HACKERMONTHLY Issue 43 December 2013

How I Failed

Tim O'Reilly

Curator
Lim Cheng Soon

Contributors
Tim O'Reilly
Jeff Wofford
David Nolen
Dennis Kubes
Dominic Szablewski
John Croisant
Felix Winkelmann
Francois Zaninotto
Bemmu Sepponen
Dave Gooden

Proofreaders
Emily Griffin
Sigmarie Soto

Illustrator
Joel Benjamin

Ebook Conversion
Ashish Kumar Jha

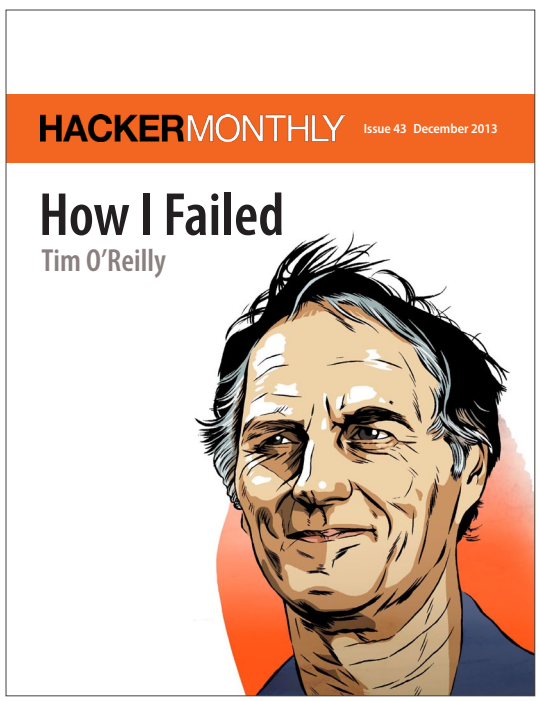
Printer
MagCloud

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

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

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



Cover Illustration: Joel Benjamin

Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-43*

How I Failed

Six Lessons Learned.

By TIM O'REILLY

WHEN YOU START out as an entrepreneur, it's just you and your idea, or you and your co-founders, and your idea. Then you add customers, and they shape and mold you and that idea until you achieve the fabled "product-market fit." If you are lucky and diligent, you achieve that fit more than once, reinventing yourself with multiple products and multiple customer segments.

But if you are to succeed in building an enduring company, it has to be about more than that: it's about the team and the institution you create together. As a management team, you aren't just working for the company; you have to work on the company, shaping it, tuning it, setting the rules that it will live by. And it's easy to give that short shrift.

At O'Reilly Media, we've built a successful business and had a big impact on our industry. But looking back, it's clear how often we failed. Some were failures of vision, some failures of nerve, but most were failures in building and cultivating company culture.

What do I mean by culture? Atul Gawande summed it up perfectly in his recent New Yorker article "Slow Ideas." You have a culture when "X is what people do, day in and day out, even when no one is watching. "You must" rewards mere compliance. Getting to "X is what we do" means establishing X as the norm."

What I Got Right

I did a good job setting the company goals: "Work on stuff that matters," "Create more value than you capture," "Change the world by spreading the knowledge of innovators." Our principles have been the lodestone that led us into new markets.

We were originally a technical writing consulting firm, but our desire to tell the truth about what works and what doesn't (rather than telling the story as the product manufacturer wanted it told) led us to publish our own books. We wanted those books to be available online, so we began working with eBooks all the way back in 1987. Influenced by the ideals of the free software movement, we didn't want those books to be hostage to proprietary software, so we worked on standards for interoperability (what became Docbook XML) and adopted the Viola browser (the first graphical web browser) as a free online book reader.

Working with Viola led us to the web, and we got so excited about it that we went out on a limb to include it at the last minute in the book we published about the Internet in 1992, The Whole Internet User's Guide and Catalog, even though there were only 200 websites at the time. The book sold a million copies.

When Barnes & Noble or Borders returned books to us, stickered and unsalable, we didn't pulp them; we sent them to Africa, where they could be useful to people who couldn't afford them. We astounded publishing competitors in the early '90s with our UNIX and X Bibliography for bookstores, a marketing piece that included their books as well as our own. We wanted to build the market, and so highlighted the best books, not just our own. We have followed that same logic in building our digital distribution business today, reselling eBooks from other technology publishers as long as they agree to go DRM-free.

We started Safari Books Online as a joint venture with our biggest competitor because we believed publishers needed to find new business models in an electronic future, and we thought that the models we were inventing would be adopted more widely if they included books from multiple publishers. We have worked tirelessly on DRM-free eBooks because we believe that locking books up in proprietary file formats is a path toward a digital dark age.

Our quest to give voice to new movements and communities led us to invest for seven years in Make Magazine and Maker Faire before the rest of the world took notice and came to the party. We published books on life-changing diseases as well as life-changing technologies (Childhood Leukemia, Childhood Cancer Survivors) until the dot com collapse of 2001 led us into drastic retrenchment. We have returned to health care with our StrataRx Conference, because there is a unique opportunity to apply data to make the health care system more effective and to improve people's lives.

We've done the same thing with open data in government, advocating the idea that government at its best acts as a platform, working to bring citizens, civil servants, and entrepreneurs together to solve problems.

And through all this, we built a profitable group of enterprises with nearly 500 employees and collective revenues approaching $200 million.

So what's not to like?

We could have been even more effective by paying attention to some key management skills.

In that spirit, here are some reflections on how we failed as an organization in the past, and what we have been doing to change that.

Failure #1: People hear only half the story

There's a great moment in a Michael Lewis interview on NPR. Why, Lewis was asked, would anyone in the financial industry talk to him for his book The Big Short after the devastating picture of Wall Street he'd painted in his first book, Liar's Poker, nearly 20 years earlier? Lewis replied that many of those people got into the financial industry after reading his book. Their big takeaway was how easy it was to make a lot of money without regard to the niceties of creating much value. He finished with the memorable line, "You never know what book you wrote until you know what book people read."

That turned out to be a major problem for me at O'Reilly. I talked so much about our ideals, our goal to create more value than we capture, to change the world by spreading the knowledge of innovators, that I forgot to make sure everyone understood that we were still a business. Even when I said things like, "Money in a business is like gas in the car. You have to fill the tank, but a road trip is not a tour of gas stations," people heard the "road trip is not a tour of gas stations" way louder than they heard "you have to fill the tank."

> **"If I were starting O'Reilly all over again, I'd spend a lot more time making sure the culture I was trying to create was the one that I actually created."**

As a result, we've had countless struggles to have employees take the business as seriously as they should. I was always pretty good at finding the sweet spot where idealism and business reality meet, but I didn't spend enough time teaching that skill my team. And I didn't check in enough about what people were actually hearing.

Reflective listening is an important skill. If I were starting O'Reilly all over again, I'd spend a lot more time making sure the culture I was trying to create was the one that I actually created.

Failure #2: "That's how it's done"

In the early days of the company, I wrote an employee manual that reflected my own homegrown HR philosophy, based on the idea that I wanted everyone in the company to have the same freedom, initiative, and excitement about our work that I did; it opened with this statement:

> *"I called this booklet "Rules of Thumb" because every rule in it is meant to be broken at some time or another, whenever there is good reason. We have no absolute policies, just guidelines based on past experience. As we grow, we will learn, and will make new empirical rules about what works best in new situations."*

It also said things like:

> *"Bring yourself to your work! We haven't hired you to act as a cog in the company machine, but to exercise your intelligence, your creativity, and your perseverance. Make things happen."*

And:

> *"Remember, too, that your job isn't just an opportunity to improve your economic standing, or that of the company, but to make yourself a better person, and this world a little better place to live. Each of your co-workers, our customers, our suppliers, and anyone else you deal with is a person, just like you. Treat them always with the care, fairness, and honesty that you'd like to experience in return."*

The only raises we had were merit raises, as you improved your skills and impact. You were expected to manage your own time, with no set hours, and the only responsibility around vacation time was to make sure that no balls got dropped.

Eventually, I hired an employment lawyer to review my draft, and he said, "That's the most inspiring employee manual I've ever read, but I can't let you use it."

I complained, but I eventually gave in. As we grew, it was harder and harder to maintain our informal processes. (I remember a real inflection point at about 50-60 employees, and another at about 100.) We gradually gave up our homegrown way of doing things, and accepted normal HR practices — vacation and sick days, regular reviews, annual salary adjustments — and bit by bit, I let the "HR professionals" take over the job of framing and managing the internal culture. That was a mistake.

I've often regretted that I hadn't kept fighting with the lawyers, working harder to balance all the legal requirements (many of them well-intentioned but designed for a top-down command-and-control culture) with my vision of how a company really ought to work. I focused my energy on product, marketing, finance, and strategy, and didn't make sure I was building the organization I wanted.

> **“O’Reilly was like a leaky bucket. We were always profitable on a P&L basis, but we never had enough cash.”**

Reading recently about the HR practices at Valve and GitHub, so reminiscent of early O’Reilly, I’m struck by the need to redefine how organizations work in the 21st century. I’m not saying that Valve or GitHub’s approach is for everyone, but they indicate a deep engagement with the problem space, and fresh approaches managing an organization. Google’s People Analytics may be a more scalable application of new HR thinking to a company of serious size.

While there’s a lot of accumulated wisdom in how to run a company, there’s a lot still to be invented, and you should bring the same entrepreneurial energy to improving the culture as you do to improving the product.

Failure #3: Cash and control

In today’s venture-capital-fueled market of “build it and see if they will come,” it’s often hard to remember that there are businesses built without investors, funded by revenue from real customers. I never took VC money because in my early days as a tech-writing consultant, I saw lots of companies go from being great places to work to being just another company, and I wanted to keep control of what I did and did not do.

I wanted control, but I missed one of the most powerful ways to have it.

Bill Janeway is the author of the outstanding book Doing Capitalism in the Internet Economy. In it, he recounts the lesson of one of his own mentors, Fred Adler, “Happiness is positive cash flow,” and talks about his working principle of “Cash and Control”: “assured access to sufficient cash in time of crisis to buy the time needed to understand the unanticipated, and sufficient control to use the time effectively.”

