Eduardo Mourao

How To Land An Airplane

HACKERMONTHLY

Issue 37 June 2013

Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com

Curator
Lim Cheng Soon

Contributors
Robin Dhar
Eduardo Mourao
Jeff Nelson
David Lieb
Chris Taylor
Dominik Dabrowski
Mike Krieger
Andrew Wulf
Sash McKinnon
Jon Bell

Illustrator
Matthew Billington

Proofreaders
Emily Griffin
Sigmarie Soto

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

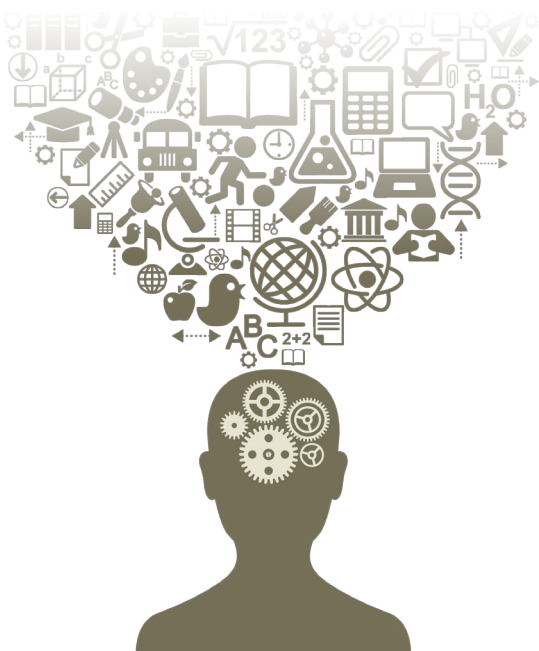
Contact
contact@hackermonthly.com

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

Cover Illustration: Matthew Billington

Contents

Alex Andon, jellyfish entrepreneur.



For links to Hacker News dicussions, visit *hackermonthly.com/issue-37*

The Jellyfish Entrepreneur

By ROBIN DHAR

When Alex Andon got his first order for a $25,000 jellyfish tank installation, he was excited. He also had a problem. He didn't know anything about jellyfish or how to make a jellyfish tank. He had a hunch that people wanted to keep jellyfish as pets, so he created a test website and bought $100 in Google search ads. Lo and behold, his phone started ringing with enquires and he got his first order for the $25,000 jellyfish tank.

Today, Alex's company Jellyfish Art [jellyfishart.com] is the leading company in the jellyfish pet space. In fact, they're pretty much the only company in the space. When they launched over four years ago, the only way to keep jellyfish at home was to pay a custom installer $10,000 – $25,000. After starting as a custom installer, Alex later developed a desktop jellyfish tank that brought the price of jellyfish ownership down to $500.

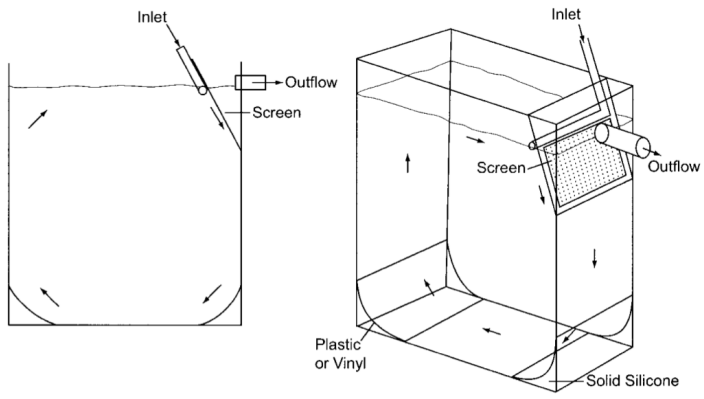
Along the way, he launched one of the first popular Kickstarter campaigns, received funding from Y Combinator, and created a market that didn't exist before.

This is the story of Alex Andon and Jellyfish Art, the world's only jellyfish startup.

Who Wants a Pet Jellyfish?

While the market for pet fish is estimated to be around $2 billion a year, the market for jellyfish is tiny. Part of the reason is that if you put a jellyfish in a regular fish tank, it will instantly be sucked into the filter and die. The other reason, according to Alex, is that until the 1990s there were no jellyfish exhibits at aquariums. In 1992, the Monterey Bay Aquarium took a chance on one, and launched the first major jellyfish exhibit in the United States. It was a smash hit.

The key to housing jellyfish without killing them was developed in 1960s by German oceanographer Wolfe Greve to house plankton.



If a jellyfish tank's water intake and outtake rate are not perfectly in sync, BOOM, you get liquefied jellyfish. Dr. Greve had previously designed a tank that he called the "Kreisel" tank that could solve this problem with a perfectly balanced filtration system (kreisel is German for carousel).

Kreisel tanks look like the fat cross section of a cylinder. A slow circular water flow along the edge of the tank keeps the jellyfish suspended in the middle and away from the filter. All water flowing into the tank is sprayed in a flat laminar sheet in front of the exit screen. If jellyfish get close to the exit screen, the incoming water blows them away to safety. The water flowing out of the tank goes through a screen with sufficiently large surface area to prevent any points of suction that could suck a jellyfish in. Only small particles pass through the exit screen, filtering the tank while the jellyfish remain safe in the center.

Most jellyfish look nondescript. They're practically transparent until you shine LED lights on them or provide a background color.

But if you create the right setting, jellyfish are stunning.

Proof of Concept

In late 2007, Alex was itching to start a company, any kind of company at all. He was two years out of college and living with tech entrepreneurs in a house in San Francisco. He worked as a lab technician at a struggling biotech firm.

A marine biology major in college, Alex noticed that jellyfish exhibits completely mesmerized aquarium visitors: "People seemed to have an obsessive infatuation with the jellies. Some people would sit in front of the tanks for hours staring at them." Since the jellyfish exhibits were so popular, he decided to explore whether there was a market for pet jellyfish.

He discovered that it was possible to keep jellyfish as pets, and possible to catch them as well.

Based on studying the design of jellyfish tanks at aquariums and conversations with breeders, he concluded that it was technically feasible to sell jellyfish to consumers.

But was there any actual demand?

To find out, Alex put up a landing page advertising the services of his (at this point non-existent) custom jellyfish tank installation business.

Alex then started a Google Adwords advertising campaign, targeting search terms like "jellyfish tank." His phone started ringing with potential customers. Before he had spent $100 on Adwords, he made his first sale, a $25,000 custom jellyfish tank for a restaurant opening in Seattle.

The First Sale

Alex made the sale, but now he had a problem: he had to deliver on the tank he promised. Alex had a general understanding of jellyfish tank construction based on googling around and talking to experts, but he didn't have enough expertise to deliver the product.

Daniel Pon, a home aquarium and maintenance expert (who now works at Jellyfish Art in addition to running his own aquarium business) remembers first meeting Alex around this time:

> *"I had lunch with him and afterwards was like "this guy is in way over his head." He doesn't know how basic things about a fish tank work and he's going to make a $25,000 jellyfish tank?"*

The experience of selling his first jellyfish tank was, as Alex put it, "a complete disaster." Eventually Alex found a local aquarium builder to

> **“We drove the tank up to Seattle. It was filled with water and jellyfish so the truck weight was 3 times its legal payload.”**

build the tank on his behalf. He got a fishing permit and caught some jellyfish in a bay near San Francisco. He then had to get the tank and jellyfish up to Seattle for installation while the restaurant was still under construction:

“A little before Christmas, a friend and I drove the tank up to Seattle. It was bad. It was filled with water and jellyfish so the truck weight was 3 times its legal payload.”

“It started snowing really hard on the way up. We had to get chain control and put chains on our tires. I’d never done that before. We went through 3 sets of chains.”

When he arrived in Seattle, the jellyfish were dead. That wasn’t such a big deal because they could be replaced. The main issue was setting up the jellyfish tank. Alex worked for five days straight with the construction company to get it installed properly. He slept at the construction site every night.

Alex got the tank installed in time for the restaurant’s opening. But the tank had a few hiccups. One day, a pipe broke and dumped 100 gallons of water into the restaurant. Other minor problems arose, too, though according to Alex, the restaurant was annoyed, but pretty cool about it. They still use the tank today, but now for fish.

Alex Andon, Jellyfish Consultant

And so with one customer under his belt, Alex decided to go into the jellyfish business. His website and advertising campaign kept producing customer leads for people that wanted custom jellyfish tanks installed. At the same time, the biotech company Alex worked for was struggling during the recession and looking for volunteers to leave the company in exchange for severance. Alex left biotech and committed to jellyfish.

He found working in the custom jellyfish tank installation business brutally difficult, but he earned an understanding of tank design, and the aquarium and pet supply industry.

Alex realized that he needed to build an affordable jellyfish tank. Over the course of a year, he finished only 3 custom installations. The market for $25,000 tanks was very small and too labor-intensive to scale. Instead of selling his installation services, he needed to sell a product.

