compensation), benefits and, most importantly, whether the relationship is "at will." The confidentiality and IP/invention assignment agreement is designed to prevent disclosure of the company's trade secrets and other confidential information and to ensure that any IP developed by the employee is legally owned by the company. I discuss this issue in paragraph 8 of my blog post "Launching a Venture: Ten Tips for Entrepreneurs."

8 **83(b) Elections.** Some founders make the mistake of not making an "83(b) election" in connection with the restricted stock (i.e., stock subject to forfeiture) issued to them. Section 83(b) of the Internal Revenue Code permits the founders to elect to accelerate the taxation of restricted stock to the grant date, rather than the vesting date. As a result, the founder would pay ordinary income tax rates on the fair market value of the stock at the time of the grant (which presumably would be quite low or would be equal to the purchase price if such stock was purchased), with any subsequent appreciation of the stock being taxed at capital gains tax rates upon its sale. Such an election is made by filing the appropriate IRS form within 30 days after the grant/purchase date (no exceptions applicable). I discuss this issue in detail in paragraph 3 of my blog post "Founder Vesting: Five Tips for Entrepreneurs."

9 **Due Diligence.** Some startups make the mistake of not diligencing the guys or gals on the other side of the table. Indeed, whether a startup is doing a financing, a partnering agreement or some other transaction, it must investigate the other party or parties involved. This means determining the reputation of both the company/firm (if it's not a marquee name) and the particular individuals with whom it is dealing. Who are these guys? Are they good guys or are they jerks? Can they be trusted? When they say they are going to do something, do they do it? Do they add value? Remember, in certain deals (such as an angel or venture capital financing), the startup will, in effect, be married to the firm and the individuals for a number of years. I discuss this issue in paragraph 1 of my blog post "Five Mistakes Entrepreneurs Make in Dealmaking – Part I."

10 **LegalZoom.** Finally, some startups make the mistake of using LegalZoom or other sites to prepare their legal documentation. Websites like LegalZoom are not law firms and do not render legal advice; nor are they able to create the kind of sophisticated documents that you need to protect yourself and to demonstrate credibility with your prospective investors. You should retain an experienced corporate lawyer to help you from the legal side. I discuss this issue in detail in the FAQ's section of my website.

Conclusion

I hope the foregoing is helpful. I realize it's a lot of information to digest; however, I see these mistakes made by startups all the time. ■

Scott Edward Walker is the founder and CEO of Walker Corporate Law Group, PLLC, a boutique corporate law firm specializing in the representation of entrepreneurs. Scott has built a strong team of lawyers, with offices in Los Angeles, San Francisco and Washington, D.C. You can follow him on Twitter as @ScottEdWalker or check out his blog.

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

Say Hello to My Little Friend

How I Became the Tony Montana of the Internet

By DAVE PELL

Back in the early days of the web I was just a dealer. And I followed the advice I got from the movie Scarface: Don't get high on your own supply. I used the web as a tool to be more efficient at achieving goals I had set for myself in the outside world. I blogged, I created sites, I worked with a bunch of interesting startups.

Don't get me wrong. I dabbled in the web as a user. But it was always with the bigger picture in mind. It was always with a purpose. I was in charge. I was in control.

Those days are over. Like Tony Montana, I didn't follow the advice about getting hooked on the product. As the realtime, social web has erupted, so too has my transition from being a dealer to being a dealer and a hardcore user. I've been denying this reality for years. I easily convinced myself that I wasn't the Nurse Jackie of the internet. I told myself I was just taking a little taste to make sure I understood the product I was serving out to others — the civilians, the suckers. But it was a lie.

The other day, after spending my usual ten to twelve hours in front of this laptop I decided to restart my machine. I checked my email. I refreshed my Tweetie. I double-checked Facebook. I loaded Google Reader to make sure I was entirely up to date on all the news from the latest Afghanistan troop levels to the attempts to stop the gallons of crude from bubbling into the Gulf to the current quotes from the Mel Gibson tapes to the latest reactions to Antennagate. Finally, after a quick check of my realtime blog stats, I took a deep breath and pressed the restart button.

Within five seconds, I picked up my iPhone and checked my email.

Suddenly self-aware, I paused. I looked at my sweat-beaded reflection in the still darkened laptop screen and I realized that yes, I am high on my own supply. I used the next couple minutes of restart time for some personal reflection about the way the internet now controls me and how, as I've written here before, I went from using a tool to being one.

A few weeks ago I was hosting my son's fourth birthday party at an old school arcade. We were running short on quarters, so I went to throw a few dollars in the change machine. While I waited for my bills to become change, I pulled my iPhone out of my pocket and checked my email. It was Sunday morning. It was my son's birthday party.

I often fall asleep to audiobooks. That leaves my iPhone on my nightstand. Recently, while my wife and I spent the sunrise hours cuddling and joking with our kids, I heard the vibration of an incoming email. I rolled over and checked it.

In the last year, I haven't driven a commute of more than 15 minutes — or walked more than five — without opening at least one app on my iPhone.

Last weekend, everyone in my house heard what sounded like a deep breathing sound in our kitchen. Then I open

the door and I heard it in the backyard too. I started to get nervous. It was the kind of sound that would provide an appropriate backdrop to a horror movie that was just about to get scary. I walked to the front of my driveway. I explored the garage. I put my ear to heating ducts and water pipes. Everywhere, I heard the sound. Inhale, exhale. Inhale, exhale. I ran back into the house to tell the kids to pack a bag, we were getting out of there. Then my daughter pointed to my pocket. I reached in and pulled out my iPhone on which I had inadvertently opened the Balloonimals app which makes a blowing noise until you start the game.

Suddenly I knew what Tony Montana meant when he said, "Say hello to my little friend."

At the moment, I felt stupid. But then I realized that the breathing was real. My iPhone is alive. I hear it breathing right now. Do you hear yours?

I went from being the Tony Montana who came to Miami with nothing and worked his way to the top through a combination of sheer will, toughness and a knack for avoiding chainsaws, to being the Tony Montana who was unconsciously fantasizing about his sister and yelling obscenities to an empty room while soaking neck-deep in a cocaine-fueled bubble bath.

The realtime web has become a habit. It's a twitch. I do it without thinking. More importantly, when I succumb to the reflex of checking it every few minutes or seconds, I do so at the expense of thinking. When is the last time you stood in line at a bank without checking your iPhone? What about waiting for a long stoplight or sitting at a restaurant counter? Those moments, now dominated by the internet reflex must have been used for something else before all this technology climbed into our pockets. What were we thinking about when we had all that extra time?

I don't remember. But I'm pretty sure it was more important than all these updates I habitually check.

When the WiFi went down during the official iPhone 4 demo, didn't you sort of wish Steve Jobs would turn to the crowd and say, "You know what, let's just talk."

But that could have never happened. We know from his late night email exchanges with customers that Steve Jobs is no longer just a dealer either.

Is there a pill for this twitch or a salve to slow this reflex? I don't know. While I search, I hear the constant repetition of an updated version of another Scarface quote.

You gotta make the money first. Then when you get the money, you get the power. Then when you get the power, then you get the women.

Then you get the iPhone. ■

Dave Pell writes Tweetage Wasteland, Confessions of an Internet Superhero. He is web entrepreneur and investor and lives in San Francisco.

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

Advanced Programming Languages

By MATT MIGHT

STUDENTS OFTEN ASK for a recommendation on what language they should learn next. If you're looking for a job in industry, my reply is to learn whatever is hot right now: C++, Java and C# — and probably Python, Ruby, PHP and Perl too.

If, on the other hand, you're interested in enlightenment, academic research or a start-up, the criterion by which you should choose your next language is not employability, but expressiveness. In academic research and in entrepreneurship, you need to multiply your effectiveness as a programmer, and since you (probably) won't be working with an entrenched code base, you are free to use whatever language best suits the task at hand.

Here you'll find descriptions of four good languages to learn — Haskell, Scala, ML and Scheme — with a list of my favorite features for each, and pointers on where to learn more.

Of course, this short list is by no means exhaustive. There are many uncommon languages that excel at niches. To name just a few more, there's also D for systems programming; Erlang or Clojure for concurrency; and Datalog for constraint programming. Then there are languages like Smalltalk — alternate yet fully capable universes that branched off from mainstream computing long ago.

I encourage my students to never stop learning niche languages. They expand your modes of thinking, the kinds of problems you solve quickly and your appreciation for the meaning of computation.

Haskell

Haskell excels as a language for writing a compiler, an interpreter or a static analyzer. I don't do a lot of artificial intelligence, natural-language processing or machine-learning research, but if I did, Haskell would be my first pick there too. (Scheme would be a strong second.) Haskell is the only widely used pure, lazy functional programming language.

Like Standard ML and OCaml, Haskell uses an extension of Hindley-Milner-style type inference, which means that the programmer doesn't have to write down (most) types, because the compiler can infer them. It has been my experience that it is difficult to get a bug through the Hindley-Milner type system. In fact, experienced programmers become adept at encoding correctness constraints directly into the Haskell type system. A common remark after programming in Haskell (or ML) for the first time is that once the program compiles, it's almost certainly correct.

As a pure language, side effects (mutations of variables or data structures and I/O) are prohibited in the language proper. This has forced the language's designers to think seriously about how to provide such functionality. Their answer, monads, enables one to perform side effects and I/O inside a safely constrained framework. Naturally, Haskell lets users define their own monads, and now the programmer has access to monads for continuations, transducers, exceptions, logic programming and more.

Aside from being pure, Haskell is also lazy. That is, an expression in Haskell is not evaluated until (and unless) its result is required to make forward computational progress. Some have argued that the promised efficiency gains from laziness haven't materialized, but that's not of concern for me. I appreciate laziness for the increase in expressiveness. In Haskell, it is trivial to describe data structures of infinite extent. Where other languages permit mutually recursive functions, Haskell permits mutually recursive values.

More pragmatically, I have found laziness useful in encoding option types, where utilizing the empty case should always nuke the program. In Haskell, you can avoid creating an option type and instead use error to produce the empty value. Because of laziness, every type in Haskell automatically has two additional values: non-termination and error. Used well, this eliminates much tedious pattern matching.

My favorite feature of Haskell is type classes. Haskell's type system allows the compiler to infer the correct code to run based on its type context, even when that type context is also inferred. The example of type classes that got me excited was bounded lattices. A bounded lattice is a mathematical structure that has a least element (bot), a greatest element (top), a partially ordered less than relation (<:), a join operation (join) and a meet operation (meet).

In Haskell, one can define a bounded lattice as a type class:

```
class Lattice a where
   top  :: a
   bot  :: a
   (<:) :: a -> a -> Bool
   join :: a -> a -> a
   meet :: a -> a -> a
```

This says that if type a is a Lattice, then a supports the expected operations.

What I really love about Haskell is that it lets the programmer define conditional instances of a class; for example:

```
instance (Ord k, Lattice a) =>
                Lattice (Map k a) where
   bot = Map.empty
   top = error $ "Cannot be
                          represented."
   f <: g = Map.isSubmapOfBy (<:) f g
   f `join` g = Map.unionWith join f g
   f `meet` g = Map.intersectionWith
                               meet f g
```

This rule says that if the type k is an instance of an order (class Ord) and the type a is an instance of a lattice, then a map from k to a is also an instance of a lattice.

As another example, you can easily turn the Cartesian product of two lattices into a lattice:

```
instance (Lattice a, Lattice b) =>
                    Lattice (a,b) where
   bot = (bot,bot)
   top = (top,top)
   (a1,b1) <: (a2,b2) = (a1 <: a2) ||
                (a1 == a2 && b1 <: b2)
   (a1,b1) `join` (a2,b2) =
          (a1 `join` a2, b1 `join` b2)
   (a1,b1) `meet` (a2,b2) =
          (a1 `meet` a2, b1 `meet` b2)
```

It's easy to make the "natural" lifting of the lattice operations, relations and elements to almost any data structure. The end result is that if you use the expression bot or the relation <: anywhere in your code, Haskell can infer, at compile-time, their "appropriate" meaning based on the type of the expression (which it can also infer).

The ML languages have functors to play the role of type classes, but they lack the ad hoc polymorphism support of Haskell's type classes. Having spent a considerable amount of time programming in the MLs and in Haskell, the practical ramifications of inference on expressiveness cannot be understated.