I learned the truth of Bill’s statement about cash and control in the ‘90s. Publishing is a fairly cash-intensive business. You pay advances to authors — many of whom never come through with the books they promised to write, or take way longer to complete them — and as your editorial, design, and production teams work hand in hand with the author, you may have years of investment before you see a penny back. And in the old days, before eBooks and print-on-demand, you then had to invest tens of thousands of dollars in inventory costs for each book.

O’Reilly was like a leaky bucket. We were always profitable on a P&L basis, but we never had enough cash. And as our publishing business accelerated through the “90s, we needed more and more of it. We borrowed against our receivables and our inventory, juggled payables till our CFO was blue in the face, but we ended up funding our growth through equity exits from companies we”d spun out and sold or invested in.

We’d sold GNN to AOL for what seemed at the time the princely sum of $15 million, much of it in stock. We were locked up for a couple of years, but because of our pressing cash needs, we had to sell our stock as soon as it became available, netting $30 or $40 million because of the increase in AOL’s value as the Internet bubble inflated. That was a nice win, but if we’d had the leisure to hold on till the peak, our stock would have been worth $1 billion, and even if we hadn’t timed things perfectly, several hundred million.

Where the shit really hit the fan was after the dot com bust of 2001. We were seriously in debt again, our business was in free, our banks pulled our loans and nearly put us

out of business. I still remember the day I had to decide which employees to cut in our first-ever layoffs. As I pored over the worksheets, I noticed hair all over my papers; I was so stressed that my hair was falling out.

It didn't need to be that way.

In the depths of the crisis, we hired a CFO who instituted new financial controls and discipline. She renegotiated contracts with suppliers. She ruthlessly cut non-performing titles, freeing up the cash from inventory. And she persuaded me to do the layoffs rather than going down with the ship and all hands.

The difference was enormous. We rebuilt O'Reilly's revenues and profits through successful new books and conferences, the growth of Safari Books Online, Maker Media, and other new businesses. But the biggest impact was the one that Laura had — on our cash.

There are four lessons here:

1. **Financial discipline matters.** If you're a venture-backed startup, financial discipline gives you more control over when you have to go out for that next round. If you're self-funded, financial discipline lets you invest in what's important in your business. So many companies agonize over the quality of their product, and work tirelessly to build their brand, yet pay the barest attention to their financials. Money is the lifeblood of your business. Take it seriously. Manage it well.

2. **Treat your financial team as co-founders.** They aren't just bean counters. They can make the difference between success and failure. Don't just look for rockstar developers or designers, look for a rockstar CFO. Hire someone who is better than you are, who can be a real partner in growing the business. Before Laura came on board, I was always the most numerate person in the organization, the one with the most sensitive finger on the pulse of our financials.

3. **Hold teams accountable for their numbers.** Every manager — in fact, every employee — needs to understand the financial side of the business. One of my big mistakes was to let people build products, or do marketing, without forcing them to understand the financial impact of their decisions. Anyone running a group with major financial impact should have their P&L tattooed on their brain. It isn't someone else's job to pay attention. Make sure financial literacy is part of your employee training, and hold people accountable for their numbers.

4. **Run lean; reinvent tirelessly.** After the bust, we laid off 20% of our staff, and while we missed many of them intensely on a personal level, as a business, we didn't skip a beat.

The Lean Startup methodology emphasizes measurement in quest of product-market fit, describing a startup as "a machine for learning." This is great. But you need to turn these measurements not just outward on the market but inward on your organization. What is the impact of each activity? Who could be repurposed toward something with greater impact? Does this job really need doing? Can it be done more efficiently and effectively?

Failure #4: Tolerating mediocrity

There was another lesson learned from those 2001 layoffs. While most of the people we laid off were great employees who went on to find good jobs elsewhere, we were appalled to discover there were some people who had built themselves a nice, cozy position but weren't working very hard. While most of us were pulling the wagon, they were simply riding on it. We even discovered several cases of fraud! That goes back to my point above about the importance of a crack financial team — one of their key jobs is to have strong controls in place. I would never have believed that one of my employees would do that. It can happen to any company. The longer you are in business, the more outrageous things you will have employees do on your watch!

Looking back, I had an extremely naive view: everyone was inspired by the same motivations as I was, passionate about their work and the impact that we were having. They loved their jobs and wanted to be great at them.

If you want that to be true, you can't just believe it; you have to work at it! You need a real emphasis on hiring, training, and mentorship — and firing! Every manager in the company has to be an expert on his or her staff and on finding "employee-company fit." HR needs to be an active partner in talent acquisition, culture, and leadership development.

When someone isn't right for the job, it's easy to shrink from the confrontation of telling them so, or to accept 60% of what you wanted because you think you can't afford the time and trouble to find a replacement. You aren't doing anyone any favors. An employee

who is not performing at 100% is just as aware as of that, and most likely isn't happy about it. Having the courage to ask them to move on is an essential management skill. (It doesn't even have to be firing; it can be coaching them to make the decision on their own.)

So, if you have a bad feeling about the role someone is playing in your organization, work the issue until you feel right about it. Take management seriously!

Failure #5: Hiring supplements, not complements

Another thing I wish I'd done earlier was to hire people who were good at things I wasn't. As a founder, you often seem to be the best at everything — the best product designer, the best marketer, the best sales person. Sometimes that's really true, but often it's just because you hire people who aren't as good as you are at the things you're good at, and don't hire people who are better than you are at the things you don't do so well. You hire supplements to do more of what you already do, rather than people who really complement your skills.

I already mentioned how I went through the first 20 years of my company's life without hiring someone who was better on the financial side than I was. We didn't build a sales and marketing culture either. We were product driven, idea driven, and while we developed a unique and powerful style of activism-driven marketing, we never developed analytical marketing discipline. And as for sales, that felt a little dirty to many of our employees.

In the past few years, we've worked hard to change that. Laura has led a successful effort to develop that analytical marketing competency and to add sales thinking to the company DNA. We now have sales training for anyone who has customer contact. We've built a team to focus on sponsorship sales for our events, more than doubling our yield and vastly improving the profitability of our events.

Failure #6: I'll take care of that

I believe it was Harold Geneen who once said, "The skill of management is to achieve your objectives through the efforts of others." Yet, like so many entrepreneurs, my first instinct was not to hire the team to go after a new product or market, but to do it myself, or with the team I already had.

Some of that was a byproduct of being a scrappy, self-funded organization, where the existing team tries new things and hires only after it is clear there's really an opportunity. It's great when your management team leads from the front. But overall, we took it too far and didn't build a strong enough culture of deliberate hiring to go after new opportunities.

Anthropologist Claude Levi-Strauss wrote in his book The Savage Mind about the difference between the bricoleur (handyman) and the engineer. The handyman makes do with what he has at hand. The engineer thinks more abstractly, figures out what he or she needs, and acquires it before beginning work. I was always a bricoleur. As we go forward, I aspire to be more of an engineer. Although it's good to remember that, as Marc Hedlund, former SVP of product development and engineering at Etsy, remarked, "People and code are…different. The approaches that work so well for getting new software to run are not directly applicable to getting people to work well together." ■

Tim O'Reilly is the founder and CEO of O'Reilly Media Inc., thought by many to be the best computer book publisher in the world.

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

What Programming a Game in 48 Hours Taught Me About Programming Games

By JEFF WOFFORD

I participated in Ludum Dare 27 this weekend, programming a complete game, Spacetime Adventure, in 48 hours. I make games for a living, but I'd never done that before. It was fun.

It was also enlightening. For the past several years I've spent most of my development time with C++11 in Xcode. I like it. Nah, I love it. But this weekend, working in Adobe Flash Professional with ActionScript 3.0, I could not believe how high my productivity was. I was knocking off tasks like they were popcorns in a fire. It helps that I used to work in Flash a lot, so I knew the drill. But I had forgotten how easy and quick it is to make games in that system.

The contest limit is 48 hours, but I actually spent 30 hours. In that time I made an entire game, and not a terribly simple one: it involves Box2D physics and time travel. It's not a highly polished game, of course. I'm going to work on it some more before really "releasing" it (though you can play it now if you want). But it has all the main bells and whistles: front end, HUD, user interface, the game proper, victory screens — even music. Not that that's anything special — the contest is to make a complete game, and over a thousand contestants did so.

Yet most of the games I make in my professional job take much longer than this. As I reached the end of the weekend, I couldn't help but ask myself, "How is it that you were able to complete this game in less than 48 hours, when most of the games you work on take upwards of several months?"

The last game I shipped, House of Shadows, took 11 months. Even if you assume that it was 10 times more complicated than Spacetime Adventure, this still leaves a productivity ratio of about 6:1. This means that if I could transfer the pace of production from Ludum Dare into my normal work, I would complete a game like House of Shadows in less than 2 months.

Now no doubt some of the differences between a Ludum Dare project and a "real" project are esoteric and non-transferable. House of Shadows, for example, is really probably more than ten times more complicated than Spacetime Adventure, thinking in terms of the internal game mechanics, rules, variation, and user interface. Spacetime Adventure gets away with being pretty simple really. But along with this kind of non-transferable difference, perhaps there are other differences that are transferable. Maybe there are things about creating a 48 hour project that can make a "real" project faster and maybe even more fun.

Differences

Intensity of focus. Almost all my waking hours were dedicated to programming during the 48 hours of the contest. I even took less sleep. This intensity of focus allowed me to maintain contact with the concepts and issues in the game so that I was able to remain productive without costly ramp-up and ramp-down times.

Expectation of constant closure. I expected to be done fast. At the macro level for the game as a whole. At the micro level for individual tasks. I was not at all happy with tasks, bugs, or setbacks that threatened the rigid deadline. I expected to make rapid, constant progress, and I made sure I did.

Freedom from IDE distractions. One of the worst hits to productivity in my usual development setup — although fortunately this is not a daily problem — is when some aspect of the tools themselves go so slow that they lose my attention. If I have to Rebuild All, or work with a slow Photoshop, or if Xcode is hanging and crashing, not only does this cost time immediately, but it also causes me to get distracted. I try to fill the time by checking email or Hacker News, and this costs yet more time. During Ludum Dare, I remained tightly interfaced with Flash. I was continually in the midst of the edit -> compile -> test -> edit loop. This was one of the largest reasons for the high pace of production. The IDE did not kick me out at any time. It gave me no reason to look away. I need that in my daily work.