Around February 2009, Alex put up a landing page on his website offering a desktop jellyfish product for around $500. He put up a photoshopped image of a tank that didn’t quite exist yet. Based on the advice of his software engineer roommates, he posted it to Hacker News.

Around this time, he got his big break, even though he wasn’t quite ready for it. The New York Times profiled him in article about people starting businesses after they lost their jobs during the recession. The article led to an influx of traffic to his site, but his affordable desktop jellyfish tank wasn’t ready yet, so it didn’t lead to any new sales. Still, the article put Alex on the map as “the jellyfish entrepreneur.” From March 2009 onwards, almost every article about jellyfish in the popular press mentioned Jellyfish Art.

> **“Alex monitored the flight online and picked up the jellyfish at San Francisco Airport, like you might pick up your in-laws.”**

The Desktop Jellyfish Product Version 1.0

A few months after the New York Times article, the first version of the company's desktop jellyfish product was ready for sale. It was a bit of a Franken-aquarium, hacked together from various off-the-shelf aquarium parts. But it worked. It kept the jellyfish alive, made them look pretty, and cost around $500.

Sales of the desktop jellyfish tank started to take off. The New York Times article ushered in a wave of articles by other publications about Jellyfish Art. Now, when visitors came to the site, they could actually buy the product. It began to look like a real business with a scalable product.

The increase in sales, however, exposed a critical problem in the business that Jellyfish Art struggles with to this day. Alex had succeeded in making an affordable jellyfish tank that people wanted. But where was he going to get a reliable supply of jellyfish to sell?

The Jellyfish Supply Chain

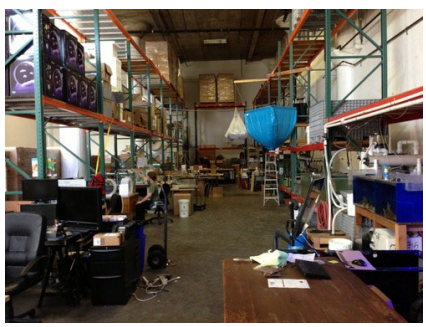
When Alex started Jellyfish Art, he caught the jellyfish himself. He got stung frequently. As an aside, you are NOT supposed to urinate on a jellyfish sting. This appears to be an urban legend derived mostly from an episode of Friends in the 1990s. Just flush it out with vinegar or if that's not available, salt water. Okay?

But back to the subject at hand. Where did Alex get the jellyfish supply from?

> *"Basically, I just asked everyone. One local aquarium gave me a list of a few people who might be able to help. One of them was my guy in [place redacted for competitive reasons] and he worked out."*

This supplier, who lived in a tropical island far from San Francisco, put 500-1000 jellyfish in Styrofoam coolers and shipped them via commercial carrier to San Francisco. This means they flew in the cargo section of a regular passenger plane. Alex monitored the flight online and picked up the jellyfish at San Francisco Airport, like you might pick up your in-laws.

The jellyfish stock is kept at the company's warehouse and office in Potrero Hill, San Francisco.



When an order is placed, they ship the jellyfish to the customer by FedEx overnight. Jellyfish can survive 48-72 hours in shipping so even if there is a delay, the jellyfish normally shows up alive. By contrast, fish typically die after twelve hours of transit. The average jellyfish lives for 6 months and Jellyfish Art guarantees that they arrive alive.

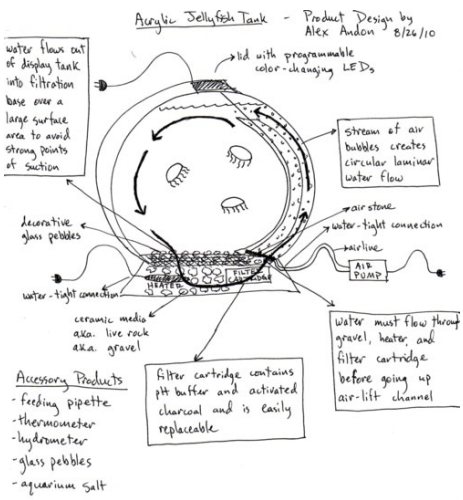
The supply chain worked this way for a year. Then one day, the tropical supplier went to his jellyfish catching spot and couldn't catch a single one. All of them were gone. Every week he checked out the same spot, but every week he went home empty-handed.

But Jellyfish Art survived. Thanks to his increased market exposure, it was easier to get jellyfish. If you breed jellyfish and want to sell them, Alex is the only game in town. He managed to find a decent supplier in Europe that provided just enough supply.

Let's Make Our Own Product

After a year of rising sales selling another company's fish tank retrofitted with their own filtration system, Alex decided it was time for Jellyfish Art to develop its own tanks. Buying someone else's tanks was expensive and manually retrofitting each one was a pain.

At this point, two and a half years after getting started, Alex knew enough to design Jellyfish Art's signature product: the desktop jellyfish tank. His original "napkin" design is below:



It took another year to get to production. By March of the next year, they had a barely functioning prototype that they unveiled at the Global Pet Expo. The Expo is the largest trade show in the pet industry that is dominated by companies with huge marketing budgets. It normally caters to dog and cat owners. Alex and his team had the smallest booth, but they won best new product of the year in the aquarium category.

And Now, 3 Years After Its Start, Jellyfish Art is an Overnight Success

After winning at the Expo, things start happening pretty fast for Jellyfish Art. They found a Chinese manufacturer for their tanks, but it was still going to be expensive to kick off production, so they secured a small business loan. Around the same time, Alex heard about Kickstarter. He figured it could be a good way to get orders and fund the manufacturing.

In August 2011, they launched a Kickstarter campaign aiming to raise $3,000. This was the amount of pre-orders they had gotten so far based on being in the New York Times and winning the Pet Expo. It seemed like an aggressive but doable goal.

They ended up raising $162,917 on Kickstarter. For the first few days of the campaign, sales trickled in. Then rap artist Jermaine Dupri tweeted out about the campaign and it massively spiked. After that tweet, everything changed in the campaign. More blogs started covering it, and Alex went on local TV and radio to talk about the campaign.

Almost immediately after they were "blowing up on Kickstarter," Alex and Jellyfish Art decided to apply to Y Combinator, the technology startup incubator and investment firm. In their application to Y Combinator, they posited that they could use jellyfish as a beachhead to become the "Amazon for pets." They were accepted into the Y Combinator Winter 2012 batch.

Mistakes Were Made

After a meteoric rise in the fall of 2011, gravity set in during the winter of 2012. As they started Y Combinator, Alex realized that the "Amazon for pets" idea wasn't a very good one. Shipping around live animals in boxes like Amazon is a niche industry with low margins. The big money in pets is in dogs and cats.

After toying with the idea of creating a database of dog breeders to connect people with the types of dogs they want, Alex decided against the idea. Their existing business, Jellyfish Art, offered no advantages for starting a dog breeder matching service. Between starting a breeders database from scratch and staying in the world of jellyfish, they chose jellyfish.

During YC, Alex felt pressure to have a jellyfish sales chart that was "up and to the right." Immediately after they shipped off the jellyfish tanks to their Kickstarter backers, they launched a sale on Fab.com, a flash sale site. The sale on Fab.com was their single largest source of sales ever, but it came at a

cost. Jellyfish Art offered the same discount on Fab as they offered their Kickstarter backers. The Kickstarter backers were livid that they received the same treatment even though Kickstarter backers funded the business and put up with a 6 month wait. Some of the people that should have been the biggest supporters of Jellyfish Art turned on the company.

In the wake of massive sales growth from Kickstarter and Fab, Alex started hiring for staff and investing in systems to make the business work. Sales were skyrocketing every day, and it was unclear just how massive this business could become. Alex explains:

> *"As fast as money was coming in the door, it was flying out. We also had no idea how high the sales would go, whether we should be bracing for more growth or planning for stability."*

> *"It turned out our product was too expensive for many of the large retail chains that originally showed interest, so sales eventually leveled out."*

As time went on, it became clear that sales wouldn't continue to rise as quickly as they had in the past few months. What was previously a profitable business was now barely so because they were spending money as if it were a high growth startup. As sales started to flatten out, Alex let go of half his staff.

Despite deciding against the "Amazon for Pets" idea, they received offers from investors to fund the idea after YC Demo Day. Alex decided to turn down the funds. Even if someone was willing to fund it, it was not the business he wanted to start.