Favorite features

- Type classes.
- A rich library.
- Monads.
- List comprehensions.
- Compact, readable, whitespace-guided syntax.

Standard ML and OCaml

The ML family is a sweet spot in the language-design space: strict, side-effectable and Hindley-Milner type-inferred. This makes these languages practical for real-world projects that need high performance and stronger guarantees of correctness. The ML family has gained traction with aerospace engineers (for its support of bug-free code) and with programmers in the financial industry (for the same reason). Standard ML was the first functional language I learned well, so I still remember being shocked by its expressiveness.

Today, OCaml seems to be the popular ML to learn, but there is at least one convincing argument in SML's favor: MLton. MLton really delivers on the thesis that functional languages offer the best opportunities at optimization. As a whole-program optimizing compiler, I've yet to see another compiler match its performance. I once created OpenGL bindings for MLton to toy around with 3D graphics, and the resulting program ran faster than the C++-based model I had used as a reference, with just 10% of the code.

The functor system in SML, while more verbose than Haskell's type class system, is more flexible. Once you instantiate a type class T for a kind/type k in Haskell, you can't instantiate that type class again for that kind/type. With functors, each instance gets its own name, so you can have multiple instances of a given functor for the same type. It's rarely been the case that I needed such expressiveness, but it has been nice in those cases where I have.

The other modern branch on the ML family tree, OCaml, is good to know because there is a large community invested in it, which means that there are a lot of libraries available. The OCaml tool-chain is also rich, with interpreters, optimizing compilers and byte-code compilers available to the developer.

Because the ML languages are more expressive than all the mainstream languages, but they still permit side effects, they make a nice stop on the way to learning Haskell. In Haskell, programmers not yet well versed in functional program design may find they repeatedly code themselves into a corner, where they don't have access to the monad that they need. The MLs keep the side effects "escape hatch" open to patch over incomplete design, which prevents projects from coming to a sudden, unexpected "refactor-or-abort" decision point. One useful measure of a language is how well it tolerates a bad or incomplete design for the software system, since design is something that inevitably changes as a program evolves. In this regard, the MLs still have the upper hand over Haskell.

Favorite features

- Flex records. (SML only)
- Pattern matching.
- Structures and functors.

Scheme

Scheme is a language with a pure core (λ-calculus and the theory of lists) and a design mandate to maximize freedom of expression. It's untyped, which makes it ideal for web-based programming and rapid prototyping. Given its Lisp heritage, Scheme is a natural fit for artificial intelligence.

With its support for arbitrary-precision numerics, Scheme is also my first choice for implementing cryptographic algorithms. [For examples, see my short implementations of RSA and the Fermat and Solovay-Strassen primality tests in Scheme.]

By far, the most compelling reason to use Scheme is its macro system. All of the macro systems available for Scheme, including the standard syntax-rules and syntax-case systems, are Turing-equivalent.

Consequently, the programmer can reconfigure Scheme to reduce the impedance mismatch between the language and the task at hand. Combined with support for first-class continuations, it is even possible to embed alternate programming paradigms (like logic programming).

For example, in the code:

```
(let ((x (amb 3 4 5))
      (y (amb 6 7 8 )))
  (assert (= (+ x y) 12))
  (display x)
  (display y))
```

It is possible to write an amb macro that "chooses" the right argument to make a subsequent assert statement be true. (This program prints 4 and then 8.)

In Scheme, during any point in the computation, the program can capture the current continuation as a procedure: invoking this procedure returns the program to the evaluation context that existed when the continuation was captured. Programming with continuations feels like traveling back and forth in time and shifting between parallel universes.

Ultimately, Scheme is so minimal and extensible that there's not a whole lot to say about it, except that Scheme allows the programmer to extract from the language whatever the programmer is willing to put into it.

Favorite features

- S-Expressions as syntax and data.
- Hygienic macros.
- Continuations.
- Higher-order functions.

Scala

Scala is a rugged, expressive, strictly superior replacement for Java. Scala is the programming language I use for tasks like writing web servers or IRC clients. In contrast to OCaml, which was a functional language with an object-oriented system grafted to it, Scala feels more like a true hybrid. That is, object-oriented programmers should be able to start using Scala immediately, picking up the functional parts only as they choose to.

I learned of Scala from Martin Odersky's invited talk at POPL 2006. At the time, I saw functional programming as strictly superior to object-oriented programming, so I didn't see a need for a language that fused functional and object-oriented programming. (That was probably because all I wrote back then were compilers, interpreters and static analyzers.)

The need for Scala didn't become apparent to me until I wrote a concurrent HTTPD from scratch to support long-polled AJAX for yaplet. In order to get multicore support, I wrote the first version in Java. I don't think Java is all that bad, and I can enjoy well-done object-oriented programming. As a functional programmer, however, the lack of terse support for functional programming features (like higher-order functions) grates on me. So, I gave Scala a chance.

Scala runs on the JVM, so I could gradually port my existing project into Scala. It also means that Scala, in addition to its own rather large library, has access to the entire Java library as well. This means you can get real work done in Scala.

As I started using Scala, I became impressed by how tightly the functional and object-oriented worlds had been blended. In particular, Scala has a powerful case class/pattern-matching system that addressed annoyances lingering from my experiences with Standard ML, OCaml and Haskell: the programmer can decide which fields of an object should be matchable (as opposed to being forced to match on all of them), and variable-arity arguments are permitted. In fact, Scala even allows programmer-defined patterns.

I write a lot of functions that operate on abstract syntax nodes, so it's nice to match on only the syntactic children, while ignoring fields for annotations or source location.

The case class system lets one split the definition of an algebraic data type across multiple files or across multiple parts of the same file. Scala also supports well-defined multiple inheritance through class-like constructs called traits. And, Scala allows operator overloading; even function application and collection update can be overloaded. Used well, this tends to make my Scala programs more intuitive and concise.

One feature that turns out to save a lot of code, in the same way that type classes save code in Haskell, is implicits. You can imagine implicits as an API for the error-recovery phase of the type-checker. In short, when the type checker needs an X but got a Y, it will check to see if there's a function marked implicit in scope that converts Y into X; if it finds one, it automatically applies the implicit function to repair the type error.

Implicits make it possible to look like you're extending the functionality of a type for a limited scope. For example, suppose you want to "add" an escapeHTML() method to type String. You can't modify the definition of String, but with implicits, you can make it so that when type-checking fails on myString.escapeHTML(), it will look for an implicit function in scope that can convert a String object into a type that supports the escapeHTML() method.

Implicits also allow cleaner domain-specific embedded languages (DSELs) in Scala, since they allow you to transparently map Scala literals (like 3 or "while") into literals in the DSEL.

Favorite features

- JVM support.
- Intelligent operator overloading.
- Extensive library.
- Case classes/pattern matching.
- Extensible pattern matching.
- Multiple inheritance via traits.
- Rich, flexible object constructors.
- Implicit type conversions.
- Lazy fields and arguments.

Resources available on the original post *http://hn.my/apl/*. ■

Matt Might is a professor of Computer Science at the University of Utah. His research interests include programming language design, static analysis and compiler optimization. He blogs at *http://matt.might.net/articles/* and tweets from @mattmight.

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

Emacs Isn't For Everyone

By BRIAN CARPER

Chas Emerick recently posted the results of his State of Clojure survey. It turns out that the (self-selected) group of Clojure-using respondents happen to prefer Emacs as their IDE of choice, eclipsing all other editors by a large margin.

Chas then has this to say:

"I continue to maintain that broad acceptance and usage of Clojure will require that there be top-notch development environments for it that mere mortals can use and not be intimidated by...and IMO, while emacs is hugely capable, I think it falls down badly on a number of counts related to usability, community/ecosystem, and interoperability."

As an avid, die-hard Vim and Emacs user for life, I'm going to agree.

Mere mortals?

Emacs isn't difficult to learn. Not in the sense of requiring skill or cleverness. It is however extremely painful to learn. I think there's a difference.

The key word is tedium. Learning Emacs is a long process of rote memorization and repetition of commands until they become muscle memory. If you're smart enough to write programs, you can learn Emacs. You just have to keep dumping time into the task until you become comfortable.

Until you're comfortable, you face the unpleasant task of un-learning all of your habits and forming new ones. And you're trying to do this at the same time you're undertaking another, even harder task: writing programs. And if you're a new Clojurist, and you're learning Emacs and Clojure from scratch at the same time, well, get the headache medication ready.

As a programmer and someone who sits in front of a computer 12+ hours a day, I consider myself pretty flexible and capable of picking up a new user interface. As someone who had been using Vim for years prior to trying Emacs, I considered myself more than capable of learning even a strange and foreign interface. I'd done it once before.

But learning Emacs still hurt. Oh how it hurt. I blogged while I was learning it, and you can see my pain firsthand. I sometimes hear people say "I tried Emacs for a whole month and I still couldn't get it". Well, it took me over a year to be able to sit down at Emacs and use it fluidly for long periods of time without tripping over the editor.

To be fair, I'm talking here about using Emacs as a programming environment. Using Emacs as a Notepad replacement could be learned in short order. C-x C-f, C-x C-s, or use the menus, there you go. Using it comfortably as a full-fledged IDE is significantly harder and requires you to touch (and master) many more features. Syntax highlighting, tab-completion, directory traversal and cwd issues, enabling line numbers, version-control integration, build tool integration, Emacs' funky regex syntax for search/replace, Emacs' bizarre kill rings and undo rings, the list goes on. These things are very flexible in Emacs, which is a great thing, but it's also an impediment to learning how to configure and use them. There's no getting around the time investment.

And it's not just a matter of learning some new keyboard shortcuts. There's a new vocabulary to learn. You don't open files, you visit them. What's a buffer? What's a window? (Not what you think it is.) What's a point? What's a mark? Kill? Yank? "Apropos"? Huh? C-c M-o means what exactly? My keyboard doesn't have a Meta key. Yeah, you can use CUA mode and get your modernized Copy/Cut/Paste shortcuts back, but that's the tip of the iceberg. It's hard even to know where to begin looking for help.

Yeah, Emacs came first, before our more common and more modern conventions were established, and that explains why it's so different. That doesn't change the fact that Emacs today is a strange beast.

Community and ecosystem

Personally I find the Emacs community to be a pretty nice bunch. In the highest tradition of hackerdom and open source software, Emacs users seem to be eager and willing to share their elisp snippets and bend over backwards to help other people learn the editor. I got lots of help when I was struggling and learning Emacs.

The Emacs wiki is an awesome resource. The official documentation is so complete (and so long) that it leaves me speechless sometimes. And there are a million 3rd-party scripts for it. Whatever you want Emacs to do is generally a short google away.

If there's anything wrong with the Emacs community, it'd be people who take Emacs evangelism overboard. The answer to "I don't want to have to use Emacs to use your language" can't be "Be quiet and learn more Emacs," or "If you're too dumb to learn Emacs, go away." In some communities there is certainly some of that. But thankfully I don't see it much in the Clojure community. Let's hope it stays that way.

Interoperability

Once someone spends the time to write a suitable amount of elisp, Emacs can interoperate with anything. I think so many people use SLIME for Clojure development precisely because it interoperates so darned well with Lisps. SLIME is amazing. You probably can't beat Paredit either, and Emacs' flexibility is precisely what makes things like Paredit possible.

The problem is the amount of time you have to spend to get that interoperability set up and to learn how to use it. After two years of using Emacs and Clojure together, every once in a while I still find myself bashing my face on my desk trying to get the latest SLIME or swank to work just right, or trying to get a broken key binding fixed, or tweaking some other aspect of Emacs that's driving me crazy. One day, curly braces stopped being recognized as matched pairs by Paredit. Why? No idea; I fixed it, but it was a half hour of wasted time.

Emacs is good at integrating with Git too. So good that there are four or five different Emacs-Git libraries, each with a different interface and feature set. I gave up eventually and went back to using the command line. (You can embed a shell / command line right in Emacs. There are three or four different libraries to do that too.)

The wealth of options of ways to do things in Emacs is simultaneously a good thing, overwhelming and confusing. If all you want is something that works and gets out of your way, too many options can be worse than one option, even if that one option isn't entirely ideal.

Emacs' Java interop, I know nothing about. Almost certainly, Emacs can come close to a modern Java IDE for fancy features like tab-completion and document lookups and project management. But how long is it going to take you to figure out that tab-completion is called hippie-expand in Emacs? That and a million other surprises await you.