Easy object placement and animation tools. The UI work in particular went incredibly fast and this was entirely due to working in Flash. I could drag a bitmap into Flash to import it, then place it, position it, add filters, animate it, and attach the animations to code all in one tight motion, all within Flash. Tasks that can take a whole day took minutes. I need this all the time.

Lower degree of polish. Spacetime Adventure is reasonably complete but it's not a final, shippable, polished game. Part of the slowness of a normal, professional game project is the degree of polish that goes into the product. As a rough estimate, I'd say that polish approximately doubles the length of a project. If I had to add features like sound effects, particles, more UI animations, button states, higher-quality art, variety of art, and additional gameplay features to Spacetime Adventure, it would have taken at least twice as long.

No responsibility for maintenance. Creating a maintainable game — one that is capable of long-term repair and expansion — is more difficult than a quick, throw-away game like Spacetime Adventure. In actual fact, Spacetime Adventure's code is generally pretty clean and maintainable, but this quality happened to come easily, in part because of the smallness of the game; it wasn't hard-won. And there is "slop" in the code that I would not have been comfortable with if I expected to have to live with the game for longer or expand it much larger. When you can write sloppy, get-it-done code, it pays to do so. When you can't, it doesn't. With Ludum Dare you always can. With production code, you rarely can.

C++ headers. More than once during the competition I would reach a point in the code and think, "Argh, I don't want to have to add/change/look up/remove that function because it would mean having to mess with the header file." Then I thought, "Oh wait, this isn't C++. There are no header files." The feeling of liberation and simplicity that hit me in those moments convinced me that for a great deal of coding situations, headers are a serious bane. They impart a constant agony of redundancy onto everything you write. Every substantial (i.e. semantic) change must touch two files, and do so in a coordinated way. The simplicity and immediacy of single-file ActionScript classes felt like a breath of fresh air.

There is a place for the header/source division. For established code, dividing classes makes for faster compilation for both the user and the provider. This is rarely an issue in "game programming" proper (as opposed to "engine programming").

This point, along with a few others in this list, convinces me once and for all that scripting languages are the way to go for most game programming. When these aren't available, my friend Wouter van Oortmerssen's Java Style Classes in C++ may provide a handy workaround. I'm thinking of trying it for my current project's game code.

> “I felt more than ever the liberation that comes when you don’t have to dance the C++ dance.”

Performance and safety obliviousness. I know this is an old lesson that needs no explication, but I was struck more forcefully than ever how C++ imposes a significant mental cost on programmers to use the language carefully. This sounds like more of a bash against C++ than it really is. You use C++ precisely when you need high performance. The reason I normally program games in C++ instead of in Flash is that my performance testing of ActionScript reveals that it is at least an order of magnitude too slow for the kind of games I make on the kinds of platforms I normally make them for. I like C++ because it gives me many of the benefits of ActionScript (and other high level languages) while enabling lightning speed performance.

But this weekend I felt more than ever the liberation that comes when you don’t have to dance the C++ dance. When deleting something I simply set the reference to null. I can do the same thing in C++ by using std::shared_ptr, but even then one still has to be mindful of cycles. The word “mindful” here is not as innocent as it sounds. The detection and anticipation of object graph cycles while in the middle of coding is non-trivial. A programmer’s chief resource is the energy of his or her mind. Everything that expends or depletes that energy makes him or her less effective, more tired, and less happy. There were several moments during the competition when I thought, “I need to delete this expensive resource. I’ll set it to null. Ah, but are there any cycles that might keep it afloat?” And then I remembered, “Yes, but the whole cycle will die along with it.” There was a palpable feeling of relief when I realized that I didn’t need to worry about the cycles. I could use that mental energy to focus on the game itself.

It’s not just memory management. The whole context of Flash/ActionScript made me less concerned about performance. I know Flash is slow. At the beginning of the project I did some testing and confirmed that it was fast enough. From that point on I never worried about performance again.

It’s remarkable how subtle and constant the performance concern is. A good C++ programmer — especially one working on a relatively slow platform like mobile phones — is continually assessing the cost of what he or she is writing. Should I use a vector here? A map? An unordered map? Will it be faster to pass this argument by reference? Should I reserve() this vector so that it doesn’t overshoot its necessary size? You use C++ because you want to squeeze frame rate out of tightly constrained hardware. Every variable, every function becomes a potential choke point, and a seasoned programmer is always measuring the ramifications of each choice. The C++ programmer is a deer sniffing the air for the scent of boots and gunpowder: everything’s an opportunity for gain; everything’s an opportunity for calamity.

When performance is of the essence, this state of alertness is an appropriate price to pay. But when you don’t have to pay that price — and in every game there are systems that have no serious likelihood of bottlenecking — you will gain mental energy back by essentially ignoring performance. You cannot do this in C++: it requires an awareness of execution and memory costs at every step. This is another argument in favor of never building

> **“You don’t have time to talk about what you’re doing — you think fast, then you act.”**

a game without a good scripting language for the highest-level code. In ActionScript I fell into an easy rhythm of doing what I needed to do for the game behavior. I did not worry about the cost of an Array vs. a Vector: I used what was convenient. I felt a little lazy being so carefree. But the approach cost me nothing: the game runs like butter even on older desktop systems.

Minimal Snowballing. In the broadest sense, a 48 hour project minimizes a problem that plagues all projects. Work tends to snowball. For every task there are “task addendums” that extend the total effort. It’s not enough just to put an asteroid into the game. Beforehand you have to design the asteroid, talk about the asteroid, and schedule the asteroid. Once the asteroid is written you have to test the asteroid, commit the code, adjust the asteroid, review the code, adjust it some more, document it, adjust the comments, fix the commit, refactor, optimize, extend. This sounds like the standard complaint about project management: projects should be simple but management adds cruft. Yet in some sense any project — even one undertaken by a single person — is susceptible to snowballing. It’s an odd thing, hard to put your finger on. Every task begets more tasks at the code level (typing, commenting, optimization) and the quality level (testing, debugging, refining).

It’s almost mathematical. For every hour you spend working, you must spend another 10 minutes responding to or expanding that work. After six hours of working you have accumulated an additional 1 hour of this metawork, which of course — being work — needs its own 10 minutes of response and expansion. Six hours of metawork later, you’ve accumulated an hour of metametawork, which needs yet another layer of response and expansion, and so on. Each layer of metawork is another layer of snow on the snowball. The larger the tasks get, the larger the tasks get.

In a 48 hour project this cycle is defeated — or at least minimized — by the sheer concentration of focus. There are no “metatasks” — there are only tasks. You don’t have to re-learn what you did yesterday, because there was no yesterday. You don’t have to plan for next week. You don’t have time to talk about what you’re doing — you think fast, then you act. This can’t be the best way to accomplish just any project, but when it’s possible it is incredibly efficient, and that efficiency is incredibly satisfying. ■

Jeff Wofford has worked in game development since 1995. Currently he is the Duke of Development for Mobile Games at Armor Games and a lecturer at Southern Methodist University’s Guildhall game development program.

Reprinted with permission of the original author. First appeared in *hn.my/48hrs* (jeffwofford.com)

ClojureScript 101

By DAVID NOLEN

While none of the ideas in `core.async` are new, understanding how to solve problems with CSP is simply not as well documented as using plain callbacks or Promises. My previous articles have mostly explored fairly sophisticated uses of `core.async`, this one instead takes the form of a very basic tutorial on using `core.async` with ClojureScript.

We're going to demonstrate all the steps required to build a simple search interface, and we'll see how `core.async` provides some unique solutions to problems common to client-side user interface programming.

I recommend using Google Chrome so that you can get good source map support. You don't need Emacs to have fun with Lisp. SublimeText 2 is pretty nice these days, I recommend installing the paredit and lispindent packages via Sublime Package Control.

If you have Leiningen installed you can run the following at the command line in whatever directory you like:

```
lein new mies async-tut1
```

This will create a template project so you don't have to worry about configuring `lein-cljsbuild` yourself.

Unless otherwise noted, files are relative to the project directory.

Change the `:dependencies` in the `project.clj` file to look like the following:

```
:dependencies
    [[org.clojure/clojure "1.5.1"]
     [org.clojure/clojurescript "0.0-2030"]
     [org.clojure/core.async "0.1.256.0-1bf8cf-
alpha"]] ;; ADD
```

In the project directory run the following to start the auto compile process:

```
lein cljsbuild auto async-tut1
```

First off we want to add the following markup to `index.html` before the first script tag which loads `goog/base.js`:

```
<input id="query" type="text"></input>
<button id="search">Search</button>
<p id="results"></p>
```

Open `index.html` in Chrome and make sure you see an input field and a text button.

Now we want to write some code so that we can interact with the DOM. We want our code to be resilient to browser differences so we'll use Google Closure to abstract this stuff away as we might with jQuery.

We require `goog.dom` and give it a less annoying alias. Change the `ns` form in `src/async_tut1/core.cljs` to the following:

```
(ns async-tut1.core
  (:require [goog.dom :as dom]))
```

We want to confirm that this will work, so let's change the `console.log` expression so it looks this instead:

```
(.log js/console (dom/getElement "query"))
```

Save the file and it should be recompiled instantly. We should be able to refresh the browser and see that a DOM element got printed in the JavaScript Console (View > Developer > JavaScript Console). Remove this little test snippet after you've confirmed it works.

So far so good.

Now we want a way to deal with the user clicking the mouse. Instead of just setting up a callback on the button directly, we're going to make the button put the click event onto a `core.async` channel.

Let's write a little helper called listen that will return a channel of the events for a particular element and particular event type. We need to require `core.async` macros and functions. Our `ns` should now look like the following:

```
(ns async-tut1.core
  (:require-macros [cljs.core.async.macros
:refer [go]])
  (:require [goog.dom :as dom]
            [goog.events :as events]
            [cljs.core.async :refer [put! chan
<!]]))
```

Again we want to abstract away browser quirks so we use goog.events for dealing with that. We include only the `core.async` macros and functions that we intend to use.

Now we can write our `listen` fn; it looks like this:

```
(defn listen [el type]
  (let [out (chan)]
    (events/listen el type
      (fn [e] (put! out e)))
    out))
```

We want to verify our function works as advertised, so we check it with following snippet of code at the end of the file:

```
(let [clicks (listen (dom/getElement "search")
"click")]
  (go (while true
        (.log js/console (<! clicks)))))
```

Note that we've created what appears to be an infinite loop here, but actually it's a little state machine. If there are no events to read from the click channel, the go block will be suspended.