The Current Situation

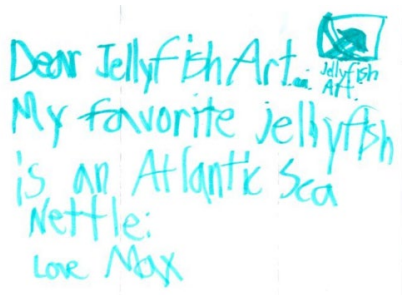
According to Alex, over the last year, sales have been strong but flat. Anyone in the world willing to pay $500 for jellyfish buys the product from Jellyfish Art. The first reason that sales are flat is because they have almost 100% market share. It's hard to improve on that. No competitors have emerged, and if you Google anything jellyfish related, you inevitably end up at the Jellyfish Art website or read an article about the company. Even Wikipedia uses images of their products in its entries. When asked if he was worried about competition, Alex demurred. The Jellyfish Art marketing machine would be hard to unseat.

So, the market of people willing to spend $500 for a jellyfish tank and jellyfish is basically tapped. According to Alex, what Jellyfish Art needs to do is roll out a $100 tank and open up a much larger market and get shelf space in large retailers.

In fact, they've already built a low cost jellyfish tank, but they can't release it. If their demand increased 5-10 times by offering their cheaper tank, they couldn't source enough jellyfish. They currently sell just about every jellyfish they can get their hands on.

The whole fate of the business hangs on whether they can breed their own jellyfish. They're sort of getting the hang of it, but right now they breed 10% of their supply and buy the other 90%. Once they reliably nail the process of producing jellyfish, then they can lower their prices grow again.

Conclusion



At every stage in Jellyfish Art's evolution, Alex Andon definitely sold well ahead of his capabilities. Without knowing much about jellyfish, he sold a $25,000 tank. Before he could build his own tank, he jerry-rigged another company's tank to hold jellyfish. Once he had a barely functioning prototype, he entered it in a trade show and won first prize. Alex was a "fake it till you make it" entrepreneur par excellence.

But then, he made it. The home jellyfish market is small, but Jellyfish Art both created it and dominates it. With the business stable, Alex has had time to experiment with living in a van in San Francisco (he doesn't recommend it), building a website for artisans to sell their crafts, and figuring out what he wants to do with his life.

Living in San Francisco with a bunch of tech entrepreneurs has heavily influenced Alex. He will tell you that what he really wants to do now is start a successful tech company. Alex thought he could turn Jellyfish Art into a tech business, but he couldn't. It's a jellyfish business. But that's still pretty awesome. ■

Rohin Dhar is the co-founder of Priceonomics Price Guides. He is also the co-founder of Personforce job boards and has an MBA from Stanford and BA from Dartmouth.

How To Land An Airplane If You Are Not A Pilot

By EDUARDO MOURAO

At flight school people always ask me:

"Can I land a plane? I have X years of flight simulator experience."

The short answer is: history shows you will probably die. Not necessarily because it is difficult, but because you don't know what you don't know. Flight simulators distort important aspects of landing airplanes: your awesome 200 degrees/3D vision, the muscle mechanics of flying and the notion of distance. In fact, flight simulators are harder than the real thing. Yet, many people in the flight school I teach landed their first flight without the need for intervention. I landed an airplane for the first time when I was 11 years old. With a bit of luck you can do this by yourself. So, in case of an emergency, this is what you could do...

There are many types of airplanes, but when it comes to landing there are pretty much two classes: heavy and light. This is more related to lift/weight ratio (and wing type) than the actual the size of the airplane. I will show you how you can land light airplanes in the easiest way possible, as long as you first open any flight simulator right now (xplane for iPhone is ok) and understand the basic controls (pitch, roll, yaw) and the relationship between speed and angle of attack. Playing with it for 10 minutes (at low speeds) should suffice. The basic mechanics are good enough. Keep in mind that brains are incredible machines and can learn things automatically, but you need to stay calm. For instance, my wife learned how to keep an airplane flying without one single instruction.

The pilot died, now what?

1 Stay calm; take control

Flying airplanes is easy, but stay calm. Your only goal now is to take control. The first thing you will do is put your hands on the yoke (or stick). Do not make sudden moves. Airplanes are not like cars; inputs must be very subtle and smooth. If the airplane is not leveled, you will instinctively and smoothly move the yoke to make the airplane level against the horizon. At this point the plane will be likely going up or down a bit (maybe you are not even aware of that), but don't worry about this now. Check if the pilot's body is blocking or pushing anything, and check if the pedals are clear (don't touch the pedals). The throttle is the black (or gray) lever in the middle of the panel or between the seats. Now, push the throttle forward until you feel where the end is and then pull back 30% of that. You should be using 70% of the engine for now. That will prevent you from crashing for a while. If you are flying a propeller airplane check the RPM (just like a car, it should be right in front of you). The RPM should be around 2.300. The speedometer and tachometer should have colors; never let it get close to the yellow or red areas. **(yellow = this is wrong, red = you are doomed)**

2 Inform the situation

Put the headset on. There should be a BLACK push button at the right side or your yoke. Your radio should be set with an approach or center. Push the button (keep pushed) and speak slowing (but briefly) what happened, start by saying "PAN PAN" or MAYDAY (depends on the country). Do not lose more than 15-20 seconds doing this since we still need to find an acceptable airspeed for this airplane! Also, you will need to find a place to land. If you know where you are, great! Also, the guys on the radio will tell you which direction you should go. If no one answers to you just keep flying straight (do not make turns yet). Ask them if there is a pilot around that knows that specific aircraft (this should help you with finding the speed).

3 Flying straight and airspeed

Now, the speed. I don't know which airplane you are flying (you probably don't either), so you will need to test a reasonable approach/flight speed. We do this by using the airplane's altitude/angle of attack. Airplanes usually fly at a certain angle of attack; the slower you fly the greater the angle of attack (and engine power) needed to continue a leveled flight (because it increases the lift and drag). What we want is to find a speed where the airplane flies with a very small angle of attack. To do that you will accelerate the airplane to about 70-90% of the throttle. Now, looking at the altimeter, stay at that altitude and start pulling the throttle back (slowly) and watch the horizon closely (or artificial horizon). As soon as you need to push the yoke back to keep flying at

the same altitude, check the speedometer: that's 90% of the speed you should be flying now. Accelerate a bit to get to the correct speed. Remember this: everything is subtle; do not push back the yoke to the point where the nose is going up more than 6-8 degrees. If there is an artificial horizon, your angle should be just enough to make a thin blue line between the piece presenting the airplane and the yellow/brown background representing the ground.

4 Making turns

Making correct turns are hard. It takes 20+ hours of instruction to teach students how to turn correctly and they still make turns that suck. You are not going to learn this. With that said, this is how your turns will work: first, where do you want a go? If this is a 180 turn, look 90 to the direction you want to turn, find a reference (trees, etc. — use the tip of the wing) and start the turn by very slowly turning the yoke to the direction you want. This is so delicate that someone looking at your hands would barely notice you are actually moving the yoke. As soon as your reference is on the other site (same position relative to the wing), you finished the turn 180 turn. The maximum angle of turning you will use for this entire flight is no more than 10 degrees (tilt your head to the right/left a little, that's more than enough). You might get a bit dizzy because you are making the turns wrong, but ignore it.

5 Approaching

Now, the guys on the radio guided you to the runway or you found the runway yourself. This is the part where flight simulators are useless. You should be higher than you probably think. Most people have a wrong perception of the height the airplanes approach because the size and direction of airliner passengers windows. That makes first-time pilots come in too low, especially flight simulator players.

You should be at least at 1000 feet above the ground. If you see the number 29.92 set in the altimeter, ignore the altimeter completely. Ask on the radio for this altimeter setting (change the setting by turning the knob on the altimeter). If you don't have a radio or GPS, try this: you should be high to the point where you can see cars but cannot possibly identify the specific color or model. This is around the 45th floor of a building. Remember: the altimeter is showing your altitude relative to a sea level configured by that number, not the distance to the ground.

Your approach will consist of getting the airplane at this height and 1 mile (or less) apart from the airfield, aligned with the runway.

6 Landing

Anyone who plays with a flight simulator should get to this point without any instructions, but now things will get stupidly fast. Adrenaline and not knowing what you are doing are the main reasons for this. To land you will have to forget everything you know about xplane of Microsoft's flight simulator.

Using small movements, you will keep the runway between your legs. Be patient and make only small corrections. If you over correct you will start zigzagging. Airplanes are like kayaks, they are always skidding and inertia make things take a bit longer, you need to wait for your input to make a difference (this impression is actually caused by our notion of space).

We should find a distance between you and the runway at which you could turn off the engine at your current altitude and still reach the runway. We can't do that now, but the good thing about light airplanes is that they lose speed very fast. With that said, you will stay at the current speed (or the speed someone tells you on the radio). If you can locate the control to lower the flaps to its next position, do it now. This will feel like the airplane is braking and could gain altitude, but keep you current speed; the flaps won't break.