What's my point?

There was a pithy quote floating around on Twitter a while back (I think quoting Rich Hickey):

"One possible way to deal with being unfamiliar with something is to become familiar with it."

That's true, and you could say that of Emacs. I strongly believe that when it comes to computers, there's no such thing as "intuitive". There's stuff you've already spent a lot of time getting used to, and there's stuff you haven't.

But certain things require more of a time investment than others. Could I learn Clojure if all the keywords were in Russian or Chinese instead of my native English? Sure, but it'd take me a long time. I'd certainly have to have a good reason to attempt it.

I learned Emacs partly because it was hard. I saw it as a challenge. It was fun, yet painful, but more pain, more glory. Mastering it makes me feel like I've accomplished something. I'd encourage other people to learn Emacs and Vim too. I think the benefits of knowing them outweigh the cost and time investment of learning them.

But I didn't learn Emacs with the goal of being productive. I learned it for the same reason some people build cars in their garages, while most people just buy one and drive it to and from work every day. I learned Emacs because I love programming and I love playing with toys, and Vim or Emacs are as nice a toy as I could ask for. (I love programming enough to form strong opinions and write huge blog posts about text editors.) For me, productivity was a beneficial side-effect.

There are only so many hours in a day. There are a lot of other challenges to conquer, some of which offer more tangible benefits than Emacs mastery would get you. Mastering an arcane text editor isn't necessarily going to be on the top of the list of everyone's goals in life, especially when there are other editors that are easier to use and give you a significant subset of what Emacs would give you. We have to pick our battles.

So I understand when people say they don't want to learn Emacs. I think maybe so many Clojurists use Emacs right now because we're still in the early adopter stage. If you're using Clojure today, you're probably pretty enthusiastic about programming. You're likely invested enough to be willing to burn the required time to learn Emacs.

If Clojure becomes "big", there are going to be a lot of casual users. A casual user of Clojure isn't going to learn Emacs. They're going to silently move on to another language. And I really think that new blood is vital to the strength of a community and necessary for the continued healthy existence of a programming language.

So Clojure does need alternatives. I'll stick with Emacs myself, but there should be practical alternatives. I'd encourage the Clojure community to continue to support and enjoy Emacs, but don't push it too hard. ■

Brian is a professional programmer and hobbyist text-editor enthusiast. He writes about these topics at *http://briancarper.net*.

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

What Every Developer Should Know About URLs

By ALAN SKORKIN

I HAVE RECENTLY WRITTEN about the value of fundamentals in software development. I am still firmly of the opinion that you need to have your fundamentals down solid, if you want to be a decent developer. However, several people made a valid point in response to that post, in that it is often difficult to know what the fundamentals actually are (be they macro or micro level). So, I thought it would be a good idea to do an ongoing series of posts on some of the things that I consider to be fundamental – this post is the first installment.

Being a developer this day and age, it would be almost impossible for you to avoid doing some kind of web-related work at some point in your career. That means you will inevitably have to deal with URLs at one time or another. We all know what URLs are about, but there is a difference between knowing URLs like a user and knowing them like a developer should know them.

As a web developer you really have no excuse for not knowing everything there is to know about URLs, there is just not that much to them. But, I have found that even experienced developers often have some glaring holes in their knowledge of URLs. So, I thought I would do a quick tour of everything that every developer should know about URLs. Strap yourself in – this won't take long :).

The Structure Of A URL

This is easy, starts with HTTP and ends with .com right :)? Most URLs have the same general syntax, made up of the following nine parts:

```
<scheme>://<username>:<password>@<host>:<port>/<path>;
<parameters>?<query>#<fragment>
```

Most URLs won't contain all of the parts. The most common components, as you undoubtedly know, are the scheme, host and path. Let's have a look at each of these in turn:

- **scheme** – this basically specifies the protocol to use to access the resource addressed by the URL (e.g. http, ftp). There are a multitude of different schemes. A scheme is official if it has been registered with the IANA (like http and ftp), but there are many unofficial (not registered) schemes which are also in common use (such as sftp, or svn). The scheme must start with a letter and is separated from the rest of the URL by the first : (colon) character. That's right, the // is not part of the separator but is in fact the beginning of the next part of the URL.

- **username** – this along with the password, the host and the port form what's known as the authority part of the URL. Some schemes require authentication information to access a resource this is the username part of that authentication information. The username and password are very common in ftp URLs, they are less common in http URLs, but you do come across them fairly regularly.
- **password** – the other part of the authentication information for a URL, it is separated from the username by another : (colon) character. The username and password will be separated from the host by an @ (at) character. You may supply just the username or both the username and password e.g.:

```
ftp://some_user@blah.com/
ftp://some_user:some_path@blah.com/
```

If you don't supply the username and password and the URL you're trying to access requires one, the application you're using (e.g. browser) will supply some defaults.

- host – as I mentioned, it is one of the components that makes up the authority part of the URL. The host can be either a domain name or an IP address, as we all should know the domain name will resolve to an IP address (via a DNS lookup) to identify the machine we're trying to access.
- port – the last part of the authority. It basically tells us what network port a particular application on the machine we're connecting to is listening on. As we all know, for HTTP the default port is 80, if the port is omitted from an http URL, this is assumed.
- path – is separated from the URL components preceding it by a / (slash) character. A path is a sequence of segments separated by / characters. The path basically tells us where on the server machine a resource lives. Each of the path segments can contain parameters which are separated from the segment by a ; (semi-colon) character e.g.:

```
http://www.blah.com/some;param1=foo/crazy;param2=bar/path.html
```

The URL above is perfectly valid, although this ability of path segments to hold parameters is almost never used (I've never seen it personally).

- **parameters** – talking about parameters, these can also appear after the path but before the query string, also separated from the rest of the URL and from each other by ; characters e.g.:

```
http://www.blah.com/some/crazy/path.html;param1=foo;
param2=bar
```

As I said, they are not very common.

- **query** – these on the other hand are very common as every web developer would know. This is the preferred way to send some parameters to a resource on the server. These are key=value pairs and are separated from the rest of the URL by a ? (question mark) character and are normally separated from each other by & (ampersand) characters. What you may not know is the fact that it is legal to separate them from each other by the ; (semi-colon) character as well. The following URLs are equivalent:

```
http://www.blah.com/some/crazy/path.html?param1=foo
&param2=bar

http://www.blah.com/some/crazy/path.html?param1=foo
;param2=bar
```

- **fragment** – this is an optional part of the URL and is used to address a particular part of a resource. We usually see these used to link to a particular section of an html document. A fragment is separated from the rest of the URL with a # (hash) character. When requesting a resource addressed by a URL from a server, the client (i.e. browser) will usually not send the fragment to the server (at least not where HTTP is concerned). Once the client has fetched the resource, it will then use the fragment to address the relevant part.

That's it, all you need to know about the structure of a URL. From now on you no longer have any excuse for calling the fragment – "that hash link thingy to go to a particular part of the html file".

Special Characters In URLs

There is a lot of confusion regarding which characters are safe to use in a URL and which are not, as well as how a URL should be properly encoded. Developers often try to infer this stuff from general knowledge (i.e. the / and : characters should obviously be encoded since they have special meaning in a URL). This is not necessary, you should know this stuff solid – it's simple. Here is the low down.

There are several sets of characters you need to be aware of when it comes to URLs. Firstly, the characters that have special meaning within a URL are known as reserved characters, these are:

```
";" | "/" | "?" | ":" | "@" | "&" | "=" | "+" | "$" | ","
```

What this means is that these characters are normally used in a URL as-is and are meaningful within a URL context (i.e. separate components from each other etc.). If a part of a URL (such as a query parameter), is likely to contain one of these characters, it should be escaped before being included in the URL. I have spoken about URL encoding before, check it out, we will revisit it shortly.

The second set of characters to be aware of is the unreserved set. It is made up of the following characters:

```
"-" | "_" | "." | "!" | "~" | "*" | "'" | "(" | ")"
```

The characters can be included as-is in any part of the URL (note that they may not be allowed as part of a particular component of a URL). This basically means you don't need to encode/escape these characters when including them as part of a URL. You CAN escape them without changing the semantics of a URL, but it is not recommended.

The third set to be aware of is the 'unwise' set, i.e. it is unwise to use these characters as part of a URL. It is made up of the following characters:

```
"{" | "}" | "|" | "\" | "^" | "[" | "]" | "`"
```

These characters are considered unwise to use in a URL because gateways are known to sometimes modify such characters, or they are used as delimiters. That doesn't mean that these characters will always be modified by a gateway, but it can happen. So, if you include these as part of a URL without escaping them, you do this at your own risk. What it really means is you should always escape these characters if a part of your URL (i.e. like a query param) is likely to contain them.

The last set of characters is the excluded set. It is made up of all ASCII control characters, the space character as well the following characters (known as delimiters):

```
"<" | ">" | "#" | "%" | '"'
```

The control characters are non-printable US-ASCII characters (i.e. hexadecimal 00-1F as well as 7F). These characters must always be escaped if they are included in a component of a URL. Some, such as # (hash) and % (percent) have special meaning within the context of a URL (they can really be considered equivalent to the reserved characters). Other characters in this set have no printable representation and therefore escaping them is the only way to represent them. The <, > and " characters should be escaped since these characters are often used to delimit URLs in text.

To URL encode/escape a character we simply append its 2 character ASCII hexadecimal value to the % character. So, the URL encoding of a space character is %20 – we have all seen that one. The % character itself is encoded as %25.

That's all you need to know about various special characters in URLs. Of course aside from those characters, alpha-numerics are allowed and don't need to be encoded :).

A few things you have to remember. A URL should always be in its encoded form. The only time you should decode parts of the URL is when you're pulling the URL apart (for whatever reason). Each part of the URL must be encoded separately, this should be pretty obvious, you don't want to try encoding an already constructed URL, since there is no way to distinguish when reserved characters are used for their reserved purpose (they shouldn't be encoded) and when they are part of a URL component (which means they should be encoded). Lastly you should never try to double encode/decode a URL. Consider that if you encode a URL once but try to decode it twice and one of the URL components contains the % character you can destroy your URL e.g.:

```
http://blah.com/yadda.html?param1=abc%613
```

When encoded it will look like this:

```
http://blah.com/yadda.html?param1=abc%25613
```

If you try to decode it twice you will get:

```
http://blah.com/yadda.html?param1=abc%613
```
Correct

```
http://blah.com/yadda.html?param1=abca3
```
Stuffed

By the way I am not just pulling this stuff out of thin air. It is all defined in RFC 2396, you can go and check it out if you like, although it is by no means the most entertaining thing you can read, I'd like to hope my post is somewhat less dry :).

Absolute vs Relative URLs

The last thing that every developer should know is the difference between an absolute and relative URL as well as how to turn a relative URL into its absolute form.

The first part of that is pretty easy, if a URL contains a scheme (such as http), then it can be considered an absolute URL. Relative URLs are a little bit more complicated.

A relative URL is always interpreted relative to another URL (hence the name :)), this other URL is known as the base URL. To convert a relative URL into its absolute form we firstly need to figure out the base URL, and then, depending on the syntax of our relative URL we combine it with the base to form its absolute form.

We normally see a relative URL inside an html document. In this case there are two ways to find out what the base is.

1. The base URL may have been explicitly specified in the document using the HTML <base> tag.
2. If no base tag is specified, then the URL of the html document in which the relative URL is found should be treated as the base.

Once we have a base URL, we can try and turn our relative URL into an absolute one. First, we need to try and break our relative URL into components (i.e. scheme, authority (host, port), path, query string, fragment). Once this is done, there are several special cases to be aware of, all of which mean that our relative URL wasn't really relative.

- if there is no scheme, authority or path, then the relative URL is a reference to the base URL
- if there is a scheme then the relative URL is actually an absolute URL and should be treated as such
- if there is no scheme, but there is an authority (host, port), then our relative URL is likely a network path, we take the scheme from our base URL and append our "relative" URL to it separating the two by ://

If none of those special cases occurred then we have a real relative URL on our hands. Now we need to proceed as follows.