Let's search Wikipedia. Define the basic URL we are going to hit via JSONP and put this right after the `ns` form.

```
(def wiki-search-url
  "http://en.wikipedia.org/w/api.php?action=open
search&format=json&search=")
```

Now we want to make a function that returns a channel for JSONP results.

We again reach for Google Closure to avoid browser quirks. Make your `ns` form look like the following:

```
(ns async-tut1.core
  (:require-macros [cljs.core.async.macros
:refer [go]])
  (:require [goog.dom :as dom]
            [goog.events :as events]
            [cljs.core.async :refer [<! put!
chan]])
  (:import [goog.net Jsonp]
           [goog Uri]))
```

Here we use `:import` so that we can use short names for the Google Closure constructors.

Note: :import is only for this use case; you never use it with ClojureScript libraries.

Our JSONP helper looks like the following (put it after listen in the file):

```
(defn jsonp [uri]
  (let [out (chan)
        req (Jsonp. (Uri. uri))]
    (.send req nil (fn [res] (put! out res)))
    out))
```

This looks pretty straight forward, very similar to listen. Let's write a simple function for constructing a query url:

```
(defn query-url [q]
  (str wiki-search-url q))
```

Again let's test this by writing a snippet of code at the bottom of the file.

```
(go (.log js/console (<! (jsonp (query-url
"cats")))))
```

In the JavaScript Console we should see we got an array of JSON data back from Wikipedia. Success!

It's time to hook everything together. Remove the test snippet and replace it with the following:

```
(defn user-query []
  (.-value (dom/getElement "query")))

(defn init []
  (let [clicks (listen (dom/getElement "search")
"click")]
    (go (while true
          (<! clicks)
          (.log js/console (<! (jsonp (query-url
(user-query)))))))))

(init)
```

Try it now. You should be able to write a query in the input field, click "Search", and see results in the JavaScript Console.

If you've done any JavaScript programming, this way of writing the code should be somewhat surprising — we don't need a callback to work with button clicks!

Think about how this works: when the page loads, `init` will run, the `go` block will try to read from `clicks`, but there will be nothing to read, so the `go` block becomes suspended. Only when you click on the button can it proceed, at which point we'll run the query and loop around. The code reads exactly how it would if you didn't have to consider asynchrony!

Instead of printing to the console we would like to render the results to the page. Let's do that now, add the following before `init`:

```
(defn render-query [results]
  (str
    "<ul>"
    (apply str
      (for [result results]
        (str "<li>" result "</li>")))
    "</ul>"))
```

The usual string concatenation stuff. We use a list comprehension here just for fun.

Now change `init` to look like the following:

```
(defn init []
  (let [clicks (listen (dom/getElement "search")
"click")
        results-view (dom/getElement "results")]
    (go (while true
          (<! clicks)
          (let [[_ results] (<! (jsonp (query-
url (user-query))))]
            (set! (.-innerHTML results-view)
(render-query results)))))))
```

Hopefully this code at this point just makes sense. Notice how we can use destructuring on the JSON array of Wikipedia results.

A beautiful succinct program! The complete listing follows:

```
(ns async-tut1.core
  (:require-macros [cljs.core.async.macros :refer [go]])
  (:require [goog.dom :as dom]
            [goog.events :as events]
            [cljs.core.async :refer [<! put! chan]])
  (:import [goog.net Jsonp]
           [goog Uri]))

(def wiki-search-url
  "http://en.wikipedia.org/w/api.php?action=opensearch&format=json&search=")

(defn listen [el type]
  (let [out (chan)]
    (events/listen el type
      (fn [e] (put! out e)))
    out))

(defn jsonp [uri]
  (let [out (chan)
        req (Jsonp. (Uri. uri))]
    (.send req nil (fn [res] (put! out res)))
    out))

(defn query-url [q]
  (str wiki-search-url q))

(defn user-query []
  (.-value (dom/getElement "query")))

(defn render-query [results]
  (str
    "<ul>"
    (apply str
      (for [result results]
        (str "<li>" result "</li>")))
    "</ul>"))

(defn init []
  (let [clicks (listen (dom/getElement "search") "click")
        results-view (dom/getElement "results")]
    (go (while true
          (<! clicks)
          (let [[_ results] (<! (jsonp (query-url (user-query))))]
            (set! (.-innerHTML results-view) (render-query results)))))))

(init)
```

David Nolen is a JavaScript developer for The New York Times. In his free time he works on a variety of open source Clojure projects including core.match, core.logic, and ClojureScript.

Basics of Function Pointers in C

By DENNIS KUBES

Function pointers are an interesting and powerful tool but their syntax can be a little confusing. This post will going into C function pointers from the basics to simple usage to some quirks about function names and addresses. In the end it will give you an easy way to think about function pointers so their usage is clearer.

A Simple Function and Function Pointer

Let's start with a very simple function to print out the message "hello world" and see how we can create a function pointer from there.

```
#include <stdio.h>

// function prototype
void sayHello();

// function implementation
void sayHello() {
  printf("hello world\n");
}

// calling from main
int main() {
  sayHello();
}
```

Here we have a function called sayHello along with its function prototype. This function returns nothing (void) and doesn't take any parameters. We call the function from main and it prints out "hello world". Pretty simple. Now let's convert main to use a function pointer instead of calling the function directly.

```
int main() {
  void (*sayHelloPtr)() = sayHello;
  (*sayHelloPtr)();
}
```

The syntax `void (*sayHelloPtr)()` on line 2 may look a little weird so let's look at it step by step.

1. We are creating a function pointer to a function that returns nothing (void) so the return type is void. That is the void keyword.
2. We have the pointer name sayHelloPtr. This is similar to creating any other pointer and it has to have a name.
3. We use the * notation to signify that it is a pointer. This is no different than declaring an int pointer or a char pointer.
4. We must have parentheses around the pointer `(*sayHelloPrt)`. If we don't have parentheses it is seen as `void *sayHelloPtr`, which is a void pointer instead of a pointer to a void function. This is a key point; function pointers must have parentheses around them.
5. We have the parameter list. Since there isn't one in this case, we just have empty parentheses `(*sayHelloPrt)()`.
6. Putting it all together we get `void (*sayHelloPtr)()`, a pointer to a function that returns void and takes no parameters.

On line 2 above we are assigning the sayHello function name to our newly created function pointer like this: `void (*sayHelloPtr)() = sayHello`. We will go into more detail about function names later, but for now understand that a function name (label) is the address of the function and it can be assigned to a function pointer. This is similar to `int *x = &myint` where we assign the address of myint to an int pointer. Only in the case of a function, the address-of the function is the function name and we don't need the address-of operator. Simply put, the function name is the address-of the function. On line 3 we dereference and call our function pointer like this `(*sayHelloPtr)()`.

1. Once created on line 2, sayHelloPtr is our function pointer name and can be treated just like any other pointer, assigned, and stored.
2. We dereference our sayHelloPtr pointer the same as we dereference any other pointer, by using the value-at-address (*) operator. This gives us `*sayHelloPtr`.
3. Again we must have parentheses around the pointer `(*sayHelloPrt)`. If we don't, it isn't a function pointer. We must have parentheses when creating a function pointer and when dereferencing it.
4. The `()` operator is used to call a function in C. It is no different on a function pointer. If we had a parameter list there would be values in the parentheses similar to any other function call. This gives us `(*sayHelloPrt)()`.
5. This function has no return value so there is no need to assign its return to any variable. The function call can standalone similar to `sayHello()`.

Now that we have shown the weird syntax, understand that often function pointers are just treated and called as regular functions after being assigned. To modify our previous example:

```
int main() {
  void (*sayHelloPtr)() = sayHello;
  sayHelloPtr();
}
```

As before we assign the sayHello function to our function pointer, but now we call the function pointer just like we would call a regular function. We will get into function names later which will show why this works but for now understand that calling a function pointer with full syntax `(*sayHelloPtr)()` is the same as calling the function pointer as a regular function `sayHelloPtr()`.

A Function Pointer with Parameters

Now lets create a function pointer that still doesn't return anything (void) but now has parameters.

```
1   #include <stdio.h>
2
3   // function prototype
4   void subtractAndPrint(int x, int y);
5
6   // function implementation
7   void subtractAndPrint(int x, int y) {
8     int z = x - y;
9     printf("Simon says, the answer is: %d\n", z);
10  }
11
12  // calling from main
13  int main() {
14    void (*sapPtr)(int, int) = subtractAndPrint;
15    (*sapPtr)(10, 2);
16    sapPtr(10, 2);
17  }
```

As before, we have our function prototype, our function implementation and the executing of the function from main using a function pointer. The signature of both the prototype and its implementation has changed. Where before our sayHello function didn't have parameters, the subtractAndPrint function takes two parameters, both integers, and subtracts one from the other and prints the result.

1. We create our sapPtr function pointer on line 14 with `void (*sapPtr)(int, int)`. The only difference from before is that instead of empty parentheses on the end when creating the function we have `(int, int)`, which matches the signature of our new function.
2. On line 15 when dereferencing and executing the function, everything is the same as when we called our sayHello function except now we have `(10, 2)` on the end passing parameters.
3. On line 16 we show executing the function pointer as a regular function.

A Function Pointer with Parameters and Return Value

Let's change our subtractAndPrint function to be called subtract and to return the result instead of printing it.

```
1   #include <stdio.h>
2
3   // function prototype
4   int subtract(int x, int y);
5
6   // function implementation
7   int subtract(int x, int y) {
8     return x - y;
9   }
10
11  // calling from main
12  int main() {
13    int (*subtractPtr)(int, int) = subtract;
14
15    int y = (*subtractPtr)(10, 2);
16    printf("Subtract gives: %d\n", y);
17
18    int z = subtractPtr(10, 2);
19    printf("Subtract gives: %d\n", z);
20  }
```

This is similar to the subtractAndPrint function, except now the subtract function returns an int. The prototype and function signatures have changed as would be expected.

1. We create our subtractPtr function pointer on line 13 with `int (*subtractPtr)(int, int)`. The only difference from before is instead of void we have an int return value. This matches our subtract method signature.
2. On line 15 when dereferencing and executing the function pointer, everything is the same as when we called our subtractAndPrint function, except now we have `int y =`, which assigns the return value of the function to y.
3. On line 16 we print out the return value.
4. On lines 18 – 19 we execute the function pointer as a regular function and print the results.