Time to dive: you kept the runway between your legs, you are 1000 ft. above the ground and the runway is 1 mile in front of you. You will point the nose of this airplane to the very beginning of the runway. The speed will start to grow and you will reduce throttle to keep your current speed. You will not overshoot the runway; don't worry about that. Keep your

> **“Controlling an airplane on the ground sometimes feels like driving a shopping cart backwards at 60 miles/hour.”**

eye on the speed. Some people will feel the pressure changing in their ears, and this is normal. You point the nose of the airplane to the beginning of the runway, but you won't be able to land there. You should cross the beginning of the runway at the height of a 4-5 story building and descending. When you reach the height of a common pole, cut the throttle completely. You will start to pull and reduce the descending speed. If you pull it sooner it will get ugly. The airplane should be as high as a very tall person now. Do not let it land. Smoothly keep pulling it more and more to try to keep this height. After a few seconds you will hear a buzz, which means the plane is starting to stall. Because you followed my instructions, you should be around 1.5 meters from the ground and the plane will land by itself. If you ever hear that buzzing sound and the distance between you and the ground is greater than a height, you can fall on your feet. Push the throttle to the end and do not pull the yoke until the buzzing stops. Get altitude and try everything again. If the airplane hits the ground, immediately cut the throttle. Some landings can be so hard that they can hurt a bit.

After the plane is on the ground, it won't go straight. It will turn to the left or right immediately after you touch the runway. The pedals, which you haven't used until now, are also used to brake and control the airplane on the ground. Do not put your whole foot on the pedals. Instead, you will put only the tips of your toes on the lower part of them (like kicking). If you push the left pedal, the airplane goes to the left (and vice versa). If you push the upper part of the rudder you will brake one of the main wheels. Unlike cars, every main wheel has its own braking pedal. Do not brake the airplane now; wait for it to get slower. When the airplane is slow, move the tip of your toe to the upper part of the pedal and push left and right, slowing and simultaneously. Controlling an airplane on the ground sometimes feels like driving a shopping cart backwards at 60 miles/hour.

You made it! Now just push buttons around and you will end up turning the engines off (red ones first).

Conclusion (TL;DR)

Playing 10 minutes with a flying simulator will make you more comfortable maintaining the airplane in the air, but it won't help you on landing. The biggest mistake most first timers make is coming too slow and too low for landing. Make sure you are high and glide to the ground without the need to use the throttle (but use it if you need it). Do not fear the ground and start to flare only when you are 10 meters high (same height as a pole). When you reach the height of a tall person, keep pulling until the airplane stalls. You have the option to give up before touching the ground, but never try to take off after touching the ground. Don't ever push the yoke when close to the ground because you will be certain to crash. If you fly by 2/3 of the runway, apply full throttle and try again. If you are on a newer/larger airplane, you will need a pilot on the radio to help you, but the good news is that it could be possible to program the airplane to land by itself.

Remember: you have time and you can keep trying as long as you can keep the airplane flying. ■

Former commercial pilot, Eduardo is the founder & lead engineer of a credit card company and founder of a startup in Brazil. He is also a flight instructor and avid sport biker.

Reprinted with permission of the original author.
First appeared in *hn.my/land* (eduardo.intermeta.com.br)

Illustration by Matthew Billington.

Inventing Chromebook

By JEFF NELSON

While working for Google back in 2006, I had the good fortune to create a new operating system.

I confess it wasn't created from scratch; it was a chopped down Linux distribution, as so many "new" operating systems are these days.

This new operating system was originally code-named "Google OS" and since 2009 has been released to the public under the product names, Google Chrome OS, Chromebook, and Chromebox. I wrote a patent for it, #8,239,662, titled "Network-based Operating System Across Devices" that was finally granted in August 7, 2012, long after I left Google.

Here are few interesting tidbits about the invention of Chromebook.

First, Chromebook was initially rejected by Google management. In fact, I wrote the first version as early as July 2006 and showed it around to management. Instead of launching a project, the response was extremely tepid. My boss complained, "You can't use it on an airplane." Actually, you could since, under the covers, it was still a bare-bones Linux distribution and could execute any Linux program installed on it.

Second, Google OS was not originally written for Chrome or called "Chrome OS." The first versions were all based on Firefox. When I wrote the first version in 2006, Google had not yet started developing a web browser of its own, nor had the name "Chrome" existed as a Google product. Chrome versions followed in 2007, after internal beta test versions of Chrome started to be passed around inside Google.

Third, Chromebook was definitely not intended to be "another device" for web browsing — as many product reviewers have characterized the Samsung Chromebook models. The first versions were bare-bones Linux distributions, but fully functional for many tasks, including code development for a Google engineer. I myself used versions of Chromebook, exclusively, every day, for over a year as my primary development box, taking it on many business trips and even some airplanes.

Fourth, the main priority of Chromebook — originally — was not to write a webapp-only operating system. In fact, the main priority when I started constructing the operating system was the need for speed — to create a super-fast operating system.

Why bother to write a super-fast operating system? I was frustrated with Windows and Linux, which I perceived were unnecessarily slow. For example, at that time my occupation was writing webapps for Google, so I was restarting my web browser frequently, sometimes hundreds of times a day, to clear browser cache and cookies as part of the code development process. Restarting the web browser was a particularly slow operation, often taking 30-45 seconds, whether IE or Firefox, Linux or Windows. (Chrome not being available in 2006.) However, even simple tasks such as displaying a directory in a file explorer were unreasonably slow operations, requiring several seconds for a task that should be nearly instantaneous. A few seconds here, 45 seconds there, might not sound like much of a delay, but when such delays occur hundreds of times a day, it adds up to a costly amount of time.

The solution? Move the entire desktop operating system into RAM. By moving the entire operating system into RAM, that immediately took off the table the largest performance bottlenecks in the operating system: File I/O.

Very few tasks that an operating system performs are CPU intensive or cause other major delays that can't be attributed to File I/O. By running the operating system entirely in RAM, most such tasks became nearly instantaneous, without having to rewrite or do any performance optimization at all for thousands of applications that make up the operating system. For example, restarting Firefox went from ~45 seconds to ~1 second. Browsing a directory in the file explorer went from ~8 seconds to ~0.01 seconds. Even compiling code became 60% faster, and I could run non-indexed, recursive greps of the entire RAM resident file system in under 15 seconds. Try doing that with a hard disk.

When discussing the RAM resident architecture of the original versions of Chromebook, nearly everyone expressed concerns about data loss. In fact, data loss was not a problem for several reasons. First, many tasks were performed as webapps, so as long as the webapps were well-written, there was no possibility of data loss. Second, I had configured my IDE to auto-save backups to a network drive, so even in the event of a system crash only a few seconds of work could be lost. Third, some version occasionally synced backups to a local storage media. Aside from that and boot loading, the operating system never accessed any local storage media aside from dynamic RAM. Ever.

Running a RAM resident operating system did pose other challenges. First, avoiding the installation of any bloated applications. A bloated application hogging a few gigabytes of hard disk space might not be painful, but hogging a few gigabytes of RAM is. Such bloat had to be avoided by replacing the functionality with webapps.

Second, many software vendors don't support Linux at all. This functionality also was replaced with webapps.

Thus, tracking down webapps to replace any and all functionality normally found on a desktop became a priority. That's how the seeds of the webapps on the Chromium desktop, albeit originally written in HTML and running on Firefox, were planted.

While running your front-end operating system entirely in RAM is a fundamental shift to the status quo of modern operating system architectures, I'm convinced the benefits far outweigh the costs. As we live our lives, connected and online, few or no resources need to be stored on the same computer as the attached keyboard, and those which are stored don't need to be accessed by spinning a magnetic platter. ■

Mr. Nelson has written two books and many magazine articles on Java and cloud computing during his twenty year career as a Java and C++ engineer and tech lead. He has extensive experience in the Big Data and Search industries, building highly scalable web services, and leading engineering teams at such companies as Google and eBay. He holds a Masters Degree in Applied Mathematics.

Reprinted with permission of the original author.
First appeared in *hn.my/chromebook* (jeff-nelson.com)

Cognitive Overhead

Why Your Product Isn't As Simple As You Think

By DAVID LIEB

It's been hard to ignore the massive shift in the last decade toward simple products. The minimalist design aesthetic pioneered by Dieter Rams in the 1960s on alarm clocks and toasters was popularized by Apple and Google in the 2000s on iPods and search boxes. Soon after, Web 2.0 took over, yielding big buttons, less text, more images, and happier users. Startup accelerators and design gurus popped up proselytizing "simplicity!" and the rapid growth of mobile in the last five years has created an almost strict requirement for simple products that work on our new small screens and increasingly small attention spans. Some of the most popular products today (Twitter, Snapchat, Instagram) all have simplicity of design and experience at their core.

This ~~Ain't~~ Is Your Grandma's Internet

So why did this happen, and why mostly in the last 10 years? Some say that good design simply lags behind technology and that design has finally caught up. Others point to the evolution of our devices and our environments — definitely a major factor.