- we inherit the scheme, and authority (host, port) from the base URL
- if our relative URL begins with /, then it is an absolute path, we append it to the scheme and authority we inherited from the base using appropriate separators to get our absolute URL
- if relative URL does not begin with / then we take the path of the base URL, discarding everything after the last / character
- we then take our relative URL and append it to the resulting path, we now need to do a little further processing which depends on the first several characters of our relative URL
- if there is a ./ (dot slash) anywhere in a resulting path we remove it (this means our relative URL started with ./ i.e. ./blah.html)
- if there is a ../ (dot dot slash) anywhere in the path then we remove it as well as the preceding segment of the path i.e. all occurrences of "/../" are removed, keep doing this step until no more ../ can be found anywhere in the path (this means our relative path started with one or more ../ i.e. ../blah.html or ../../blah.html etc.)
- if the path ends with .. then we remove it and the preceding segment of the path, i.e. "/.." is removed (this means our relative path was .. (dot dot))
- if the path ends with a . (dot) then we remove it (this most likely means our relative path was . (dot))

At this point we simply append any query string or fragment that our relative URL may have contained to our URL using appropriate separators and we have finished turning our relative URL into an absolute one.

Here are some examples of applying the above algorithm:

```
1)
base: http://www.blah.com/yadda1/yadda2/
yadda3?param1=foo#bar
relative: rel1

final absolute: http://www.blah.com/yadda1/yadda2/rel1

2)
base: http://www.blah.com/yadda1/yadda2/
yadda3?param1=foo#bar
relative: /rel1

final absolute: http://www.blah.com/rel1

3)
base: http://www.blah.com/yadda1/yadda2/
yadda3?param1=foo#bar
relative: ../rel1

final absolute: http://www.blah.com/yadda1/rel1

4)
base: http://www.blah.com/yadda1/yadda2/
yadda3?param1=foo#bar
relative: ./rel1?param2=baz#bar2

final absolute: http://www.blah.com/yadda1/yadda2/
rel1?param2=baz#bar2

5)
base: http://www.blah.com/yadda1/yadda2/
yadda3?param1=foo#bar
relative: ..

final absolute: http://www.blah.com/yadda1/
```

Now you should be able to confidently turn any relative URL into an absolute one, as well as know when to use the different forms of relative URL and what the implications will be. For me this has come in handy time and time again in my web development endeavours.

There you go that's really all there is to know about URLs, it's all relatively simple (forgive the pun :)) so no excuse for being unsure about some of this stuff next time. Talking about next time, one of the most common things you need to do when it comes to URLs is recognise if a piece of text is in fact a URL, so next time I will show you how to do this using regular expressions (as well as show you how to pull URLs out of text). It should be pretty easy to construct a decent regex now that we've got the structure and special characters down. Stay tuned. ■

Alan Skorkin is a developer and aspiring software craftsman from Melbourne, Australia. He is often found causing controversy on his blog skorks.com, while sharing his thoughts about hacking, the software development profession and the people who work in it.

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

Understanding and Applying Operational Transformation

Algorithm Behind Google Wave and Google Docs

By DANIEL SPIEWAK

Almost exactly a year ago, Google made one of the most remarkable press releases in the Web 2.0 era. Of course, by "press release", I actually mean keynote at their own conference, and by "remarkable" I mean potentially-transformative and groundbreaking. I am referring of course to the announcement of Google Wave, a real-time collaboration tool which has been in open beta for the last several months.

For those of you who don't know, Google Wave is a collaboration tool based on real-time, simultaneous editing of documents via a mechanism known as "operational transformation". Entities which appear as messages in the Wave client are actually "waves". Within each "wave" is a set of "wavelets", each of which contains a set of documents. Individual documents can represent things like messages, conversation structure (which reply goes where, etc), spell check metadata and so on. Documents are composed of well-formed XML with an implicit root node. Additionally, they carry special metadata known as "annotations" which are (potentially-overlapping) key/value ranges which span across specific regions of the document. In the Wave message schema, annotations are used to represent things like bold/italic/underline/strikethrough formatting, links, caret position, the conversation title and a host of other things. An example document following the Wave message schema might look something like this:

```
<body>
  <line/>Test message
  <line/>
  <line/>Lorem ipsum dolor sit amet.
</body>
```

(assuming the following annotations):

- style/font-weight -> bold
- style/font-style -> italic
- link/manual -> http://www.google.com

You will notice that the annotations for `style/font-style` and `link/manual` actually overlap. This is perfectly acceptable in Wave's document schema. The resulting rendering would be something like this:

Test message

Lorem ipsum dolor sit amet.

The point of all this explaining is to give you at least a passing familiarity with the Wave document schema so that I can safely use its terminology in the article to come. See, Wave itself is not nearly so interesting as the idea upon which it is based. As mentioned, every document in Wave is actually just raw XML with some ancillary annotations. As far as the Wave server is concerned, you can stuff whatever data you want in there, just so long as it's well-formed. It just so happens that Google chose to implement a communications tool on top of this data backend, but they could have just as easily implemented something more esoteric, like a database or a windowing manager.

The key to Wave is the mechanism by which we interact with these documents: operational transformation. Wave actually doesn't allow you to get access to a document as raw XML or anything even approaching it. Instead, it demands that all of your access to the document be performed in terms of operations. This has two consequences: first, it allows for some really incredible collaborative tools like the Wave client; second, it makes it *really* tricky to implement any sort of Wave-compatible service. Given the fact that I've been working on Novell Pulse (which is exactly this sort of service), and in light of the fact that Google's documentation on the subject is sparing at best, I thought I would take some time to clarify this critical piece of the puzzle. Hopefully, the information I'm about to present will make it easier for others attempting to interoperate with Wave, Pulse and the (hopefully) many OT-based systems yet to come.

Operations

Intuitively enough, the fundamental building block of operational transforms are operations themselves. An operation is exactly what it sounds like: an action which is to be performed on a document. This action could be inserting or deleting characters, opening (and closing!) an XML element, fiddling with annotations, etc. A single operation may actually perform many of these actions. Thus, an operation is actually made up of a sequence of operation *components*, each of which performs a particular action with respect to the *cursor* (not to be confused with the *caret*, which is specific to the client editor and not at all interesting at the level of OT).

There are a number of possible component types. For example:

- insertCharacters — Inserts the specified string at the current index
- deleteCharacters — Deletes the specified string from the current index
- openElement — Creates a new XML open-tag at the current index
- deleteOpenElement — Deletes the specified XML open-tag from the current index
- closeElement — Closes the first currently-open tag at the current index
- deleteCloseElement — Deletes the XML close-tag at the current index
- annotationBoundary — Defines the changes to any annotations (starting or ending) at the current index
- retain — Advances the index a specified number of items

Wave's OT implementation actually has even more component types, but these are the important ones. You'll notice that every component has something to do with the cursor index. This concept is central to Wave's OT implementation. Operations are effectively a stream of components, each of which defines an action to be performed which effects the content, the cursor or both.

For example, we can encode the example document from earlier as follows:

1. openElement('body')
2. openElement('line')
3. closeElement()
4. annotationBoundary(startKeys: ['style/font-weight'], startValues: ['bold'])
5. insertCharacters('Test message')
6. annotationBoundary(endKeys: ['style/font-weight'])
7. openElement('line')
8. closeElement()
9. annotationBoundary(startKeys: ['style/font-style'], startValues: ['italic'])
10. openElement('line')
11. closeElement()
12. insertCharacters('Lorem ')
13. annotationBoundary(startKeys: ['link/manual'], startValues: ['http://www.google.com'])
14. insertCharacters('ipsum')
15. annotationBoundary(endKeys: ['style/font-style'])
16. insertCharacters(' dolor')
17. annotationBoundary(endKeys: ['link/manual'])
18. insertCharacters(' sit amet.')
19. closeElement()

Obviously, this isn't the most streamlined way of referring to a document's content for a human, but a stream of discrete components like this is *perfect* for automated processing. The real utility of this encoding though doesn't become apparent until we look at operations which only encode a partial document; effectively performing a particular mutation. For example, let's follow the advice of *Strunk and White* and capitalize the letter 'm' in our title of 'Test message'. What we want to do (precisely-speaking) is delete the 'm' and insert the string 'M' at its previous location. We can do that with the following operation:

1. retain(8)
2. deleteCharacters('m')
3. insertCharacters('M')
4. retain(38)

Instead of adding content to the document at ever step, most of this operation actually leaves the underlying document untouched. In practice, retain() tends to be the most commonly used component by a wide margin. The trick is that every operation must span the full width of the document. When evaluating this operation, the cursor will start at index 0 and walk forward through the existing document and the incoming operation one item at a time. Each XML tag (open or close) counts as a single item. Characters are also single items. Thus, the entire document contains 47 items.

Our operation above cursors harmlessly over the first eight items (the <body> tag, the <line/> tag and the string 'Test '). Once it reaches the 'm' in 'message', we stop the cursor and perform a mutation. Specifically, we're using the deleteCharacters() component to remove the 'm'. This component doesn't move the cursor, so we're still sitting at index 8. We then use the insertCharacters() component to add the character 'M' at precisely our currently location. This time, some new characters have been inserted, so the cursor advances to the end of the newly-inserted string (meaning that we are now at index 9). This is intuitive because we don't want to have to retain() over the text we just inserted. We do however want to retain() over the remainder of the document, seeing as we don't need to do anything else. The final rendered document looks like the following:

Test Message

Lorem ipsum dolor sit amet.

Composition

One of Google's contributions to the (very old) theory behind operational transformation is the idea of operation composition. Because Wave operations are these nice, full-span sequences of discrete components, it's fairly easy to take two operations which span the same length and merge them together into a single operation. The results of this action are really quite intuitive. For example, if we were to compose our document operation (the first example above) with our 'm'-changing operation (the second example), the resulting operation would be basically the same as the original document operation, except that instead of inserting the text 'Test message', we would insert 'Test Message'. In composing the two operations together, all of the retains have disappeared and any contradicting components (e.g. a delete and an insert) have been directly merged.

Composition is extremely important to Wave's OT as we will see once we start looking at client/server asymmetry. The important thing to notice now is the fact that composed operations *must* be fundamentally compatible. Primarily, this means that the two operations must span the same number of indexes. It also means that we cannot compose an operation which consists of only a text insert with an operation which attempts to delete an XML element. Obviously, that's not going to work. Wave's `Composer` utility takes care of validating both the left and the right operation to ensure that they are compatible as part of the composition process.

Please also note that composition is *not* commutative; ordering is significant. This is also quite intuitive. If you type the character `a` and then type the character `b`, the result is quite different than if you type the character `b` and then type the character `a`.

Transformation

Here's where we get to some of the really interesting stuff and the motivation behind all of this convoluted representational baggage. Operational Transformation, at its core, is an *optimistic* concurrency control mechanism. It allows two editors to modify the same section of a document at the same time without conflict. Or rather, it provides a mechanism for sanely resolving those conflicts so that neither user intervention nor locking become necessary.

This is actually a harder problem than it sounds. Imagine that we have the following document (represented as an operation):

1. insertCharacters('go')

Now imagine that we have two editors with their cursors positioned at the end of the document. They simultaneously insert a `t` and `a` character (respectively). Thus, we will have two operations sent to the server. The first will retain 2 items and insert a `t`, the second will retain 2 items and insert `a`. Naturally, the server needs to enforce atomicity of edits at some point (to avoid race conditions during I/O), so one of these operations will be applied first. However, as soon as either one of these operations is applied, the retain for the other will become invalid. Depending on the ordering, the text of the resulting document will either be 'goat' or 'gota'.

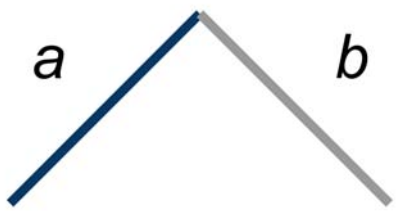
In and of itself, this isn't really a problem. After all, any asynchronous server needs to make decisions about ordering at some point. However, issues start to crop up as soon as we consider relaying operations from one client to the other. Client A has already applied its operation, so its document text will be 'got'. Meanwhile, client B has already applied its operation, and so its document text is 'goa'. Each client needs the operation from the other in order to have any chance of converging to the same document state.

Unfortunately, if we naïvely send A's operation to B and B's operation to A, the results will *not* converge:

- 'got' + (retain(2); insertCharacters('a') = 'goat'
- 'goa' + (retain(2); insertCharacters('t') = 'gota'

Even discounting the fact that we have a document size mismatch (our operations each span 2 indexes, while their target documents have width 3), this is obviously not the desired behavior. Even though our server may have a sane concept of consistent ordering, our clients obviously need some extra hand-holding. Enter OT.