This isn't much different from before; we just added the int return value. Let's move on to a little more complex example where we pass a function pointer into another function as a parameter.

Passing a Function Pointer as a Parameter

We have stepped through the main parts of the declaring and executing function pointers with and without parameters and return values. Now let's look at using a function pointer to execute different functions based on input.

```
1   #include <stdio.h>
2
3   // function prototypes
4   int add(int x, int y);
5   int subtract(int x, int y);
6   int domath(int (*mathop)(int, int), int x,
    int y);
7
8   // add x + y
9   int add(int x, int y) {
10    return x + y;
11  }
12
13  // subtract x - y
14  int subtract(int x, int y) {
15    return x - y;
16  }
17
18  // run the function pointer with inputs
19  int domath(int (*mathop)(int, int), int x,
    int y) {
20    return (*mathop)(x, y);
21  }
22
23  // calling from main
24  int main() {
25
26    // call math function with add
27    int a = domath(add, 10, 2);
28    printf("Add gives: %d\n", a);
29
30    // call math function with subtract
31    int b = domath(subtract, 10, 2);
32    printf("Subtract gives: %d\n", b);
33  }
```

Let's break this down:

1. We have two functions with the same signature `int function(int, int)`, add and subtract. Both return an integer and both take two integers as parameters.
2. On line 6 we have `int domath(int (*mathop)(int, int), int x, int y)`. The first parameter `int (*mathop)(int, int)` is a pointer to a function that takes two integers as input and returns an integer. We have seen this before, and the syntax is no different here. The last two parameters x and y are just integer inputs into the domath function. So the domath function takes a function pointer and two integers as parameters.
3. On lines 19 – 21 the domath function executes the function pointer passed with the x and y integers passed. This could also have been done as `mathop(x, y);`.
4. Lines 27 and 31 are somewhat new. We are calling the domath function and we are passing in the function names. Function names are the address-of the function and can be used in place of function pointers.

The main function calls domath twice, once for add and once for subtract, printing out the results.

Function Names and Addresses

Let's wrap up by talking a bit about function names and addresses as promised. A function name (label) is converted into a pointer to itself. This means that function names can be used where function pointers are required as input. It also leads to some very funky looking code that actually works. Take a look at some examples:

```
1   #include <stdio.h>
2
3   // function prototypes
4   void add(char *name, int x, int y);
5
6   // add x + y
7   void add(char *name, int x, int y) {
8     printf("%s gives: %d\n", name, x + y);
9   }
10
11  // calling from main
12  int main() {
13
14    // some funky function pointer assignment
15    void (*add1Ptr)(char*, int, int) = add;
16    void (*add2Ptr)(char*, int, int) = *add;
17    void (*add3Ptr)(char*, int, int) = &add;
18    void (*add4Ptr)(char*, int, int) = **add;
19    void (*add5Ptr)(char*, int, int) = ***add;
20
21    // execution still works
22    (*add1Ptr)("add1Ptr", 10, 2);
23    (*add2Ptr)("add2Ptr", 10, 2);
24    (*add3Ptr)("add3Ptr", 10, 2);
25    (*add4Ptr)("add4Ptr", 10, 2);
26    (*add5Ptr)("add5Ptr", 10, 2);
27
28    // this works too
29    add1Ptr("add1PtrFunc", 10, 2);
30    add2Ptr("add2PtrFunc", 10, 2);
31    add3Ptr("add3PtrFunc", 10, 2);
32    add4Ptr("add4PtrFunc", 10, 2);
33    add5Ptr("add5PtrFunc", 10, 2);
34  }
```

Run this code and every function pointer will execute. Yes, you will get some warnings about char conversion as this is a simple example. But the function pointers still work.

1. Line 15: the function name add by itself gives the address of the function. It is implicitly converted to a function pointer. Function names can be used where function pointers are required as input.
2. Line 16: the value-at-address operator `*add` when applied to the function name gives the function at that address, which is converted to a function pointer implicitly just like the function name.
3. Line 17: address-of (&) operators when applied to a function name gives the address of the function. This yields a function pointer, too.
4. Lines 18 and 19: the pointers to the function keep yielding themselves over and over again, returning the function address, which is converted to a function pointer. In the end, it is same as just the function name.
5. This code isn't an example of best practice. The takeaway is this: One, function names are converted to function pointers implicitly the same way that array names are converted to pointers implicitly when passed into functions. Function names can be used wherever a function pointer is required. Two, the address-of (&) and value-at-address (*) operators are almost always redundant when used against function names.

Conclusion

I hope this helps clarify some things about function pointers and their usage. When understood, function pointers become a powerful tool in the C toolbox. In future posts I may go into more detailed usage of function pointers for things like callbacks and basic OOP in C. ■

Dennis lives in Plano, Texas and has been programming for over 15 years. He uses many different languages, whatever works best for the job. He thinks programming is an art, algorithms can be elegant and mathematics can be beautiful.

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

HTML5 Live Video Streaming Via WebSockets

By DOMINIC SZABLEWSKI

When I built my Instant Webcam App, I was searching for a solution to stream live video from the iPhone's Camera to browsers. There were none.

When it comes to (live) streaming video with HTML5, the situation is pretty dire. HTML5 Video currently has no formalized support for streaming whatsoever. Safari supports the awkward HTTP Live Streaming, and there's an upcoming Media Source Extension standard as well as MPEG-DASH. But all these solutions divide the video in shorter segments, each of which can be downloaded by the browser individually. This introduces a minimum lag of 5 seconds.

So here's a totally different solution that works in any modern browser: Firefox, Chrome, Safari, Mobile Safari, Chrome for Android, and even Internet Explorer 10.

It's quite backwards, uses outdated technology, and doesn't support audio at the moment. But it works. Surprisingly well.

The Camera Video is encoded by ffmpeg sent to a tiny nodejs script over HTTP that simply distributes the MPEG stream via WebSockets to all connected Browsers. The Browser then decodes the MPEG stream in JavaScript and renders the decoded pictures into a Canvas Element.

You can even use a Raspberry Pi to stream the video. It's a bit on the slow side, but in my tests it had no problem encoding 320x240 video on the fly with 30fps. This makes it, to my knowledge, the best video streaming solution for the Raspberry Pi right now.

Here's how to set this up. First get a current version of ffmpeg. Up-to-date packages are available at deb-multimedia. If you are on Linux, your Webcam should be available at `/dev/video0` or `/dev/video1`. On OSX or Windows you may be able to feed ffmpeg through VLC somehow.

Make sure you have nodejs installed on the server through which you want to distribute the stream. Get the `stream-server.js` script from jsmpeg and change the default password at the top of the file. This password is there to ensure that no one can hijack the video stream.

Now install its dependency to the ws WebSocket package and start the server:

```
npm install ws
node stream-server.js yourpassword
```

You should see the following output when the server is running correctly:

```
Listening for MPEG Stream on http://127.0.0.1:80
82/<secret>/<width>/<height>
Awaiting WebSocket connection on ws://127.0.0.1:8084
```

With the server started, you can now start ffmpeg and point it to the domain and port where it is running:

```
ffmpeg -s 640x480 -f video4linux2 -i /dev/video0
\ -f mpeg1video -b 800k -r 30
\ http://example.com:8082/yourpassword/640/480/
```

This starts capturing the webcam video in 640x480 and encodes an MPEG video with 30fps and a bitrate of 800kbit/s. The encoded video is then sent to the specified host and port via HTTP. Make sure to provide the correct secret as specified in the `stream-server.js`. The width and height parameters in the destination URL also have to be set correctly; the stream server otherwise has no way to figure out the correct dimensions.

On the Raspberry Pi you will probably have to turn down the resolution to 320x240 to still be able to encode with 30fps.

To view the stream, get the `stream-example.html` and `jsmpg.js` from the jsmpeg project [hn.my/jsmpeg]. Change the WebSocket URL in the `stream-example.html` to your server's and open it in your favorite browser.

If everything works, you should be able to see a smooth camera video with less than 100ms lag. Quite nice for such hackery and a humble MPEG decoder in JS. ■

Dominic is the author of the HTML5 Game Engine Impact [impactjs.com]. He writes about all things JavaScript and Web-Technology on his personal blog PhobosLab [phoboslab.org] and create games in his free time.

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

Behind the Scenes with CHICKEN Scheme

By JOHN CROISANT & FELIX WINKELMANN

For the past couple years, I've been playing with the Lisp family of languages, namely Common Lisp, Clojure, and Scheme. One of my favorite languages for hobby coding is CHICKEN Scheme [call-cc.org], a mature, high-performance implementation of Scheme that compiles to portable C code. CHICKEN's variety of built-in features and downloadable libraries, excellent FFI support, and helpful community make it a very appealing language.

Recently, I came across SPOCK, a compiler and runtime system for compiling Scheme code into JavaScript. As it turns out, SPOCK and CHICKEN have the same creator: Felix Winkelmann, a software developer in Göttingen, Germany. Intrigued, I got in touch with Felix to ask him about CHICKEN, SPOCK, how he got started, and what keeps him motivated to keep working on CHICKEN after more than a decade.

Felix, thanks for agreeing to an interview. Many of our readers probably haven't heard of CHICKEN Scheme before. What is it? What kinds of software is it good for? What sets it apart from other Scheme implementations?

CHICKEN is, at its core, just another implementation of the Scheme programming language. It is R5RS-compliant and provides numerous extension libraries for all sorts of things. CHICKEN compiles Scheme into portable C code, which can subsequently be compiled into a standalone executable or a library. An interpreter is also available for interactive development.

There are a large number of extensions (what we call "eggs") that cover a large spectrum of functionality, like bindings to C and C++ libraries and handling of many databases, protocols, networking, graphics, and user interface programming. So I'd say it is good for a lot of things. Dynamic typing combined with a compiler that can generate quite efficient code allows CHICKEN to be used for everything including scripting, application programming, and systems programming.

What I like about CHICKEN is that it makes it very easy to work with existing libraries. The foreign function interface makes integrating C code a snap. Many people have contributed extensions to our library, and installing these extensions is straightforward. CHICKEN tries to be developer-friendly and easy to use, and it puts an emphasis on making those things simple that

have traditionally been neglected in dynamic languages, like generating real standalone executables. Full support for Scheme is provided, including the parts that are usually hard to implement or implemented inefficiently.