But I believe the high-order bit is even more straightforward: It's only been in the last 10 years that technology products have reached the mass market. The market size of the entire broadband Internet in 2000 was 50 million people; today it is 2 billion people; in a few short years with the shift to mobile it will be more than 5 billion people. This mass market is comprised mostly of people who sit in the middle of the tech-adopter bell curve, and since they aren't product designers, computer programmers, and tech bloggers, they require an even higher degree of simplicity.

"Simple" Isn't What You Think

But "simplicity" comes in many flavors. We can make products simpler by optimizing along a number of vectors:

- minimize number of steps in the flow
- minimize time required
- minimize number of features
- minimize elements on each page
- ....

But the most important, and often most overlooked, is Cognitive Simplicity. This is an idea that slowly emerged as my company, Bump, tried to understand exactly why Bump is so popular, especially in the non-tech crowd. We believe product builders should first and foremost minimize the Cognitive Overhead of their products, even though it often comes at the cost of simplicity in other areas.

> **“We take our ability to cut through cognitive overhead for granted; our mental circuits for our products’ patterns are well practiced.”**

Cognitive Overhead

There isn’t yet much written about cognitive overhead in our field. The best definition on the web comes from a web designer and engineer in Chicago named David Demaree:

> *Cognitive Overhead — “How many logical connections or jumps your brain has to make in order to understand or contextualize the thing you’re looking at.”*

Minimizing cognitive overhead is imperative when designing for the mass market. Why? Because most people haven’t developed the pattern matching machinery in their brains to quickly convert what they see in your product (app design, messaging, what they heard from friends, etc.) into meaning and purpose. We, the product builders, take our ability to cut through cognitive overhead for granted; our mental circuits for our products’ patterns are well practiced.

This is especially pronounced for mass market mobile products. Normal people already have to use more of their mental horsepower to cut through cognitive overhead. Now imagine the added burden of having to do that while on a crowded bus, or in line at Starbucks, or while opening your app for the first time while eating dinner with a friend and texting another. This isn’t 1999 when your users were sitting in their quiet bedrooms checking out your website on a large monitor while waiting for their Napster downloads to finish; they are out in the real world being bombarded with distractions.

My, What Big Cognitive Overhead You Have

To illustrate the difference between generic simplicity and cognitive simplicity, let’s look at a couple products that, on the surface, might be regarded as being simple to use, but rank in my book as some of the most cognitively complex products of late.

- **QR Codes** — Designed to check the simplicity boxes of speed, ubiquity, and small number of steps, QR codes really dropped the ball on cognitive overhead. “So it’s a barcode? No? It’s a website? Ok. But I open websites with my web browser, not my camera. So I take a picture of it? No, I take a picture of it with an app? Which app?”
- **iCloud / PhotoStream** — When we heard Steve Jobs preach the utopian future where all of our photos and data would be seamlessly synchronized among all our devices, we smelled the Apple simplicity we’d all grown to love. But in practice, iCloud is rife with cognitive overhead — it only backs up your most recent photos, it works on certain select apps but not others, you have to create an icloud.com email account for it to sync your mail and notes but not everything else. Oh, and it works on new iPhone and iPads and Macs running OS X v10.7.4 or later, but not your PC or Android tablet. Try explaining that to your mother.

Cognitive Simplicity Winners

So which products really nail cognitive simplicity? Here are a couple examples:

- **Shazam** — An app that magically hears what song is playing and tells you what it is? Seems pretty complex, and what's happening under the covers actually is. But Shazam does a phenomenal job keeping the user's cognitive burden low. They force people to press a button to "start listening," show real-time feedback that shows the app is hearing the sounds, and it buzzes when a result is found. Shazam could have made the flow faster or fewer taps, but it would come at the cost of cognitive simplicity.
- **Nintendo Wii** — In most ways, the Wii was far more complicated than its game console peers in 2006. It used accelerometers and IR blasters and detectors that required setup and calibration, and it was a departure from the mental model most people had for video games. But the payoff was a system with low cognitive overhead — you swing the controller to the left, and the little avatar on screen swings his racquet to the left. And voila, toddlers and grandparents alike suddenly became gamers.

Could Go Either Way?

Finally, a couple of my personal favorite daily-use products that could be argued either way. What do you think?

- **Dropbox** — I love Dropbox. All of my stuff is in my Dropbox; Dropbox is on all my devices; so all my stuff is on all my devices. Pretty cognitively simple. But there are certainly some potential cognitive hurdles, or, perhaps better put, cognitive activation energy required before reaching the low cognitive overhead state. Is Dropbox a folder on your desktop or a cloud-storage website? Oh and it's a program to install on my computer, too? When do things get backed up? Did it work?
- **Facebook** — Facebook started out with very low cognitive overhead — it was a digital version of the paper Facebooks that already commanded high engagement and retention of college kids. Question: Has Facebook's cognitive overhead increased or decreased as it has expanded to the mass market? What cognitive hurdles have arisen recently that weren't present in the past? Should this worry Facebook?

How To make Cognitively Simple Products

Make people work more, not less.
Put your user in the middle of your flow. Make them press an extra button, make them provide some inputs, let them be part of the service-providing, rather than a bystander to it. If they are part of the flow, they have a better vantage point to see what's going on. Automation is great, but it's a layer of cognitive complexity that should be used carefully. (Bump puts the user in the middle of the flow quite physically. While there were other ways to build a scalable solution without the physical bump, it's very effective for helping people internalize exactly what's going on.)

Give people real-time feedback.
If your user has to wonder, "So, did it work?" you've failed. Walk people through using your product like a magician leads the audience through an illusion. Point out the steps along the way, or whatever magic your product is providing could be lost to the user.

Slow down your product.
We've all heard stories of Google's relentless quest for search-result speed, but sometimes you need to let your user understand and appreciate what your service is doing for them. Studies have shown that intentionally slowing down results on travel search websites can actually increase perceived user value — people realize and appreciate that the service is doing a lot of work searching all the different travel options on their behalf.

How To Know If You've Succeeded

Test on the young, the old…and the drunk.

The very young and the very old are even more sensitive to cognitive overhead, as their brains aren't accustomed to the sort of logical leaps our products sometimes require. Grandparents and children make great cognitive overhead detectors.

When you can't find old or young people, drunk people are a good approximation. In fact, while building Bump 3.0, we took teams of designers and engineers to bars in San Francisco and Palo Alto and watched people use Bump, tweaking the product to accommodate.

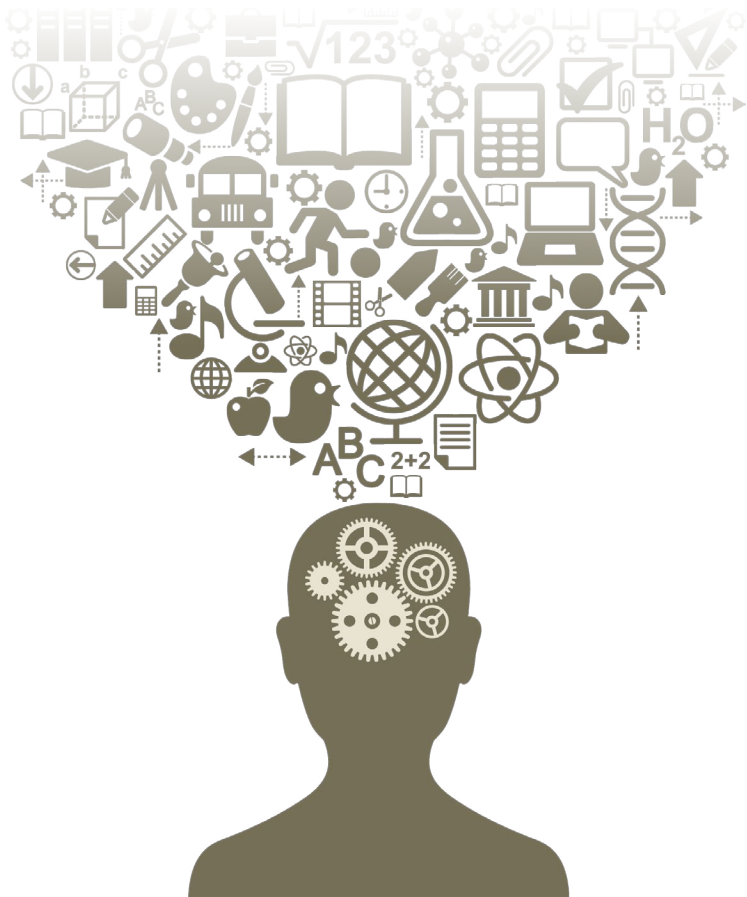
Ask your users/customers to repeat what your product does and how it works.

Let people use your product, and then ask them to tell you what it does. They'll think you are crazy for not knowing already, but what you hear can point to cognitive hurdles you've missed. One technique that scales that we use at Bump is to show a one question survey to a small fraction of users inside the app right after they are done bumping, asking "What is Bump for?" or "How do you use Bump?" The answers help us eliminate cognitive hurdles that remain.