What we have here is a simple one-step diamond problem. In the theoretical study of OT, we generally visualize this situation using diagrams like the following:



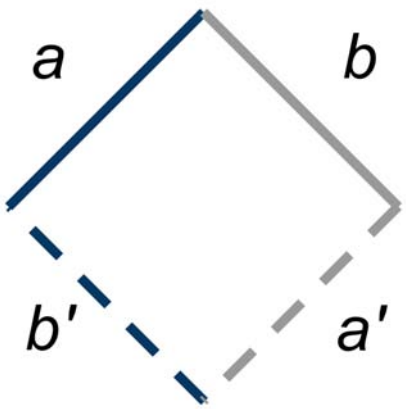
The way you should read diagrams like this is as a graphical representation of operation application on two documents at the same time. Client operations move the document to the left. Server operations move the document to the right. Both client and server operations move the document downward. Thus, diagrams like these let us visualize the application of operations in a literal "state space". The dark blue line shows the client's path through state space, while the gray line shows the server's. The vertices of these paths (not explicitly rendered) are points in state space, representing a particular state of the document. When both the client and the server line pass through the same point, it means that the content of their respective documents were in sync, at least at that particular point in time.

So, in the diagram above, operation a could be client A's operation (retain(2); insertCharacters('t')) and operation b could be client B's operation. This is of course assuming that the server chose B's operation as the "winner" of the race condition. As we showed earlier, we cannot simply naïvely apply operation *a* on the server and *b* on the client, otherwise we could derive differing document states ('goat' vs 'gota'). What we need to do is automatically adjust operation *a* with respect to *b* and operation *b* with respect to *a*.

We can do this using an operational transform. Google's OT is based on the following mathematical identity:

$$\text{xform}(a, b) = (a', b'), \text{ where } b' \circ a \equiv a' \circ b$$

In plain English, this means that the `transform` function takes two operations, one server and one client, and produces a pair of operations. These operations can be applied to their counterpart's end state to produce exactly the same state when complete. Graphically, we can represent this by the following:



Thus, on the client-side, we receive operation b from the server, pair it with `a` to produce `(a', b')`, and then compose `b'` with a to produce our final document state. We perform an analogous process on the server-side. The mathematical definition of the `transform` function guarantees that this process will produce the exact same document state on both server and client.

Coming back to our concrete example, we can finally solve the problem of 'goat' vs 'gota'. We start out with the situation where client A has applied operation a, arriving at a document text of 'got'. It now receives operation b from the server, instructing it to retain over 2 items and insert character 'a'. However, before it applies this operation (which would obviously result in the wrong document state), it uses operational transformation to derive operation `b'`. Google's OT implementation will resolve the conflict between 't' and 'a' in favor of the server. Thus, `b'` will consist of the following components:

1. retain(2)
2. insertCharacters('a')
3. retain(1)

You will notice that we no longer have a document size mismatch, since that last retain() ensures that the cursor reaches the end of our length-3 document state ('got').

Meanwhile, the server has received our operation a and it performs an analogous series of steps to derive operation a'. Once again, Google's OT must resolve the conflict between 't' and 'a' in the same way as it resolved the conflict for client A. We're trying to apply operation a (which inserts the 't' character at position 2) to the server document state, which is currently 'goa'. When we're done, we must have the exact same document content as client A following the application of b'. Specifically, the server document state must be 'goat'. Thus, the OT process will produce the operation a' consisting of the following components:

retain(3)
insertCharacters('t')

Client A applies operation `b'` to its document state, the server applies operation a' to its document state, and they both arrive at a document consisting of the text 'goat'. Magic!

It is very important that you really understand this process. OT is all about the transform function and how it behaves in this exact situation. As it turns out, this is all that OT does for us in and of itself. Operational transformation is really just a concurrency primitive. It doesn't solve every problem with collaborative editing of a shared document (as we will see in a moment), but it does solve this problem very well.

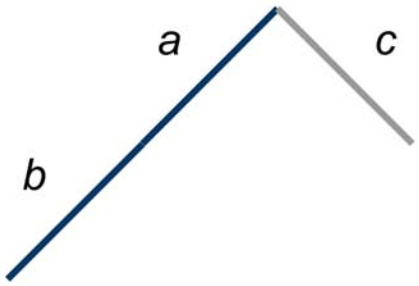
One way to think of this is to keep in mind the "diamond" shape shown in the above diagram. OT solves a very simple problem: given the top two sides of the diamond, it can derive the bottom two sides. In practice, often times we only want one side of the box (e.g. client A only needs operation `b'`, it doesn't need `a'`). However, OT always gives us both pieces of the puzzle. It "completes" the diamond, so to speak.

Compound OT

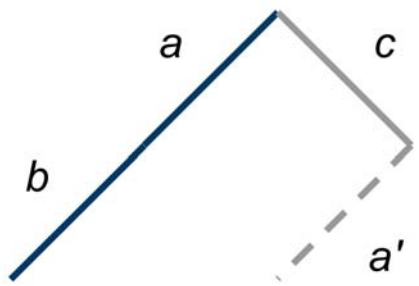
So far, everything I have presented has come pretty directly from the white-papers on waveprotocol.org. However, contrary to popular belief, this is not enough information to actually go out and implement your own collaborative editor or Wave-compatible service.

The problem is that OT doesn't really do all that much in and of itself. As mentioned above, OT solves for two sides of the diamond in state space. It only solves for two sides of a simple, one-step diamond like the one shown above. Let me say it a third time: the case shown above is the only case which OT handles. As it turns out, there are other cases which arise in a client/server collaborative editor like Google Wave or Novell Pulse. In fact, most cases in practice are much more complex than the one-step diamond.

For example, consider the situation where the client performs *two* operations (say, by typing two characters, one after the other) while at the same time the server performs one operation (originating from another client). We can diagram this situation in the following way:



So we have two operations in the client history, `a` and `b`, and only one operation in the server history, `c`. The client is going to send operations a and b to the server, presumably one after the other. The first operation (`a`) is no problem at all. Here we have the simple one-step diamond problem from above, and as well know, OT has no trouble at all in resolving this issue. The server transforms `a` and `c` to derive operation `a'`, which it applies to its current state. The resulting situation looks like the following:



Ok, so far so good. The server has successfully transformed operation `a` against `c` and applied the resulting `a'` to its local state. However, the moment we move on to operation `b`, disaster strikes. The problem is that the server receives operation `b`, but it has nothing against which to transform it!

Remember, OT only solves for the bottom two sides of the diamond given the top two sides. In the case of the first operation (`a`), the server had both top sides (`a` and `c`) and thus OT was able to derive the all-important `a'`. However, in this case, we only have one of the sides of the diamond (`b`); we don't have the server's half of the equation because the server never performed such an operation!

In general, the problem we have here is caused by the client and server diverging by more than one step. Whenever we get into this state, the OT becomes more complicated because we effectively need to transform incoming operations (e.g. `b`) against operations which never happened! In this case, the phantom operation that we need for the purposes of OT would take us from the tail end of a to the tail end of `a'`. Think of it like a "bridge" between client state space and server state space. We need this bridge, this second half of the diamond, if we are to apply OT to solve the problem of transforming `b` into server state space.

Operation Parentage

In order to do this, we need to add some metadata to our operations. Not only do our operations need to contain their components (retain, etc), they also must maintain some notion of parentage. We need to be able to determine exactly what state an operation requires for successful application. We will then use this information to detect the case where an incoming operation is parented on a state which is not in our history (e.g. `b` on receipt by the server).

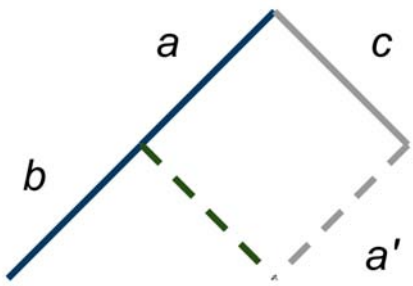
For the record, Google Wave uses a monotonically-increasing scalar version number to label document states and thus, operation parents. Novell Pulse does the exact same thing for compatibility reasons, and I recommend that anyone attempting to build a Wave-compatible service follow the same model. However, I personally think that compound OT is a lot easier to understand if document states are labeled by a hash of their contents.

This scheme has some very nice advantages. Given an operation (and its associated parent hash), we can determine instantly whether or not we have the appropriate document state to apply said operation. Hashes also have the very convenient property of converging exactly when the document states converge. Thus, in our one-step diamond case from earlier, operations `a` and `b` would be parented off of the same hash. Operation `b'` would be parented off of the hash of the document resulting from applying a to the initial document state (and similarly for `a'`). Finally, the point in state space where the client and server converge once again (after applying their respective operations) will have a single hash, as the document states will be synchronized. Thus, any further operations applied on either side will be parented off of a correctly-shared hash.

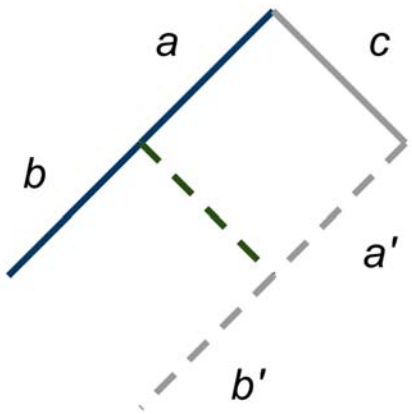
Just a quick terminology note: when I say "parent hash", I'm referring to the hash of the document state prior to applying a particular operation. When I say "parent operation" (which I probably will from time to time), I'm referring to the hash of the document state which results from applying the "parent operation" to its parent document state. Thus, operation `b` in the diagram above is parented off of operation `a` which is parented off of the same hash as operation `c`.

Compound OT

Now that our operations have parent information, our server is capable of detecting that operation b is not parented off of any state in its history. What we need to do is derive an operation which will take us from the parent of b to some point in server state-space. Graphically, this operation would look something like the following (rendered in dark green):



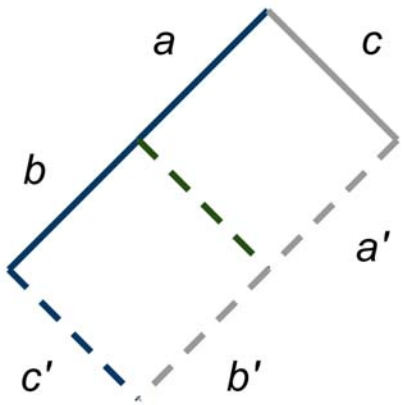
Fortunately for us, this operation is fairly easy to derive. In fact, we already derived and subsequently threw it away! Remember, OT solves for *two* sides of the diamond. Thus, when we transformed `a` against `c`, the resulting operation pair consisted of `a'` (which we applied to our local state) and another operation which we discarded. That operation is precisely the operation shown in green above. Thus, all we have to do is re-derive this operation and use it as the second top side of the one-step diamond. At this point, we have all of the information we need to apply OT and derive `b'`, which we can apply to our local state:



At this point, we're almost done. The only problem we have left to resolve is the application of operation `c` on the client. Fortunately, this is a fairly easy thing to do; after all, `c` is parented off of a state which the client has in its history, so it should be able to directly apply OT.

The one tricky point here is the fact that the client must transform `c` against not one but *two* operations (`a` and `b`). Fortunately, this is fairly easy to do. We could apply OT twice, deriving an intermediary operation in the first step (which happens to be exactly equivalent to the green intermediary operation we derived on the server) and then transforming that operation against `b`. However, this is fairly inefficient. OT is fast, but it's still `O(n log n)`. The better approach is to first compose `a` with `b` and then transform `c` against the composition of the two operations. Thanks to Google's careful definition of operation composition, this is guaranteed to produce the same operation as we would have received had we applied OT in two separate steps.

The final state diagram looks like the following:



Client/Server Asymmetry

Technically, what we have here is enough to implement a fully-functional client/server collaborative editing system. In fact, this is very close to what was presented in the 1995 paper on the Jupiter collaboration system. However, while this approach is quite functional, it isn't going to work in practice.

The reason for this is in that confusing middle part where the server had to derive an intermediary operation (the green one) in order to handle operation `b` from the client. In order to do this, the server needed to hold on to operation `a` in order to use it a second time in deriving the intermediary operation. Either that, or the server would have needed to speculatively retain the intermediary operation when it was derived for the first time during the transformation of `a` to `a'`. Now, this may sound like a trivial point, but consider that the server must maintain this sort of information essentially indefinitely for *every* client which it handles. You begin to see how this could become a serious scalability problem!