But what really makes CHICKEN special is its community. A group of helpful and faithful fanatics is actively maintaining and improving it, sometimes at a frightening pace. If you need help, ask on the mailing lists or IRC channel and you get it. Always.

What motivated you to create CHICKEN?

I was scratching my own itch: having a decent compiler for a powerful and elegant language, one that I can use for day-to-day programming instead of banging my head against the limitations of the mainstream languages that I have to use otherwise. Something that doesn't get in the way of solving a particular programming problem.

Do you often use CHICKEN in your own programming? What kinds of software to you create with it?

I use CHICKEN as much as I can. I have done some freelancing writing Scheme, but haven't had the chance so far to use it at work (I'd love to, though). Unfortunately, I don't have much time left between work and maintenance, even though my head is exploding with ideas. If I find the time, I usually implement other programming languages — I'm one of those programmers that always end up implementing programming languages in the hope of using them to write something interesting in the future… But I never get beyond the first stage. :-)

When did you first become interested in computers or programming? How did you learn to program?

I started around the age of 12, I think, at the start of the home computing era. I never got the computers with the cool games like the other kids, so I had to dive into BASIC programming pretty early. Later I studied mathematics and computer science but quickly realized that I'm way too dumb for math and dropped out after just a year or so.

I'm addicted to computer books, so I was able to pick up a lot of different subjects, but I always ended up learning about programming languages.

How did you first learn about Scheme/Lisp? Did you find it challenging at first, or did it come easily? What made you like it enough to create a Scheme implementation?

I somehow came upon a small book about Lisp, which was very challenging and strange. But my fascination started early and I sucked up everything I could about Lisp, its various variants and the implementation techniques involved in making it run. Scheme, being such a clean, minimal and elegant language got me quickly hooked, as it did to so many others.

Internet access came very late, so to get access to a Lisp system I had to write one myself. I wrote countless Lisp and Scheme implementations — most of them were rubbish, and none was ever complete. But implementing Lisp is the true way of learning the language, and in the end, reading Henry Baker's "Cheney on the M.T.A." paper and Andrew Appel's wonderful book "Compiling with Continuations" showed a way that was just so elegant that I had to try it out.

You first released CHICKEN in 2000 — over a decade ago! What motivates you to keep working on it after so many years? Have you ever had times of low motivation, where you didn't want to work on it anymore? How did you cope?

Yes, I think hacking began about 15 years ago. It's hard to believe that it has been such a long time. I wanted to stop more than once, but what made the difference was the feedback I got. Even when the system was barely usable (actually even when it wasn't usable at all), people tried it out, sent patches, suggested improvements and, most surprisingly, they used it! For real stuff! That was both baffling and highly motivating. Being so grateful for the feedback, I couldn't stop working on it.

Maintaining such a project, especially one that is growing very fast, can be quite a piece of work. Over the years, a core team of very capable, motivated and friendly folks has emerged that do all the hard work and additionally keep up with my moods. But before that, keeping up with the project (bug-fixing, porting, testing) turned out to be a full-time job. I was ready to walk away more than once, and not having the time to use the stuff you worked on for such a long time can be quite frustrating. Usually, taking a few weeks of vacation from all things CHICKEN related helps, until my fingers start itching, the ideas start flowing, and I throw myself back into the project.

In your initial announcement of CHICKEN, you included a disclaimer: "This is *not* a production quality/high-performance system." A lot has changed since then. Would you say now that CHICKEN is a "production quality/high performance system"?
Yes, I think I'd say that. The compiler can generate very fast code, if you know what you're doing and if you have an idea of how it operates. A massive amount of code has been fed to the system, which weeded out a countless number of bugs. So it is not too immodest to say that CHICKEN has become quite mature.

It will never be bug free, of course, but that is the price you pay for keeping up a fast pace of development. With maturity, the class of bugs shifts to more advanced and obscure parts of the system. Additionally, we do an awful lot of automated testing, which is of tremendous help.

Do you have future plans for CHICKEN? Where would you like it to be in 5 years?
There are many things that need to be improved. People are using it heavily, and companies have started using it for getting real stuff done, so there is always something to fix and improve. The next Scheme standard (R7RS) is around the corner, and we plan to support it, which will be another piece of work. A lot of infrastructure has been created (testing, bug-tracking, code repositories, documentation, etc.) that needs constant attention.

I don't know. I think in 5 years I would like it to be like it is now — just better.

Let's talk about SPOCK. What is it? Is it ready for people to use? Why would someone want to use it?
SPOCK is a compiler from a subset of R5RS Scheme to JavaScript. It uses a compilation strategy similar to CHICKEN, but it is more lightweight and cuts a few corners of the Scheme standard to be practical. It has not been used a lot so far, but it works, and I think it has some potential to be a useful glue language for Scheme-based web software. But, I'm not an expert in web programming, so my opinions must be taken with a grain of salt.

The interesting bit is that the distinction between server-side and client-side gets fuzzier — a Scheme server can emit Scheme code to run on the client, and Scheme's powerful syntactic extension mechanisms can make this look like a single piece of code. SPOCK is not what I'd call ready for production yet. But I'd say there is potential.

What motivated you to create SPOCK?
Originally, I wanted to have a clean compiler core for Scheme, using the "Cheney on the M.T.A." compilation strategy (which is also used in CHICKEN). JavaScript is an interesting and powerful target language that already takes care of a lot of things (garbage collection, dynamic typing, etc.), so it was a natural choice. After the usual frustration of getting it to work on all major browsers, the parts just fell into place.

Have you used it for any projects yet?
I have only done experiments with it. I'd love to do more, but I severely lack the experience in web programming.

You mentioned that CHICKEN and SPOCK both use Henry Baker's "Cheney on the M.T.A." compilation strategy. How much did your experience developing CHICKEN help with creating SPOCK? Are the implementations similar?
Baker's method is really incredibly clever — naturally, I have to say that — but the code that it produces takes some getting used to. It's a bit of a challenge to read code that has been converted to continuation-passing style (CPS) and translated to another language. Without the experience I gained from CHICKEN, SPOCK would have taken much more time. SPOCK is a good deal simpler and cleaner than CHICKEN, but of course it supports a much smaller language, it's not fully R5RS compliant, and it doesn't have to cope with the horrors of POSIX, the Windows API, or C compiler issues. And JavaScript takes care of a lot of dirty details, of course.

Much of Baker's paper seems pretty specific to the memory management and function call conventions of C. What gave you the idea of applying it to JavaScript? Are the techniques described in the paper relevant in JavaScript?
I think they are relevant to every language. Baker's compilation strategy is applicable to nearly every language that has activation frames with limited extent. It elegantly combines garbage collection with stack frame management and continuation creation, so static languages like C are a natural choice. JavaScript already provides garbage collection, but Baker's method gives us tail-call optimization and first-class continuations. There has been at least one CPS-based Scheme-to-JavaScript compiler before, but it didn't explicitly use Baker's method, as far as I know.

SPOCK's documentation includes a warning that it "stresses JavaScript implementations in unusual ways." Are there significant performance issues with SPOCK? If so, do you think performance will improve as SPOCK matures?
That is possible, yes. SPOCK creates deeply nested functions, and this stresses existing JavaScript engines in unexpected ways. It even uncovered a bug in Mozilla's JavaScript engine — which is fixed now, thanks to the engine's maintainers. There may be corner cases that haven't been thought of yet. SPOCK just needs more users and more testing.

What's next for SPOCK? Are you going to continue developing it?
Currently, I'm just waiting for people to use it.

But, SPOCK is clean enough to be grokked by whoever wants to hack on it. It is not under active development at the moment, but it has a reasonable size and complexity, which makes it easier to maintain than, say, CHICKEN. It would be interesting to see how people use it, and I'll be available in case something breaks.

Both CHICKEN and SPOCK are open source. If someone is interested in contributing, what is the best way to get started?

Just give it a try. Play with it, learn about it, write something useful, or even something useless. Then get in touch with the community, ask questions on the mailing list, or enjoy the daily fun on our IRC channel. Submitting a new CHICKEN library or extension module is very easy. There are endless things to do, even if it is just testing, and we are happy about every little bit of help we can get, and happy to provide help to those that need it themselves. Every line of code contributed makes CHICKEN better, increases our corpus of testing code, or at least gives us something to think about.

One last question: What inspired the names CHICKEN and SPOCK? Do they mean anything, aside from the bird and the well-known Star Trek character?
That question always comes up, sooner or later.

I had a plastic toy of Feathers McGraw on my desk, the evil penguin (disguised as a chicken!) from the Wallace and Gromit movie, "The Wrong Trousers." Looking for a preliminary working title for the compiler, I used the first thing that came to my mind that day. I'm somewhat superstitious about names for software projects, and things were progressing well, so I didn't dare to change the name.

Also, there is the old philosophical question: which came first, the chicken or the egg? This applies to CHICKEN, too. The compiler is written in Scheme, so you need CHICKEN in order to compile CHICKEN.

For SPOCK, the story is not that interesting. I just like whacky names, and it seemed nice to have some sort of "persona" to associate with the compiler. Like CHICKEN, "Spock" was just the first thing that came to mind.

After SPOCK, I worked for a while on a rudimentary compiler that produced C++ instead of JavaScript, but it was never finished. It was quite bare "bones," so naturally I called it MCCOY. ■

John Croisant is a self-taught programmer in a variety of languages, including Python, C/C++, Lisp/Scheme, and especially Ruby. In his downtime, he enjoys reading fiction, watching old movies and TV series, and (of course) playing video games.

Felix Winkelmann is the implementor and lead-maintainer of CHICKEN, a popular Scheme implementation.

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

Client-side Full-text Search in CSS

By FRANÇOIS ZANINOTTO

USING DATA- ATTRIBUTES for indexation and a dynamic stylesheet with a CSS3 selector for search is is straightforward way to implement a client-side full-text search in CSS rather than JavaScript. Here is an example.