There's never been a time when cognitive simplicity matters more. As the mobile wave continues over the next five years, the world will see arguably the most rapid deployment of any new technology in our history. Products that are truly mass market will need to simultaneously target the Silicon Valley early adopter and the kid riding on the back of a motor scooter in Thailand. Which products will win, and which will lose? My money is on those that focus on cognitive simplicity. ■

David Lieb is co-founder and CEO of Bump, creators of the popular app that lets people share contact information, photos, and other content by bumping their phones together. Bump has been downloaded more than 130 million times.

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



The Algebra of Algebraic Data Types

By CHRIS TAYLOR

In this article, I'll explain why Haskell's data types are called algebraic — without mentioning category theory or advanced math.

The algebra you learned in high school starts with numbers (e.g. 1, 2, 3…) and operators (e.g. addition and multiplication). The operators give you a way to combine numbers and make new numbers from them. For example, combining 1 and 2 with the operation of addition gives you another number, 3 — a fact that we normally express as

```
1+2=3
```

When you get a little older you are introduced to variables (e.g. x, y, z …) which can stand for numbers. Further still, and you learn about the laws that algebra obeys. Laws like

```
0+x=x
1·x=x
```

which hold for all values of x. There are other laws as well, which define properties of numbers or of operations.

When mathematicians talk about algebra, they mean something more general than this. A mathematical algebra has three parts:

- **Objects** are the "things" of the algebra. The collection of objects defines what we're talking about.
- **Operations** give us ways to combine old things to make new things.
- **Laws** are relationships between the objects and the operations.

In high school algebra the objects are numbers and the operations are addition, multiplication and friends.

The algebra of Haskell types

In the algebra of Haskell types, the objects are *types*, for example `Bool` and `Int`. The operators take types that already exist and generate new types from them. An example is the type constructor `Maybe`. It's not a type itself, but you use it to create types; for example `Maybe Bool` and `Maybe Int`, which are types. Another example is `Either`, which creates a new type from two old types; for example `Either Int Bool`.

Counting

A connection to the more familiar algebra of numbers can be seen by counting the possible values that a type has. Take `Bool`, defined by

```
data Bool = False | True
```

There are two values that an object of type `Bool` can have — it is either `False` or `True` (technically it could also be `undefined` — a fact that I'm going to ignore for the rest of the post). Loosely, the type `Bool` corresponds to the number "2" in the algebra of numbers.

If `Bool` is 2, then what is 1? It should be a type with only one value. In the computer science literature such a type is often called `Unit` and defined as

```
data Unit = Unit
```

In Haskell there is already a type with only one value — it's called `()` (pronounced "unit"). You can't define it yourself, but if you could it would look like

```
data () = ()
```

Using this counting analogy, `Int` corresponds to the number 232, as this is the number of values of type `Int`.

Addition

In principle we could types corresponding to 3, 4, 5 and so on. Sometimes we might have a genuine need to do this — for example, the type corresponding to 7 is useful for encoding days of the week. But it would be nicer if we could build up new types from old. This is where the operators of the algebra come in.

A type corresponding to addition is

```
data Add a b = AddL a | AddR b
```

That is, the type a + b is a tagged union, holding either an a or a b. To see why this corresponds to addition, we can revisit the counting argument. Let's say that a is `Bool` and b is `()`, so that there are 2 values a and 1 value for b. How many values of type `Add Bool ()` are there? We can list them out:

```
addValues = [AddL False, AddL True, AddR ()]
```

There are three values, and 3 = 2 + 1. This is often called a sum type. In Haskell the sum type is often called `Either`, defined as

```
data Either a b = Left a | Right b
```

but I'll stick with `Add`.

Multiplication

A type corresponding to multiplication is

```
data Mul a b = Mul a b
```

That is, the type a · b is a container holding both an a and a b. The counting argument justifies the correspondence with multiplication — if we fix a and b to both be `Bool`, the possible values of the type `Mul Bool Bool` are

```
mulValues = [Mul False False, Mul False True,
Mul True False, Mul True True]
```

There are four values, and 4 = 2 x 2. This is often called a product type. In Haskell the product is the pair type:

```
data (,) a b = (a, b)
```

but I'll stick with `Mul`.

Zero

Using addition and multiplication we can generate types corresponding to all the numbers from 1 upwards — but what about 0? That would be a type with no values. It sounds odd, but you can define such a type:

```
data Void
```

Notice that there are no constructors in the data definition, so you can't ever construct a value of type `Void` — it has zero values, just as we wanted!

Laws in the algebra of Haskell types

What are the laws for the types we've just defined? Just like in the algebra of numbers, a law will assert the equality of two objects — in our case, the objects will be types.

However, when I talk about equality, I don't mean Haskell equality, in the sense of the `(==)` function. Instead, I mean that the two types are in one-to-one correspondence — that is, when I say that two types a and b are equal, I mean that you could write two functions

```
from :: a -> b
to   :: b -> a
```

that pair up values of a with values of b, so that the following equations always hold (here the `==` is genuine, Haskell-flavored equality):

```
to (from a) == a
from (to b) == b
```

For example, I contend that the types `Bool` and `Add () ()` are equivalent. I can demonstrate the equivalence with the following functions:

```
to :: Bool -> Add () ()
to False = AddL ()
to True  = AddR ()

from :: Add () () -> Bool
from (AddL _) = False
from (AddR _) = True
```

I'll use the triple equality symbol, `===`, to denote this kind of equivalence between types.

Laws for sum types

Here are two laws for addition:

```
Add Void a === a
```

which says that there are as many values of type `Add Void a` as there are of type `a`, and

```
Add a b === Add b a
```

which says that it doesn't matter which order you add things in. These laws are probably more familiar to you in the algebra of numbers as

```
0+x=x
x+y=y+x
```

If you fancy an exercise, you can demonstrate the correctness of the laws in the Haskell algebra — either with a counting argument, or by writing the functions `from` and `to`.

Laws for product types

There are three useful laws for multiplication:

```
Mul Void a === Void
```

which says that if you multiply anything by `Void`, you get `Void` back,

```
Mul () a === a
```

which says that if you multiply by `()` you don't change anything, and

```
Mul a b === Mul b a
```

which says that it doesn't matter which order you multiply in. The more familiar forms of these laws are:

```
0·x=0
1·x=x
x·y=y·x
```

Two more exercises: (i) prove the validity of these laws in the Haskell algebra, and (ii) explain why we don't need laws of the form:

```
Mul a Void === Void
Mul a ()   === a
```

There's also a law that relates the addition and multiplication operators:

```
Mul a (Add b c) === Add (Mul a b) (Mul a c)
```

This one is a bit trickier to reason about, but writing the corresponding `from` and `to` functions isn't too hard. The arithmetic version of this law is the friendlier-looking

```
a·(b+c)=a·b+a·c
```

called the distributive law.

Function types

As well as concrete types like `Int` and `Bool`, in Haskell you also have function types, like `Int -> Bool` or `Double -> String`. How do these fit into the algebra?

To figure this out we can go back to the counting argument. How many functions of type a·b are there?

Let's be concrete, and fix a and b to both be `Bool`. The value `False` can map to either `False` or `True`, and similarly for the value `True` — thus there are $2 \cdot 2 = 2^2 = 4$ possible functions `Bool -> Bool`. To be really explicit, we could enumerate them:

```
f1 :: Bool -> Bool -- equivalent to 'id'
f1 True  = True
f1 False = False

f2 :: Bool -> Bool -- equivalent to 'const
False'
f2 _     = False

f3 :: Bool -> Bool -- equivalent to 'const True'
f3 _     = True

f4 :: Bool -> Bool -- equivalent to 'not'
f4 True  = False
f4 False = True
```

What happens if b is still `Bool` (with two values) and a is a type with three values, say:

```
data Trio = First | Second | Third
```

Then each of `First`, `Second`, and `Third` can map to two possible values, and in total there are $2 \cdot 2 \cdot 2 = 23 = 8$ functions of type `Trio -> Bool`.

The same argument holds in general. If there are A values of type a, and B values of type b, then the number of values of type a · b is

$$B^A$$

This justifies the common terminology for function types as exponential types.

Laws for functions

There are two laws for function types that involve the unit type. They are:

```
() -> a === a
```

which says that there are as many functions `() -> a` as there are values of type `a`, and

```
a -> () === ()
```

which says that there is only one function `a -> ()` — in particular, it is `const ()`. The arithmetic versions of these laws are

$a^1=a$
$1^a=1$

There is also a law that allows factoring out of common arguments:

```
(a -> b, a -> c) === a -> (b,c)
```

whose arithmetic form is

$b^a \cdot c^a=(bc)^a$

and a law about functions that return other functions:

```
a -> (b -> c) === (b,a) -> c
```

whose arithmetic form is

$(c^b)^a=c^{b \cdot a}$

This last law may be more familiar when the order of the variables in the pair on the right-hand side is switched, and the parens on the left hand side are removed:

```
a -> b -> c === (a,b) -> c
```

which just says that we can curry and uncurry functions. Again, it's an interesting exercise to prove all of these laws by writing the corresponding `to` and `from` functions. ■

Chris Taylor is a researcher at a London hedge fund. He is interested in using mathematics to write safer and more composable programs.