In order to solve this problem, Wave (and Pulse) imposes a very important constraint on the operations incoming to the server: any operation received by the server must be parented on some point in the server's history. Thus, the server would have rejected operation `b` in our example above since it did not branch from any point in server state space. The parent of `b` was `a`, but the server didn't have `a`, it only had `a'` (which is clearly a different point in state space).

Of course, simply rejecting any divergence which doesn't fit into the narrow, one-step diamond pattern is a bit harsh. Remember that practically, *almost all* situations arising in collaborative editing will be multi-step divergences like our above example. Thus, if we naïvely rejected anything which didn't fit into the one-step mold, we would render our collaborative editor all-but useless.

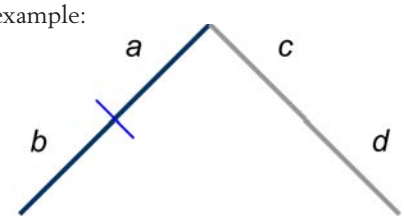
The solution is to move all of the heavy lifting onto the client. We don't want the server to have to track every single client as it moves through state space since there could be thousands (or even millions) of clients. But if you think about it, there's really no problem with the client tracking the *server* as it moves through state space, since there's never going to be any more than one (logical) server. Thus, we can offload most of the compound OT work onto the client side.

Before it sends any operations to the server, the client will be responsible for ensuring those operations are parented off of some point in the server's history. Obviously, the server may have applied some operations that the client doesn't know about yet, but that's ok. As long as any operations sent by the client are parented off of some point in the server's history, the server will be able to transform that incoming operation against the composition of anything which has happened since that point without tracking any history other than its own. Thus, the server never does anything more complicated than the simple one-step diamond divergence (modulo some operation composition). In other words, the server can always directly apply OT to incoming operations, deriving the requisite operation extremely efficiently.

Unfortunately, not all is sunshine and roses. Under this new regime, the client needs to work twice as hard, translating its operations into server state space and (correspondingly) server operations back into its state space. We haven't seen an example of this "reverse" translation (server to client) yet, but we will in a moment.

In order to maintain this guarantee that the client will never send an operation to the server which is not parented on a version in server state space, we need to impose a restriction on the client: we can never send more than one operation at a time to the server. This means that as soon as the client sends an operation (e.g. a in the example above), it must wait on sending b until the server acknowledges a. This is necessary because the client needs to somehow translate b into server state space, but it can't just "undo" the fact that b is parented on a. Thus, wherever b eventually ends up in server state space, it has to be a descendant of a', which is the server-transformed version of a. Literally, we don't know where to translate b into until we know exactly where a fits in the server's history.

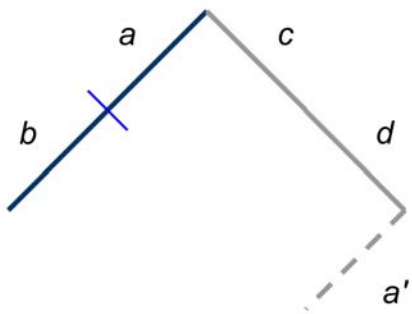
To help shed some light into this rather confusing scheme, let's look at an example:



In this situation, the client has performed two operations, a and b. The client immediately sends operation a to the server and buffers operation b for later transmission (the lighter blue line indicates the buffer boundary). Note that this buffering in no way hinders the application of local operations. When the user presses a key, we want the editor to reflect that change immediately, regardless of the buffer state. Meanwhile, the server has applied two other operations, c and d, which presumably come from other clients. The server still hasn't received our operation a.

Note that we were able to send a immediately because we are preserving every bit of data the server sends us. We still don't know about c and d, but we do know that the last time we heard from the server, it was at the same point in state space as we were (the parent of a and c). Thus, since a is already parented on a point in server state space, we can just send it off.

Now let's fast-forward just a little bit. The server receives operation a. It looks into its history and retrieves whatever operations have been applied since the parent of a. In this case, those operations are c and d. The server then composes c and d together and transforms a against the result, producing a'.

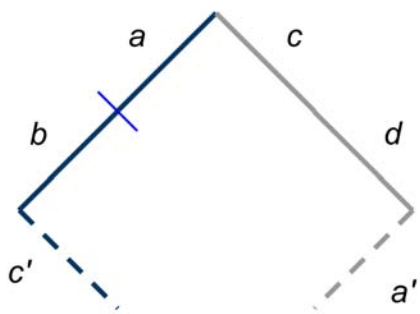


After applying a', the server broadcasts the operation to all clients, including the one which originated the operation. This is a very important design feature: whenever the server applies a transformed operation, it sends that operation off to all of its clients without delay. As long as we can guarantee strong ordering in the communication channels between the client and the server (and often we can), the clients will be able to count on the fact that they will receive operations from the server in *exactly* the order in which the server applied them. Thus, they will be able to maintain a locally-inferred copy of the server's history.

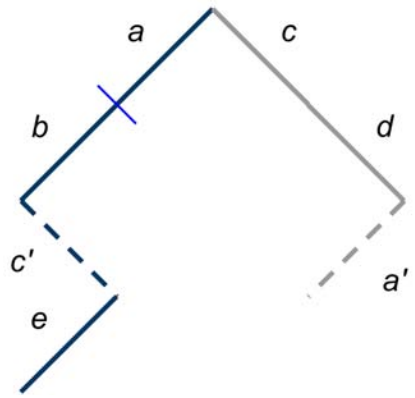
This also means that our client is going to receive a' from the server just like any other operation. In order to avoid treating our own transformed operations as if they were new server operations, we need some way of identifying our own operations and treating them specially. To do this, we add another bit of metadata to the operation: a locally-synthesized unique ID. This unique ID will be attached to the operation when we send it to the server and preserved by the server through the application of OT. Thus, operation a' will have the same ID as operation a, but a very different ID from operations c and d.

With this extra bit of metadata in place, clients are able to distinguish their own operations from others sent by the server. Non-self-initiated operations (like c and d) must be translated into client state space and applied to the local document. Self-initiated operations (like a') are actually server acknowledgements of our currently-pending operation. Once we receive this acknowledgement, we can flush the client buffer and send the pending operations up to the server.

Moving forward with our example, let's say that the client receives operation c from the server. Since c is already parented on a version in our local history, we can apply simple OT to transform it against the composition of a and b and apply the resulting operation to our local document:



Of course, as we always need to keep in mind, the client is a live editor which presumably has a real person typing madly away, changing the document state. There's nothing to prevent the client from creating another operation, parented off of c' which pushes it even further out of sync with the server:

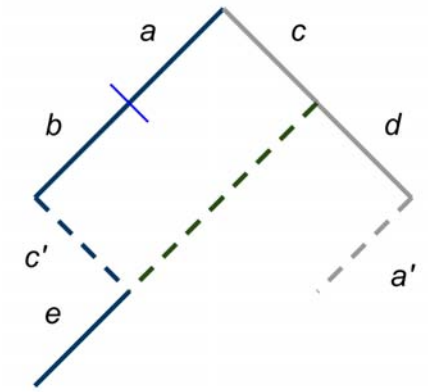


This is really getting to be a bit of a mess! We've only sent one of our operations to the server, we're trying to buffer the rest, but the server is trickling in more operations to confuse things and we still haven't received the acknowledgement for our very first operation! As it turns out, this is the most complicated case which can ever arise in a Wave-style collaborative editor. If we can nail this one, we're good to go.

The first thing we need to do is figure out what to do with d. We're going to receive that operation before we receive a', and so we really need to figure out how to apply it to our local document. Once again, the problem is that the incoming operation (d) is not parented off of any point in our state space, so OT can't help us directly. Just as with b in our fundamental compound OT example from earlier, we need to infer a "bridge" between server state space and client state space. We can then use this bridge to transform d and slide it all the way down into position at the end of our history.

To do this, we need to identify conceptually what operation(s) would take us from the parent of d to the the most recent point in our history (after applying e). Specifically, we need to infer the green dashed line in the diagram below.

Once we have this operation (whatever it is), we can compose it with e and get a single operation against which we can transform d.



The first thing to recognize is that the inferred bridge (the green dashed line) is going to be composed exclusively of client operations. This is logical as we are attempting to translate a server operation, so there's no need to transform it against something which the server already has. The second thing to realize is that this bridge is traversing a line parallel to the composition of a and b, just "shifted down" exactly one step. To be precise, the bridge is what we would get if we composed a and b and then transformed the result against c.

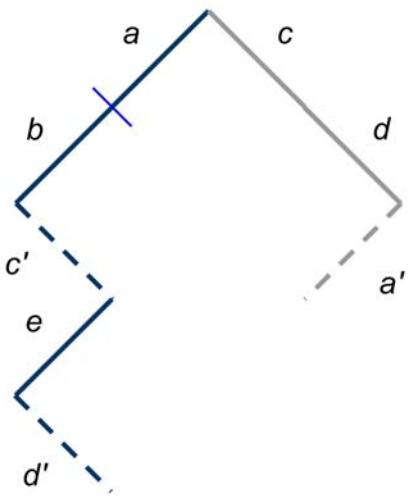
Now, we could try to detect this case specifically and write some code which would fish out a and b, compose them together, transform the result against c, compose the result of that with e and finally transform d against the final product, but as you can imagine, it would be a mess. More than that, it would be dreadfully inefficient. No, what we want to do is proactively maintain a bridge which will always take us from the absolute latest point in server state space (that we know of) to the absolute latest point in client state space. Thus, whenever we receive a new operation from the server, we can directly transform it against this bridge without any extra effort.

Building the Bridge

We can maintain this bridge by composing together all operations which have been synthesized locally since the point where we diverged from the server. Thus, at first, the bridge consists only of a. Soon afterward, the client applies its next operation, b, which we compose into the bridge. Of course, we inevitably receive an operation from the server, in this case, c. At this point, we use our bridge to transform c immediately to the correct point in client state space, resulting in c'. Remember that OT derives both bottom sides of the diamond. Thus, we not only receive c', but we also receive a new bridge which has been transformed against c. This new bridge is precisely the green dashed line in our diagram above.

Meanwhile, the client has performed another operation, e. Just as before, we immediately compose this operation onto the bridge. Thanks to our bit of trickery when transforming c into c', we can rest assured that this composition will be successful. In other words, we know that the result of applying the bridge to the document resulting from c will be precisely the document state before applying e, thus we can cleanly compose e with the bridge.

Finally, we receive d from the server. Just as with c, we can immediately transform d against the bridge, deriving both d' (which we apply to our local document) as well as the new bridge, which we hold onto for future server translations.

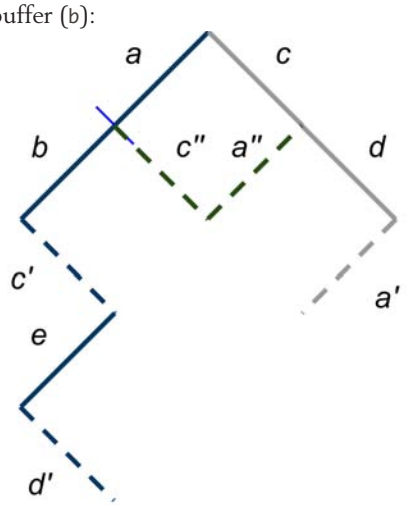


With d' now in hand, the next operation we will receive from the server will be a', the transformed version of our a operation from earlier. As soon as we receive this operation, we need to compose together any operations which have been held in the buffer and send them off to the server. However, before we send this buffer, we need to make sure that it is parented off of some point in server state space. And as you can see by the diagram above, we're going to have troubles both in composing b and e (since e does not descend directly from b) and in guaranteeing server parentage (since b is parented off of a point in client state space not shared with the server).

To solve this problem, we need to play the same trick with our buffer as we previously played with the translation bridge: any time the client or the server does anything, we adjust the buffer accordingly. With the bridge, our invariant was that the bridge would always be parented off of a point in server state space and would be the one operation needed to transform incoming server operations. With the buffer, the invariant must be that the buffer is always parented off of a point in server state space and will be the one operation required to bring the server into perfect sync with the client (given the operations we have received from the server thus far).