The Searchable List

```
<!-- Data generated by Faker, see https://
github.com/fzaninotto/Faker -->
<ul class="contacts">
  <!-- Add text to the data-index attribute to
enable full-text search -->
  <!-- Don't forget to lowercase it to make
search case-insensitive -->
  <li class="searchable" data-
index="onabednarschamberger.frank@wuckert.com1-
265-479-1196x714">
    <dl>
      <dt>First Name</dt><dd>Ona</dd>
      <dt>Last Name</dt><dd>Bednar</dd>
      <dt>Email</dt><dd>schamberger.frank@wuck-
ert.com</dd>
      <dt>Phone</dt><dd>1-265-479-1196x714</dd>
    </dl>
  </li>
  <li class="searchable" data-
index="newtoncronintorphy.dorothea@gmail.
com(121)644-5577">
    <dl>
      <dt>First Name</dt><dd>Newton</dd>
      <dt>Last Name</dt><dd>Cronin</dd>
      <dt>Email</dt><dd>torphy.dorothea@gmail.
com</dd>
      <dt>Phone</dt><dd>(121)644-5577</dd>
    </dl>
  </li>
  <!-- add as much data as you want -->
</ul>
```

The Search Code

The search is very straightforward: it relies on two well-supported CSS3 selectors (`:not()` and `[attr*=]`) and the rewriting of a style rule each time the search input is modified:

```
<input type="text" placeholder="search"
id="search">
<style id="search_style"></style>
<script type="text/javascript">
var searchStyle = document.
getElementById('search_style');
document.getElementById('search').
addEventListener('input', function() {
  if (!this.value) {
    searchStyle.innerHTML = "";
    return;
  }
  // look ma, no indexOf!
  searchStyle.innerHTML =
".searchable:not([data-index*=\"" + this.value.
toLowerCase() + "\"]) { display: none; }";
  // beware of css injections!
});
</script>
```

The advantage of using CSS selectors rather than JavaScript `indexOf()` for search is speed: you only change one element at each keystroke (the `<style>` tag) instead of changing all the elements matching the query. Using the `:not()` selector, this implementation works on IE9+, but it could easily be made compatible with IE8+ by using two rules instead of one. ■

François Zaninotto is the CEO a digital innovation workshop named marmelab, located in eastern France. Former Propel lead developer, former Symfony lead documenter, he is still involved in various open-source projects in PHP and Node.js. He regularly blogs about open-source, Lean Startup, Domain-Drive Design and tech trends in *redotheweb.com*

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

Hack Your Motivation

By BEMMU SEPPONEN

Nothing is better than being truly motivated by an exciting project. But if you're stuck, here are some things to try for a temporary boost. The common theme among these is switching your perspective from thinking about your project as a huge endeavor and instead concentrating on the next practical step.

"Just one change"

You should really be working on your project, but it just seems too daunting today to get into it. Open one file in your project and try to improve just one line. Just make one tiny change. That change often leads to another and can get you going.

Time challenge

This can turn a mundane task into an interesting challenge. Should you need to gather some receipts or other documents to submit to your bookkeeper each month, turn the boring task into a challenge by keeping a high score list of how long it takes you each time. Last month you did it in four hours. Can you do it in less time? Try to beat your record.

Time slotting

Sometimes you are not in the mood for speed challenges and even a bit of progress today would be a victory. Maybe in reality you have the whole evening to work, but pretend it is not so. Try allocating just an hour. If you could choose, what would be the best thing to work on today between 10 - 11am. If you could clone yourself for an hour and make the clone do that task, what would you have it do? When the hour comes, you might actually find yourself doing the task you allocated, because after all you yourself decided that to be the most important thing you could be working on at that point in time.

Make a list of goals

Make a list of current goals or revisit an existing one. That, and the realization that your time on this planet is limited, might scare you into action.

Help one person

If you have received some feedback related to your project, go read some. Could you help this person, or better yet improve your project in some small way to make it less likely for the same trouble to happen in the future?

Structured procrastination

If nothing helps you get started in your current task, is there another task which seems more appealing? Thinking about all the things you need to do, can you find the motivation to do one of them? If none of these help to get you started, maybe your mind or body is trying to tell you something. It could be time to take a break.

How to maintain your motivation

You managed to get started, now how to keep going?

Seinfeld method

Jerry Seinfeld once described his method for making better jokes: work on it every day. His system is to have a wall calendar and mark an X on it for every day that he put effort into writing his jokes. After getting a chain of X marks in the calendar, you are motivated by not wanting to break the chain. GitHub also has this feature, every day you contribute to a repo, they mark that day in green.

> **“Having even one person paying for your stuff will greatly increase how motivated you get in trying to improve it.”**

Solicit feedback

If you already have some audience, try to get them to interact with you. If you start getting emails or tweets about your task, it becomes natural to put more effort into working on it. For example if you have a blog, at the end you could invite users to vote on new topics for you to write about. If you have a web app, you could add a live chat or feedback widget or prominently mention your email address to make it easy for people to reach out to you. If you receive a problem report this way, it feels wrong NOT to get to work immediately.

Install RescueTime

This is an app you can install on your computer that monitors which apps you are using. You can mark activities as productive or not productive. You can tell RescueTime that being in a text editor is productive, but being on Facebook is not. Based on this it knows how many productive hours you had and can send you a congratulation email when you reach your daily productivity goal and make you have an extra feel-good association with staying productive.

Make a dollar

If you have a side project that you are currently doing for free, try asking for payment. Not because you are greedy, but because getting paid is a strong signal from others that they find value in what you are doing and want you to work on that thing. You might find that having even one person paying for your stuff will greatly increase how motivated you get in trying to improve it. If you feel like “I can’t do that, I could let them down,” well, that’s exactly the point: you’ll get a boost of motivation from it. And if you really do feel that you let them down, there are always refunds.

Write a ridiculously detailed battle plan for tomorrow

Before going to bed, think about what the perfect day would look like. Maybe you would get up, get your inbox to zero, write some code, do some copywriting or have a nice session of exercise or study. If you can picture the perfect day, you could try writing it down in detail, down to the hour (remember to leave plenty of room for rest and breaks, too). Now tomorrow it will be clear what constitutes a success for that day.

Leave a small task undone

To jump start your productivity the next day, leave a task open from today. Before calling it a night, leave just one line of code unfinished so you can jump in and finish that as the first natural task for getting into a productive mood tomorrow. ■

Bemmu Sepponen is an expat developer. He also runs Candy Japan [candyjapan.com], a Japanese sweets subscription service.

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

10 Inglorious Years of Bootstrapping

By DAVE GOODEN

In 2002, after several years of running a small but successful e-commerce business, my business partner (and friend since kindergarten) Cameron Henkel and I were both searching for vacation homes to purchase as family getaways. After a lot of hassle (I'll spare you the details), we realized that there could be a big opportunity in the space. In 2002 the real estate industry was way, way behind the curve when it came to applying technology to the process, so we set out to fix the problem.

In 2003 we launched LakePlace.com, a niche classified ads website for lake homes and lake lots in Minnesota and Wisconsin. Like all marketplaces, we faced the chicken and the egg problem. We had no listings and no visitors. Common sense told us that we needed to build the supply side first and worry about the demand side later…so that's what we set out to do.

We spent the next 12 months on the phones, sending emails, attending conferences and trade shows, and meeting with real estate agents in person to convince them to list their properties on LakePlace.com. We offered everyone a free trial and 100% satisfaction guarantee (people like 100% satisfaction guarantees). Our e-commerce success of the past was built on SEO, before it was called SEO, so we knew if we could get listings, we could get visitors… and that's exactly what happened. We started getting some listings and then started getting some traffic, more listings, more traffic. Once we noticed specific agents getting multiple contacts on properties, we swooped in with the sales call. It was time to upgrade to a paid account or lose the service. By 2006 we had 600+ paying customers listing thousands of lakeshore properties on our site.

Along the way, something else happened. We noticed our visitors asking our listing agents if any of their listings were available for rent. After the 100th (or 500th) request, we decided to open up a vacation rental marketplace. Using what we learned the first time around, we got back on the phones and offered resort owners and vacation rental managers free trials. We went as far as inputting all of their information, uploading their pictures, etc… whatever it took to get them to try LakePlace.com. Once they received 10-20 inquiries, we let them know that the free trial was over and it was time to become a paying customer. I think we had a 99% retention rate when converting free trial users to paying customers. Today, LakePlace.com's Minnesota vacation rental section is about the same size as Homeaway and VRB — and way bigger than AirBnB's — and we are a very close second in Wisconsin (I hope to fix that this year).

Lesson #1: What's one way to make a free-to-pay (free trial) transition work? Base it on a success rate, not a time limit.

The (first) Big Pivot

In 2006, at the height of the real estate boom, some of our 600+ real estate advertisers were closing 6-8 transactions per month that could be directly attributed to LakePlace.com leads. If you multiply that number out, the top agents using our website were clearing $30k+ per month for a $59/month investment. Crazy. After looking at all of our options, we decided that we needed to wiggle our way into a piece of the action, and there was only one way to do it: we needed to become a real estate brokerage.

We concluded that we should be "referring" leads to agents in exchange for a 25% referral fee on closed transactions. Because real estate is so heavily regulated, this required us to become a licensed real estate brokerage in two states, which presented a couple of problems:

1. My partner and I were not real estate agents. To become a broker/brokerage in MN and WI (besides a bunch of class hours and exams), you need to have at least two years of experience as a licensed agent.
2. We would have to cancel all of our subscriptions (lose most of our revenue and upset a lot of people).

We decided to make a call to the commercial banker who helped us purchase an office building during our e-commerce days. Not only was he the top commercial banker in the U.S. at a huge bank, he was also an attorney and licensed real estate broker. After a dozen meetings and several dinners with him and his wife, we convinced him to resign his position at the bank, invest some money, and join our team as the broker and CEO. Before jumping in though, he wanted to make sure that the idea for our new business model was sound…so he picked us up in his S600 and we drove 200 miles north to visit our biggest advertiser.

Our top advertiser was a young real estate broker in a popular resort town in northern Minnesota. Every time we launched a new advertising opportunity, he signed up (and paid up) almost immediately. He had just been named one of the "30 Under 30" by Realtor Magazine and was selling $60M in lakeshore properties every year. If we could convince him to buy into our new model, everything else would probably fall into place pretty easily. When we asked him if he was willing to pay a $1,000 annual fee + a 25% referral fee in exchange for market exclusivity on LakePlace.com (vs. $59/month flat fee and no exclusivity), he said "yes" without hesitation. The conversations on the drive back to the Twin Cities were filled with excitement as we solidified the deal with our new CEO and discussed the wire transfer and his start date.