Reprinted with permission of the original author.
First appeared in *hn.my/algebraic* (chris-taylor.github.io)

Python Libraries You Should Know About

By DOMINIK DABROWSKI

In my years of programming in Python and roaming around GitHub's Explore section, I've come across a few libraries that stood out to me as being particularly enjoyable to use. This article is an effort to further spread that knowledge.

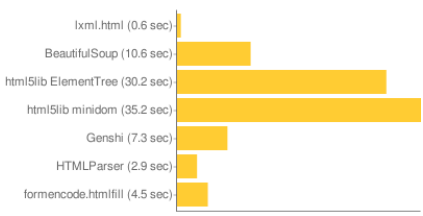
I specifically excluded awesome libs like requests, SQLAlchemy, Flask, fabric, etc. because I think they're already pretty "mainstream." If you know what you're trying to do, it's almost guaranteed that you'll stumble over the aforementioned. This is a list of libraries that in my opinion should be better known, but aren't.

1 pyquery (with lxml)

```
pip install pyquery
```

For parsing HTML in Python, Beautiful Soup [hn.my/soup] is oft recommended and it does a great job. It sports a good Pythonic API and it's easy to find introductory guides on the web. All is good in parsing-land…until you want to parse more than a dozen documents at a time and immediately run head-first into performance problems. It's — simply put — very, very slow.

Just how slow? Check out this chart from the excellent Python HTML Parser comparison Ian Bicking compiled in 2008:



What immediately stands out is how fast lxml is. Compared to Beautiful Soup, the lxml docs are pretty sparse and that's what originally kept me from adopting this mustang of a parsing library. lxml is pretty clunky to use. Yeah, you can learn and use `Xpath` or `cssselect` to select specific elements out of the tree and it becomes kind of tolerable. But once you've selected the elements that you actually want to get, you have to navigate the labyrinth of attributes lxml exposes, some containing the bits you want to get at, but the vast majority just returning None. This becomes easier after a couple dozen uses, but it remains unintuitive.

So either **slow and easy to use** or **fast and hard to use**, right?

Wrong!

Enter PyQuery

Oh, PyQuery, you beautiful seductress:

```
from pyquery import PyQuery
page = PyQuery(some_html)

last_red_anchor = page('#container >
a.red:last')
```

Easy as pie. It's ever-beloved jQuery but in Python!

There are some gotchas. For example, PyQuery, like jQuery, exposes its internals upon iteration, forcing you to re-wrap:

```
for paragraph in page('#container > p'):
    paragraph = PyQuery(paragraph)
    text = paragraph.text()
```

That's a wart the PyQuery creators ported over from jQuery (where they'd fix it if it didn't break compatibility). Understandable but still unfortunate for such a great library.

2 dateutil

```
pip install python-dateutil
```

Handling dates is a pain. Thank god `dateutil` exists. I won't even go near parsing dates without trying `dateutil.parser` first:

```
from dateutil.parser import parse

>>> parse('Mon, 11 Jul 2011 10:01:56 +0200
(CEST)')
datetime.datetime(2011, 7, 11, 10, 1, 56,
tzinfo=tzlocal())

# fuzzy ignores unknown tokens

>>> s = """Today is 25 of September of 2003,
exactly
...         at 10:49:41 with timezone -03:00."""
>>> parse(s, fuzzy=True)
datetime.datetime(2003, 9, 25, 10, 49, 41,
                  tzinfo=tzoffset(None, -10800))
```

Another thing that `dateutil` does for you that would be a total pain to do manually is recurrence:

```
>>> list(rrule(DAILY, count=3,
byweekday=(TU,TH),
...             dtstart=datetime(2007,1,1)))
[datetime.datetime(2007, 1, 2, 0, 0),
 datetime.datetime(2007, 1, 4, 0, 0),
 datetime.datetime(2007, 1, 9, 0, 0)]
```

3 fuzzywuzzy

```
pip install fuzzywuzzy
```

`fuzzywuzzy` allows you to do fuzzy comparison on wuzzes strings. This has a whole host of use cases and is especially nice when you have to deal with human-generated data.

Consider the following code that uses the Levenshtein distance comparing some `user input` to an array of possible choices.

```
from Levenshtein import distance

countries = ['Canada', 'Antarctica', 'Togo',
...]

def choose_least_distant(element, choices):
    'Return the one element of choices that is
most similar to element'
    return min(choices, key=lambda s:
distance(element, s))

user_input = 'canaderp'
choose_least_distant(user_input, countries)
>>> 'Canada'
```

This is all nice and dandy, but we can do better. The ocean of 3rd party libs in Python is so vast, that in most cases we can just `import` something and be on our way:

```
from fuzzywuzzy import process

process.extractOne("canaderp", countries)
>>> ("Canada", 97)
```

4 watchdog

```
pip install watchdog
```

`watchdog` is a Python API and shell utilities to monitor file system events. This means you can watch some directory and define a "push based" system. Watchdog supports all kinds of problems. A solid piece of engineering that does it much better than the 5 or so libraries I tried before finding out about it.

5 sh

```
pip install sh
```

`sh` allows you to call any program as if it were a function:

```
from sh import git, ls, wc

# checkout master branch
git(checkout="master")

# print(the contents of this directory
print(ls("-l"))

# get the longest line of this file
longest_line = wc(__file__, "-L")
```

6 pattern

```
pip install pattern
```

This behemoth of a library advertises itself quite modestly:

> *Pattern is a web mining module for the Python programming language.*

... that does Data Mining, Natural Language Processing, Machine Learning and Network Analysis all in one. I myself have yet to play with it, but a friend's verdict was very positive.

7 path.py

```
pip install path.py
```

When I first learned Python, `os.path` was my least favorite part of the `stdlib`.

Even something as simple as creating a list of files in a directory turned out to be grating:

```
import os

some_dir = '/some_dir'
files = []

for f in os.listdir(some_dir):
    files.append(os.path.joinpath(some_dir, f))
```

That `listdir` is in `os` and not `os.path` is unfortunate and unexpected, and one would really hope for more from such a prominent module. And then all this manual fiddling for what really should be as simple as possible.

But with the power of `path`, handling file paths becomes fun again:

```
from path import path

some_dir = path('/some_dir')

files = some_dir.files()
```

Done!

Other goodies include:

```
>>> path('/').owner
'root'

>>> path('a/b/c').splitall()
[path(''), 'a', 'b', 'c']

# overriding __div__
>>> path('a') / 'b' / 'c'
path('a/b/c')

>>> path('ab/c').relpathto('ab/d/f')
path('../d/f')
```

Best part of it all? `path` subclasses Python's `str` so you can use it completely guilt-free without constantly being forced to cast it to `str` and worrying about libraries that check `isinstance(s, basestring)` (or even worse `isinstance(s, str)`).

That's it! I hope I was able to introduce you to some libraries you didn't know before. ■

Dominik grew up in Austria and started his first business at sixteen, helping to repair gaming consoles. He then studied CS in Vienna for a year before dropping out, instead graduating from HackerSchool batch #3 and now works as a Software engineer at Smarkets.

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

stripe

Accept payments online.

Handling Growth with Postgres

5 Tips From Instagram Engineering

By MIKE KRIEGER

As we've scaled Instagram to an ever-growing number of active users, Postgres has continued to be our solid foundation and the canonical data storage for most of the data created by our users. While less than a year ago, we blogged about how we "stored a lot of data" at Instagram at 90 likes per second, we're now pushing over 10,000 likes per second at peak — and our fundamental storage technology hasn't changed.

Over the last two and a half years, we've picked up a few tips and tools about scaling Postgres that we wanted to share — things we wish we knew when we first launched Instagram. Some of these are Postgres-specific while others are present in other databases as well. For background on how we've horizontally partitioned Postgres, check out our Sharding and IDs [hn.my/sharding] at Instagram post.

1 Partial Indexes

If you find yourself frequently filtering your queries by a particular characteristic, and that characteristic is present in a minority of your rows, partial indexes may be a big win.