The one wrinkle in this plan is the fact that the buffer cannot contain the operation which we have already sent to the server (in this case, a). Thus, the buffer isn't really going to be parented off of server state space until we receive a', at which point we should have adjusted the buffer so that it is parented precisely on a', which we now know to be in server state space.

Building the buffer is a fairly straightforward matter. Once the client sends a to the server, it goes into a state where any further local operations will be composed into the buffer (which is initially empty). After a, the next client operation which is performed is b, which becomes the first operation composed into the buffer. The next operation is c, which comes from the server. At this point, we must somehow transform the buffer with respect to the incoming server operation. However, obviously the server operation (c) is not parented off of the same point as our buffer (currently b). Thus, we must first transform c against a to derive an intermediary operation, c", which is parented off of the parent of the buffer (b):



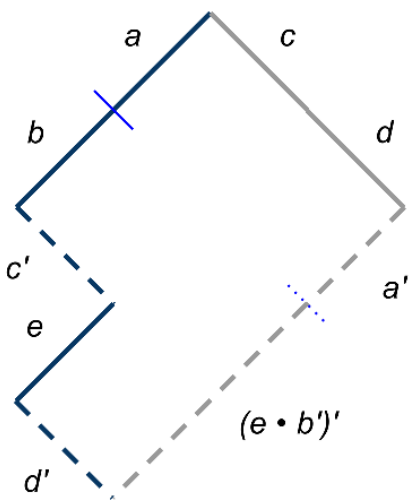
Once we have this inferred operation, c", we can use it to transform the buffer (b) "down" one step. When we derive c", we also derive a transformed version of a, which is a". In essence, we are anticipating the operation which the server will derive when it transforms a against its local history. The idea is that when we finally do receive the real a', it should be exactly equivalent to our inferred a".

At this point, the client performs another operation, e, which we immediately compose into the buffer (remember, we also composed it into the bridge, so we've got several things going on here). This composition works because we already transformed the buffer (b) against the intervening server operation (c). So e is parented off of c', which is the same state as we get were we to apply a" and then the buffer to the server state resulting from c. This should sound familiar. By a strange coincidence, a" composed with the buffer is precisely equivalent to the bridge. In practice, we use this fact to only maintain one set of data, but the process is a little easier to explain when we keep them separate.

Checkpoint time! The client has performed operation a, which it sent to the server. It then performed operation b, received operation c and finally performed operation e. We have an operation, a" which will be equivalent to a' if the server has no other intervening operations. We also have a buffer which is the composition of a transformed b and e. This buffer, composed with a", serves as a bridge from the very latest point in server state space (that we know of) to the very latest point in client state space.

Now is when we receive the next operation from the server, d. Just as when we received c, we start by transforming it against a" (our "in flight" operation). The resulting transformation of a" becomes our new in flight operation, while the resulting transformation of d is in turn used to transform our buffer down another step. At this point, we have a new a" which is parented off of d and a newly-transformed buffer which is parented off of a".

Finally, we receive a' from the server. We could do a bit of verification now to ensure that a" really is equivalent to a', but it's not necessary. What we do need to do is take our buffer and send it up to the server. Remember, the buffer is parented off of a", which happens to be equivalent to a'. Thus, when we send the buffer, we know that it is parented off of a point in server state space. The server will eventually acknowledge the receipt of our buffer operation, and we will (finally) converge to a shared document state:



The good news is that, as I mentioned before, this was the most complicated case that a collaborative editor client ever needs to handle. It should be clear that no matter how many additional server operations we receive, or how many more client operations are performed, we can simply handle them within this general framework of buffering and bridging. And, as when we sent the *a* operation, sending the buffer puts the client back into buffer mode with any new client operations being composed into this buffer. In practice, an actively-editing client will spend most of its time in this state: very much out of sync with the server, but maintaining the inferred operations required to get things back together again.

Conclusion

The OT scheme presented in this article is precisely what we use on Novell Pulse. And while I've never seen Wave's client code, numerous little hints in the waveprotocol.org whitepapers as well as discussions with the Wave API team cause me to strongly suspect that this is how Google does it as well. What's more, Google Docs recently revamped their word processing application with a new editor based on operational transformation. While there hasn't been any word from Google on how exactly they handle "compound OT" cases within Docs, it looks like they followed the same route as Wave and Pulse (the tell-tale sign is a perceptible "chunking" of incoming remote operations during connection lag).

None of the information presented in this article on "compound OT" is available within Google's documentation on waveprotocol.org (unfortunately). Anyone attempting to implement a collaborative editor based on Wave's OT would have to rediscover all of these steps on their own. My hope is that this article rectifies that situation. To the best of my knowledge, the information presented here should be everything you need to build your own client/server collaborative editor based on operational transformation. So, no more excuses for second-rate collaboration! ■

Resources

- To obtain Google's OT library, you must take a Mercurial clone of the wave-protocol repository:

```
$ hg clone https://wave-protocol.googlecode.com/hg/ wave-protocol
```

- Once you have the source, you should be able to build everything you need by simply running the Ant build script. The main OT classes are `org.waveprotocol.wave.model.document.operation.algorithm.Composer` and `org.waveprotocol.wave.model.document.operation.algorithm.Transformer`. Their use is exactly as described in this article. Please note that `Transformer` does not handle compound OT, you will have to implement that yourself by using `Composer` and `Transformer`. Operations are represented by the `org.waveprotocol.wave.model.document.operation.DocOp` interface, and can be converted into the more useful `org.waveprotocol.wave.model.document.operation.BufferedDocOp` implementation by using the `org.waveprotocol.wave.model.document.operation.impl.DocOpUtil.buffer` method.

  All of these classes can be found in the **fedone-api-0.2.jar** file.

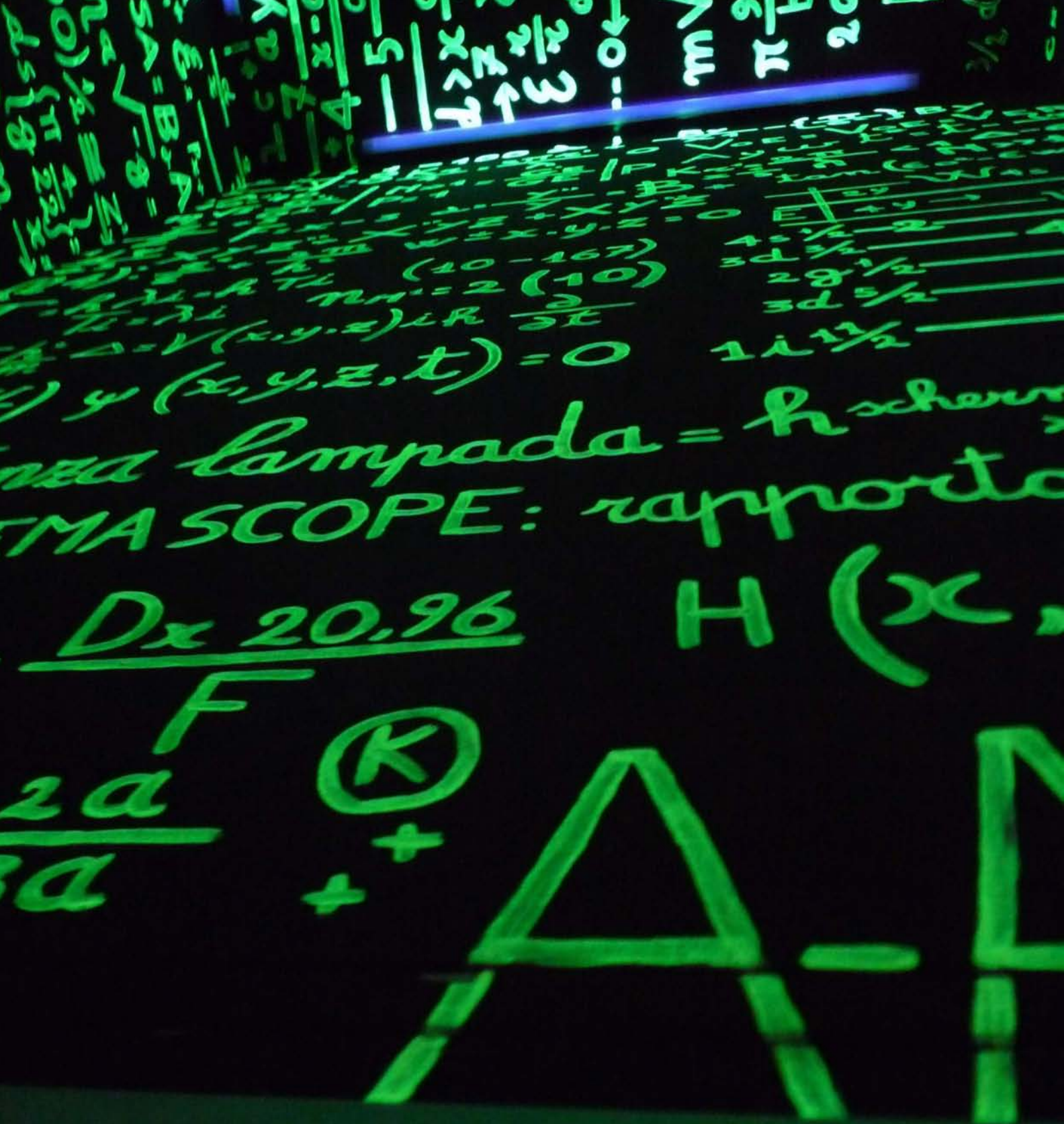
- Google's Own Whitepaper on OT: http://www.waveprotocol.org/whitepapers/operational-transform
- The original paper on the Jupiter system (the primary theoretical basis for Google's OT): http://doi.acm.org/10.1145/215585.215706
- Wikipedia's article on operational transformation (surprisingly informative): http://en.wikipedia.org/wiki/Operational_transformation

---

Daniel Spiewak is a software developer based out of Wisconsin, USA. Over the years, he has worked with Java, Scala, Ruby, C/C++, ML, Clojure and several experimental languages. He currently spends most of his free time researching parser theory and methodologies, particularly areas where the field intersects with functional language design, domain-specific languages and type theory.

Daniel regularly writes articles on his weblog, Code Commit (www.codecommit.com), including his popular introductory series, Scala for Java Refugees.

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



Math Library Functions That Seem Unnecessary

By JOHN D. COOK

This post will give several examples of functions include in the standard C math library that seem unnecessary at first glance.

Function log1p(x) = log(1 + x)

The function `log1p` computes log(1 + x). How hard could this be to implement?

```
log(1 + x);
```

Done.

But wait. What if x is very small? If x is 10-16, for example, then on a typical system 1 + x = 1 because machine precision does not contain enough significant bits to distinguish 1 + x from 1. (For details, see Anatomy of a floating point number.) That means that the code `log(1 + x)` would first compute 1 + x, obtain 1, then compute log(1), and return 0. But $\log(1 + 10^{-16}) \approx 10^{-16}$. This means the absolute error is about 10-16 and **the relative error is 100%**. For values of x larger than 10^{-16} but still fairly small, the code `log(1 + x)` may not be completely inaccurate, but the relative error may still be unacceptably large.

Fortunately, this is an easy problem to fix. For small x, log(1 + x) is approximately x. So for very small arguments, just return x. For larger arguments, compute log(1 + x) directly.

Why does this matter? The absolute error is small, even if the code returns a zero for a non-zero answer. Isn't that OK? Well, it might be. It depends on what you do next. If you add the result to a large number, then the relative error in the answer doesn't matter. But if you multiply the result by a large number, your large *relative* error becomes a large *absolute* error as well.

Function expm1(x) = exp(x) – 1

Another function that may seem unnecessary is `expm1`. This function computes $e^x - 1$. Why not just write this?

```
exp(x) - 1.0;
```

That's fine for moderately large x. For very small values of x, exp(x) is close to 1, maybe so close to 1 that it actually equals 1 to machine precision. In that case, the code `exp(x) - 1` will return 0 and result in 100% relative error. As before, for slightly larger values of x the naïve code will not be entirely inaccurate, but it may be less accurate than needed. The solution is to compute exp(x) - 1 directly without first computing exp(x). The Taylor series for exp(x) is $1 + x + x^2/2 \ldots$ So for very small x, we could just return x for exp(x) – 1. Or for slightly larger x, we could return $x + x^2/2$.