The Call

In preparation for our new business model, Cam and I hit the phones hard. We studied the maps, carved out 53 unique lakeshore markets throughout Minnesota and Wisconsin, and called our top advertiser in each market. We let them all know that we were going to be changing our model and asked if they would be interested in joining LakePlace.com as our exclusive affiliate in their market (annual fee + 25% referral fee). In short order, we filled all 53 slots and validated our new model. On the other side of the coin, the news of our changes spread quickly throughout the real estate industry and we had to field a lot of calls and emails from angry advertisers. Some agents cancelled their advertising subscriptions immediately, others decided to continue advertising to the end and join a waiting list to become an affiliate.

The fuse was lit. In about a month our company would have a licensed broker and a healthy bank account. The plan was to join every MLS (19 in all) in Minnesota and Wisconsin (expensive), pull and combine all of the lakeshore listings from these different databases and build an easy-to-use, seamless search function (difficult). If we could make this happen we would be able to give our users a complete market overview…which is exactly what Cam and I wanted when we were searching for our vacation homes.

The night before our investor/CEO was to give his 30 day notice at the bank, Cam and I were working late. Cam's cell phone rang at about 10pm, it was our investor/CEO's wife. She asked if I was present and then asked Cam to put her on speaker. She started with "I need to have a difficult conversation with you guys…." She went on to tell us that her husband left his law practice because of heart problems, he was put on beta blockers at a very early age, and she could see all of his symptoms coming back. She told us, in no short order: "I'm sorry, but I'm not going to let him do this."

We were floored. We were dead. We had put our reputations on the line with 53 agents/brokers and burned bridges with many others. I

> **“We were junior varsity level players, and it was time to come clean and own up to our inadequacies.”**

can't really explain the feeling that came over me that night, but I can tell you that I hope I never feel it again. Anyway, the next morning, after confirming the news, we had to make our own difficult phone calls. It was time to pull back the curtain and admit that we had no clue about what we were doing. We were junior varsity level players, and it was time to come clean and own up to our inadequacies. The first, dreadful call we decided to make was to our top advertiser, the "30 Under 30" guy. He sat quiet on the other end of the phone while Cam and I did our best to explain why we could not move forward. After a long, awkward silence, the first words he said were "…it sounds like you need a broker and some money. What if I can bring that to the table?" Cam and I looked at each other, eyes wide open, and one of us said "…it would be game on." Forty-five days later he had sold his brokerage, moved his family to the Twin Cities, and joined our company as a minority shareholder and COO. We were back.

Lesson #2: A deal isn't done until it's done.

Lesson #3: Good things are sometimes born of disasters.

The Trough of Sorrow

2006-2009 was full of ups and downs. Almost immediately after launching our new model we entered acquisition talks with 3 large companies. The one company we were most interested in working with took the talks very far, but after several meetings with their C-level execs, lawyers, and IT teams, the deal fell apart. Our new model started out great and was a moderate success, but as the housing market collapsed, our referral fee revenue began to dry up. Markets that were closing 20 referred deals per year in 2006 turned into 1 or 2 closed deals by 2009. Less revenue meant that we did not have the resources to audit our affiliates and the whole thing was spiraling in the wrong direction. To make matters worse, it didn't take long for Cam and I to realize that we had some major personality conflicts with our new COO. After 12 months on the team, we all agreed it would be best if he moved on. The terms of our buy/sell agreement required him to remain the broker of record for 2 years while we bought our shares back. This gave my business partner Cam time to get a real estate license and eventually his broker's license. It also allowed our former COO to earn 3x on his investment (not great, but not bad).

Note: While things did not go as planned, our COO was able to parlay his experience at LakePlace.com into a Director of Franchise Sales position at a big, national real estate company (he also became a contestant on "The Apprentice"), and we were able to keep our business alive as a licensed real estate brokerage. All in all, we have no hard feelings and I look back on it as a win-win. I think he feels the same.

A New Beginning (The Second Pivot)

By late 2009, after reclaiming 100% ownership of our company, it was becoming obvious to Cam and me that our referral brokerage model was probably not going to be the driver of success that we had hoped for. We had spent more than six years searching for a successful model that we could attempt to scale, and during this time companies like Zillow, Trulia, and Redfin had soared (Facebook and LinkedIn launched at about the same time as LakePlace.com) while we were spinning our wheels. After 30+ years as bff's and more than a decade as business partners, the stress of long hours and shrinking incomes was coming to a head, and Cam and I were starting to resent one another. And then we got the phone call that would change everything.

At the beginning of 2009, I helped one of our affiliates set up a blog, and over the course of the year, I gave him a little SEO advice that helped his blog rise to the top of the SERPs. He worked for a large brokerage that was in the largest lakeshore market in Minnesota (probably the largest lakeshore market in the country). When sh*t hit the fan at his company, he called us right away and asked if we ever thought about opening our own real estate office. It's kind of funny, because even though we had been working directly with realtors and brokers, day in and day out for 6 years, the thought of opening up our own real estate office was never really on the table. Like I mentioned above, our goal was to build a scalable product ("products scale, services don't…blah, blah, blah").

After a lengthy conversation, Cam and I got excited and agreed to setup a "secret" meeting at a remote lake home with a dozen realtors from our affiliate's brokerage. At the end of our presentation, all twelve agents at the table committed to joining LakePlace.com if we committed to opening an office. In April of 2010 LakePlace.com-Crosslake (Brainerd Lakes) opened its doors.

Lesson #4: Helping people can provide unexpected returns on investment.

So how's switching from a product to a service going so far? In 2009 we received two (2) referral fee checks from our affiliate in 1/53 markets. In 2010 (April-Dec), after opening our own office, our agents closed 53 transactions from company generated leads in that market. When we ran the numbers across all of our affiliate markets, the path forward was obvious… so we started executing. Here's the timeline:

- 2003: LakePlace.com launches as classified ads website
- 2006: LakePlace.com pivots into a referral brokerage
- 2010: LakePlace.com opens first real estate office in Crosslake, MN (LakePlace.com-Crosslake)
- 2011: LakePlace.com acquires ReMax Woodland Realty (now LakePlace.com-Birchwood)
- 2011: LakePlace.com-Crosslake merges with Century 21 Gold Shores (now LakePlace.com-Crosslake)
- 2012: LakePlace.com opens Wayzata, MN office (LakePlace.com-Metro)
- 2012: LakePlace.com opens new headquarters in Bloomington, MN
- 2012: LakePlace.com merges with ReMax Northwoods Realty (now LakePlace.com-Siren)
- 2012: LakePlace.com acquires ReMax North Country (now LakePlace.com-Hackensack)
- 2012: LakePlace.com opens Detroit Lakes, MN office (LakePlace.com-Detroit Lakes)
- 2013: LakePlace.com opens Alexandria, MN office (LakePlace.com-Alexandria)
- 2013: Loads of new bullet points coming ("knock on wood")

Our sales and revenue doubled in 2010, 2011, and 2012 but we still have a long, long way to go. And even though our business isn't quite the technology product we envisioned when we started, Cam and I are happy again and loving what we do. Every day presents a new challenge, and we get to attack it from an angle that most companies in our industry can't see. We may not be "changing the world," but we feel like we are pioneering a new way to build a successful real estate company, and that in and of itself has been very satisfying.

Lesson #5: Hard work and perseverance pays off.

Final Thoughts

I'm not going to pretend I'm some sort of startup guru with magic advice that will change your life. There are already tons of people out here doing that, and plenty of them have built things that make our company look like a lemonade stand. I will, however, share some things (read: anecdotes) that have worked for me.

1. **Never quit.** I don't mean be irresponsible. I mean if you are working on something that you truly, wholeheartedly believe in, but it's taking longer than you anticipated to get traction, don't stop. It takes time, and it's probably going to be a lot harder and more painful than you thought it would be, but don't quit.

2. **There are riches in niches.** Niches are almost always the best/easiest place to start if you're a bootstrapper.

3. **Sales cure (almost) everything.** Money isn't the most important thing in the world, but it's what we use to keep score…and it keeps the lights on. More money = less pressure…so always be selling

4. **Know your customer.** Your customer is the person who gives you money in exchange for your product or service. It's easier than you might think to get confused about this one.

5. **Listen to your users.** Don't add/remove features on every whim, but if you ask and then listen very closely, you'll find nuggets of gold.

6. **Everything scales.** If you build a successful business model (product, service, whatever), it's scalable. It might not be easy, but it can be scaled. Don't believe me? Ask the founders of Walmart, Home Depot, McDonald's, HandR Block, Re/Max or Fantastic Sam's.

7. **Always think big.** Cameron and I operate with a short-term, mid-term, and long-term plan. The long term plan is our "take over the world" plan. Our 3-plan approach helps to keep us focused on our day-to-day operations but also keeps us alert and looking for opportunities that may help us reach our ultimate goal.

8. **No Excuses.** Can't raise money? Figure something else out. Don't have connections? Neither do we. Cam and I started out on our entrepreneurial journey with $600/each on credit cards (no savings) and did $1M in revenue our first year…because we needed to. I've applied to Y-Combinator twice and talked to local investors a few times. Everyone said "no." We said "fuck "em, we"re doing it anyway."

9. **Pick a fight.** Don't do this publicly, but always have an enemy: at least one person and/or company whose ass you are trying to kick. Don't stop until you have their head on a stick…and then pick a bigger enemy.

10. **It's really, really hard.** Cam and I have gone several months working insane hours without a pay check or health insurance (multiple times). Entrepreneurship is definitely not for everyone, but if you're like me, you can't imagine doing anything else. Ever. ■

Dave Gooden is the co-founder and CEO of *LakePlace.com*, an accidental (but awesome) real estate brokerage. Follow *@davegooden* on Twitter.

Reprinted with permission of the original author.
First appeared in *hn.my/10years* (davegooden.com)

mailjet

EMAIL FOR YOUR APPS

SEND. TRACK. DELIVER.

Your one stop shop for **ALL your email needs.**
Manage lists as well. No extra fees for **Newsletters.**
Priority headers to deliver notifications in **real time.**

Go for mailjet.com

Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com