Functions erf(x) and erfc(x)

The C math library contains a pair of functions erf and erfc. The "c" stands for "complement" because erfc(x) = 1 – erf(x). The function erf(x) is known as the error function and is not trivial to implement. But why have a separate routine for `erfc`? Isn't it trivial to implement once you have code for `erf`? For some values of x, yes, this is true. But if x is large, erf(x) is near 1, and the code `1 - erf(x)` may return 0 when the result should be small but positive. As in the examples above, the naïve implementation results in a complete loss of precision for some values and a partial loss of precision for other values.

Functions Γ(x) and log Γ(x)

The standard C math library has two functions related to the gamma function: `tgamma` that returns Γ(x) and `lgamma` that return log Γ(x). Why have both? Why can't the latter just use the log of the former? The gamma function grows extremely quickly. For moderately large arguments, its value exceeds the capacity of a computer number. Sometimes you need these astronomically large numbers as intermediate results. Maybe you need a moderate-sized number that is the ratio of two very large numbers. In such cases, you need to subtract `lgamma` values rather than take the ratio of `tgamma` values.

Conclusion

The standard C math library distills a lot of experience. Some of the functions may seem unnecessary, and so they are for some arguments. But for other arguments these functions are indispensable. ■

John D. Cook is an applied mathematician. He lives in Houston, Texas where he works for M. D. Anderson Cancer Center. His interests include numerical analysis and Bayesian statistics.

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

Ruby/Rails Engineer

Backupify *(http://backupify.com)*
Cambridge Mass (exceptional candidates may work remotely)
We're looking for kick ass, agile friendly, Hacker News reading, Rails developers who are passionate about coding, not afraid of the command line, and love working with emerging technologies. Backupify is venture-funded and you'll enjoy working on a loaded MacBook Pro with a solid team. This is a full-time position.
To Apply: Send links of past projects to *jobs@backupify.com*.

Software Engineer

ExtraHop Networks
(http://www.extrahop.com)
Seattle
About You: You're a good programmer. You're skilled or interested in some or all of the following: networking, systems programming, C programming, Python programming, UI/UX design, and problem solving. You enjoy a startup environment, which means getting things done, not being specialized, and not attending many meetings.
To Apply: Please send us an email at *jobs@extrahop.com*.

PHP Developer

Altruja GmbH *(http://www.altruja.de)*
Munich, Germany
We are looking for a PHP Developer, who should have experience developing Web-Applications (LAMP, MVC, Symfony). Knowledge of (X)HTML/CSS/JS would be good, too. We're a Start-Up with funding and offer a nice work environment and a great team.
To Apply: *jobs@altruja.de*.

Systems Administrator

Berklee College of Music
(http://www.berkleemusic.com)
Back Bay, Boston, MA
Berklee College of Music's online extension school is looking for a sysadmin to join their Operations team to support our server infrastructure. Position requires knowledge of Linux, Amazon AWS, storage systems and deployment of high availability websites. Off-hours support required.
To Apply: Send resume and cover letter to *work@berkleemusic.com*.

Security Researcher

Harris Corp – Crucial Security Programs *(http://www.harris.com/csp)*
Various/US
Security Researcher: User & kernel-mode C/C++ x86 assembly, reverse engineering, debugging, & os internals; automated executable analysis, virtualization, emulation engines; pen testing; research; low-level SW protection methods & executable dissection algorithms; writing for highly technical audience; security components & debug tools; malware, rootkits, protection schemes, & virtualization theory; Clearable.
To Apply:
kirsten.renner@harris.com.

Android Developer

Bump Technologies, Inc.
(http://bu.mp)
Mountain View, CA
Design and build the next generation of Bump's Android app and API. Requirements: Expert in Java (Python and C/C++/Obj-C are a plus), experience developing native Android apps or other mobile apps is great but not required, an eye for great design and user experiences, and ability to communicate within a close-knit team.
To Apply: Email *jobs@bu.mp*.

Software Engineer

XIA LLC *(http://www.xia.com)*
Bay Area
We make processing electronics for x-ray and gamma-ray detectors and our software controls precision instruments. Things we like: C, C#, F#, Ruby, Lisps, Emacs, hg, flexible work hours and Hacker News. Don't get too hung up on the specific languages: we are looking for skilled developers and are always open to using new languages in our projects.
To Apply: Resume and cover letter to *jobs@xia.com*.

C++ or Java Developer

Capital Markets Placement
(http://www.CMP.jobs)
New York, Chicago and Washington, DC
Several investment banks, one hedge fund, one publishing and one interactive client company. Hiring very actively C++ or Java developers. C++ positions paying $100-$250k base, depending on background and in lieu with prior compensation numbers, for example trading systems will be compensated to the max. For Java $90-140k, with Spring/Hibernate and Javascript.
To Apply: Either apply directly through website or contact *boris@cmp.jobs* with a resume or questions.

Senior Systems Architect

MySpace *(http://www.myspace.com)*
Beverly Hills
The Systems group is looking for a hacker architect. You have autonomy to do awesome things. Your C#/C/C++ is masterful, but you use Ruby/Powershell/Python where it makes sense. Same goes for assembly. You are equally comfortable in front of a whiteboard or windbg/gdb. Flexible hours, good pay and free food. Will relocate.
To Apply: `6362656c6c406d797 3706163652d696e632e636f6d`

Awesome Java Developer

Dubit Limited
(http://dubitplatform.com)
Leeds, Yorkshire, United Kingdom
We're looking for a multi-purpose, penknife of a developer, who can handle maintaining our server side technology and integrate it with our Flex front end. You'll need to be experienced in Java, with a computing related degree, and you'll need to know what wait/notify are for too. Any experience with socket programming or servlets is a plus.
To Apply: Send your CV and covering letter to *thomas.williams@dubitlimited.com*.

Linux Systems Administrator

Simply Hired

(http://www.simplyhired.com)

Mountain View, CA

Simply Hired is seeking a great Linux Sys Admin to help take our production environment to the next level of stability, scalability, automation, and transparency. Experience in high scale web site operations; Linux/Unix / open-source, Perl or Python; x86 servers, switches, routers; LAMP/Java/open-source stacks.

To Apply: *http://hn.my/simplyhired*

Sr. Java Software Engineer

Clearspring Technologies

(http://www.clearspring.com)

McLean, VA

We're looking for an extremely talented engineer to help us tackle some of the largest-scale data processing challenges around. Clearspring's tools are seen by over 1 billion people across the web every month. From that data, you'll be unraveling the mysteries of web-wide social behavior and delivering powerful insights to publishers.

To Apply: *jobs@clearspring.com*.

Freelance Front End Web Developer

Butchershop Creative

(http://butchershopcreative.com)

San Francisco, CA

Butchershop, a San Francisco marketing and creative agency is looking for an imaginative, innovative, visual, organized, self-motivated individual to help with the following: Design, Color, Typography, Composition, Illustration, UI/UX, HTML, CSS, JavaScript. Bonus if you have used Python or PHP.

To Apply: Email *info@butchershopcreative.com*.

Senior Software Engineer

Democratic National Committee/Organizing for America

(http://www.barackobama.com)

Washington, DC

The Democratic National Committee is seeking a talented software engineer to empower communities across the country. The Senior Software Engineer will build and maintain a variety of open, scalable web services that will be critical pieces of party infrastructure.

To Apply: Email *techresume@dnc.org*.

UI and Software Engineers

Meetup *(http://www.meetup.com)*

New York, NY (Soho)

Meetup is fast-growing, venture capital-backed, just-turned-profitable + a great place for top talent to do their best work. We're hiring exceptional engineers (QA, UI, Software & Systems) with web experience to work on a product that's already helped millions of people find + build local community worldwide.

To Apply: *http://www.meetup.com/jobs*

Freelance Cake Baker

FlickEvents

(http://www.flickevents.com)

Internet or Singapore

FlickEvents is a startup in Singapore that focuses on tools that make life easier for event organizers, and to make this tradtionally boring industry a little bit more bearable. We are looking for a freelance developer who does CakePHP 1.2 and above, and who understands design patterns, SimpleTest or TDD practices, and Git.

To Apply: *job@flickevents.com*.

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*.

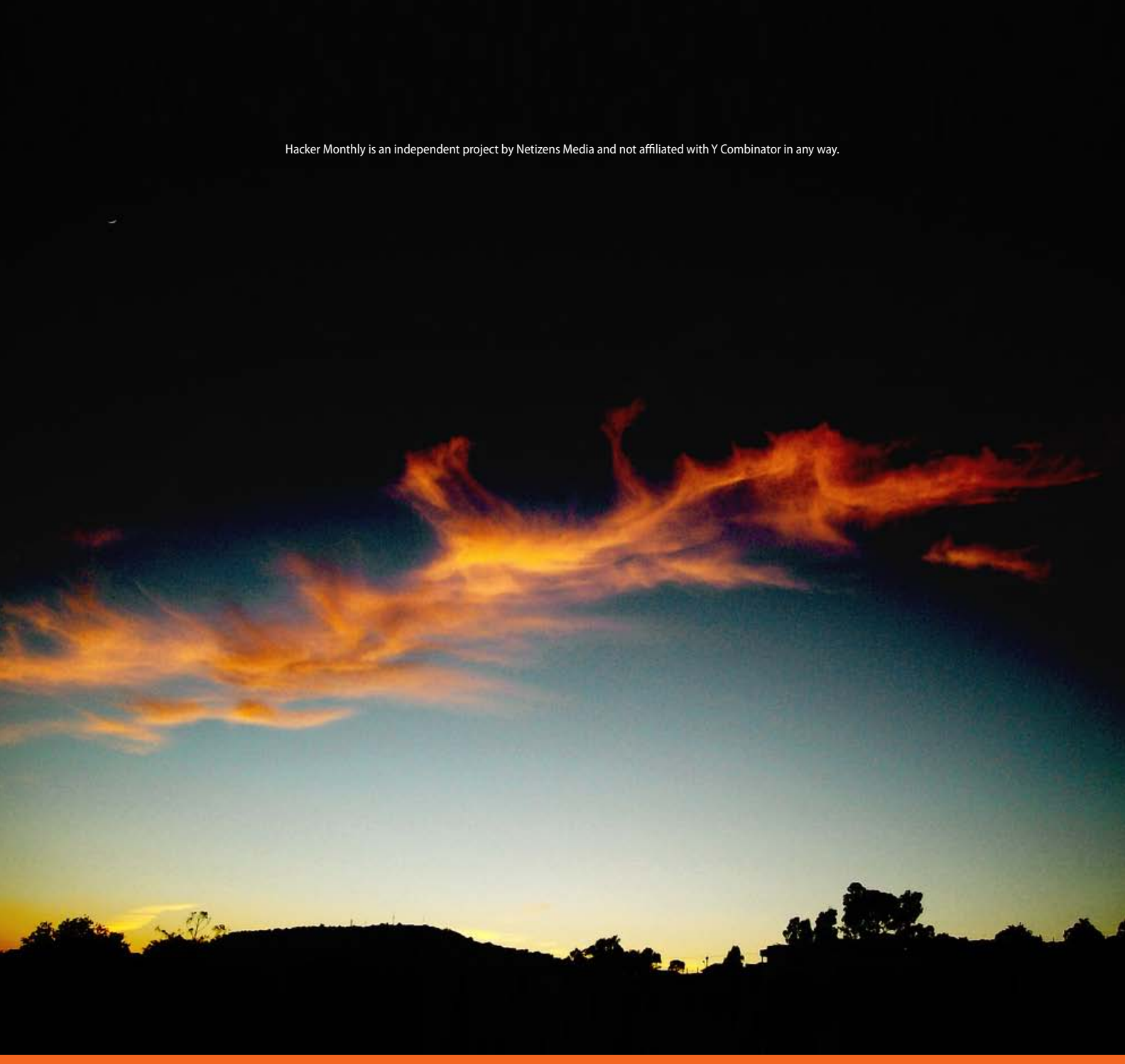
Senior Software Engineer

F5 Networks *(http://www.f5.com)*

Seattle

Interested in taking your software engineering career to the next level? Ever wonder what 100 Gb/s of TCP traffic looks like? In GDB? We are looking for top engineers with system-level C, networking, protocol and kernel expertise. Join our team of super smart engineers working in a fun and fast-paced highly technical environment where your ideas become part of the solution!

To Apply: *http://hn.my/f5*



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/