As an example, when searching tags on Instagram, we try to surface tags that are likely to have many photos in them. While we use technologies like ElasticSearch for fancier searches in our application, this is one case where the database was good enough. Let's see what Postgres does when searching tag names and ordering by number of photos:

```
EXPLAIN ANALYZE SELECT id from tags WHERE name
LIKE 'snow%' ORDER BY media_count DESC LIMIT 10;
QUERY PLAN
---------
 Limit  (cost=1780.73..1780.75 rows=10 width=32)
(actual time=215.211..215.228 rows=10 loops=1)
   ->  Sort  (cost=1780.73..1819.36 rows=15455
width=32) (actual time=215.209..215.215 rows=10
loops=1)
         Sort Key: media_count
         Sort Method:  top-N heapsort  Memory:
25kB
         ->  Index Scan using tags_search
on tags_tag  (cost=0.00..1446.75 rows=15455
width=32) (actual time=0.020..162.708 rows=64572
loops=1)
               Index Cond: (((name)::text
~>=~ 'snow'::text) AND ((name)::text ~<~
'snox'::text))
               Filter: ((name)::text ~~
'snow%'::text)
 Total runtime: 215.275 ms
(8 rows)
```

Notice how Postgres had to sort through 15,000 rows to get the right result. Since tags (for example) exhibit a long-tail pattern, we can instead first try a query against tags with over 100 photos; we'll do:

```
CREATE INDEX CONCURRENTLY on tags (name text_
pattern_ops) WHERE media_count >= 100
Now the query plan looks like:
EXPLAIN ANALYZE SELECT * from tags WHERE name
LIKE 'snow%' AND media_count >= 100 ORDER BY
media_count DESC LIMIT 10;

QUERY PLAN
 Limit  (cost=224.73..224.75 rows=10 width=32)
(actual time=3.088..3.105 rows=10 loops=1)
   ->  Sort  (cost=224.73..225.15 rows=169
width=32) (actual time=3.086..3.090 rows=10
loops=1)
         Sort Key: media_count
         Sort Method:  top-N heapsort  Memory:
25kB
         ->  Index Scan using tags_tag_name_
idx on tags_tag  (cost=0.00..221.07 rows=169
width=32) (actual time=0.021..2.360 rows=924
loops=1)
               Index Cond: (((name)::text
~>=~ 'snow'::text) AND ((name)::text ~<~
'snox'::text))
               Filter: ((name)::text ~~
'snow%'::text)
 Total runtime: 3.137 ms
(8 rows)
```

Notice that Postgres only had to visit 169 rows, which was way faster. Postgres' query planner is pretty good at evaluating constraints too; if you later decided that you wanted to query tags with over 500 photos, since those are a subset of this index, it will still use the right partial index.

2 Functional Indexes

On some of our tables, we need to index strings (for example, 64 character base 64 tokens) that are quite long, and creating an index on those strings ends up duplicating a lot of data. For these, Postgres' functional index feature can be very helpful:

```
CREATE INDEX CONCURRENTLY on tokens
(substr(token), 0, 8)
```

While there will be multiple rows that match that prefix, having Postgres match those prefixes and then filter down is quick, and the resulting index was 1/10th the size it would have been had we indexed the entire string.

3 pg_reorg For Compaction

Over time, Postgres tables can become fragmented on disk (due to Postgres' MVCC concurrency model, for example). Also, most of the time, row insertion order does not match the order in which you want rows returned. For example, if you're often querying for all likes created by one user, it's helpful to have those likes be contiguous on disk, to minimize disk seeks.

Our solution to this is to use `pg_reorg`, which does a 3-step process to "compact" a table:

1. Acquire an exclusive lock on the table
2. Create a temporary table to accumulate changes, and add a trigger on the original table that replicates any changes to this temp table
3. Do a CREATE TABLE using a SELECT FROM… ORDER BY, which will create a new table in index order on disk
4. Sync the changes from the temp table that happened after the SELECT FROM started
5. Cut over to the new table

There are some details in there around lock acquisition etc, but that's the general approach. We vetted the tool and tried several test runs before running in production, and we've run dozens of reorgs across hundreds of machines without issues.

4 WAL-E for WAL archiving and backups

We use and contribute code to WAL-E [hn.my/wale], Heroku's toolkit for continuous archiving of Postgres Write-Ahead Log files. Using WAL-E has simplified our backup and new-replica bootstrap process significantly.

At its core, WAL-E is a program that archives every WAL files generated by your PG server to Amazon's S3, using Postgres' `archive_command`. These WAL files can then be used, in combination with a base backup, to restore a DB to any point since that base backup. The combination of regular base backups and the WAL archiving means we can quickly bootstrap a new read-replica or failover slave, too.

We've made our simple wrapper script for monitoring repeated failures to archive a file available on GitHub. [gist.github.com/4550560]

5 Autocommit mode and async mode in psycopg2

Over time, we've started using more advanced features in `psycopg2`, the Python driver for Postgres.

The first is autocommit mode; in this mode, `psycopg2` won't issue BEGIN/COMMIT for any queries; instead, every query runs in its own single-statement transaction. This is particularly useful for read-only queries where transaction semantics aren't needed. It's as easy as doing:

```
connection.autocommit = True
```

This lowered chatter between our application servers and DBs significantly, and lowered system CPU as well on the database boxes. Further, since we use `PGBouncer` for our connection pooling, this change allows connections to be returned to the pool sooner.

Another useful `psycopg2` feature is the ability to register a `wait_callback` for coroutine support. Using this allows for concurrent querying across multiple connections at once, which is useful for fan-out queries that hit multiple nodes — the socket will wake up and notify when there's data to be read (we use Python's select module for handling the wake-ups). This also plays well with cooperative multi-threading libraries like `eventlet` or `gevent`; check out psycogreen [hn.my/psycogreen] for an example implementation.

Overall, we've been very happy with Postgres' performance and reliability. If you're interested in working on one of the world's largest Postgres installations with a small team of infrastructure hackers, get in touch at *infrajobs@instagram.com* ■

Mike Krieger is a Brazilian entrepreneur and software engineer best known as the co-founder of Instagram, along with Kevin Systrom. Born in São Paulo, Brazil, Krieger moved to California in 2004 to attend Stanford University. At Stanford, where he studied symbolic systems, he met Kevin Systrom. The two of them co-founded Instagram in 2010.

Reprinted with permission of the original author.
First appeared in *hn.my/instagres* (instagram-engineering.tumblr.com)

MEET MANDRILL

By MailChimp

Mandrill is the fastest way to send transactional, triggered, and personalized emails.
It's also the world's largest species of monkey.

MANDRILL.COM

How Hotel Reservations Work

By ANDREW WULF

A RECENT COMPLAINT FROM a small hotel operator which was posted on Hacker News [hn.my/complain] made me decide to talk about the whole process of reserving a room in a hotel.

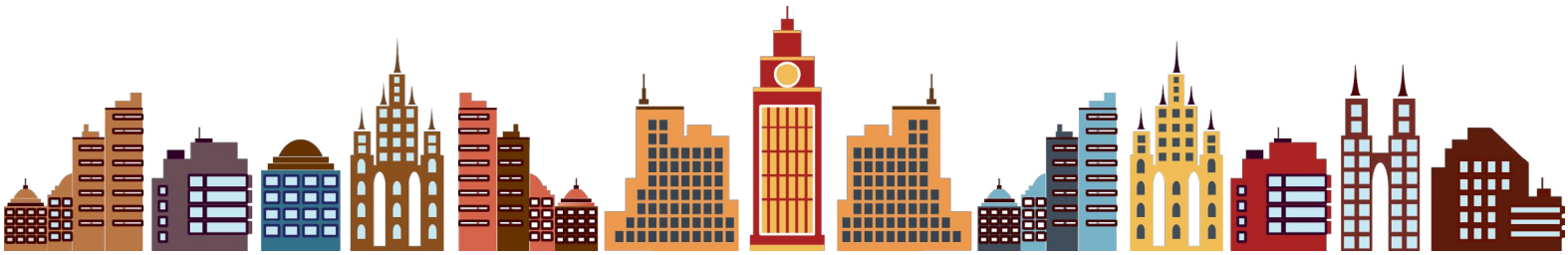
I work for an OTA (which stands for online travel aggregator) which provides flight, hotel, car, and cruise reservations. The major players are Priceline, Expedia, Orbitz and Travelocity. These own many other familiar brands (like lastminute.com is owned by Travelocity, and booking.com is owned by Priceline); plus there are many smaller brands which target niche markets and sometimes provide booking through a major player. Other companies like Kayak and Tripadvisor provide information but handle booking through others as well.

In the U.S. alone there are around 400,000 hotels, motels, lodges, and bed-and-breakfasts alone. Worldwide I have no idea but I am sure there are millions of places to stay. All of them want customers to fill their rooms. Many of them have access to computerized reservation systems, but many still operate on phone calls and fax machines. The challenge as an OTA is how to make this all work. It's pretty crazy.

The average hotel in the U.S. has around 200 rooms. These are available for 365 days a year, so the total room-nights is around 73,000 per year. Each one is a potential reservation. Hotels generally average around 70% occupancy for tonight, which is the only night that really matters, the one where someone is occupying a room. Unlike people selling widgets, who can make fewer widgets or more depending on demand, hotels have a fixed supply. An empty room brings in nothing. A room with guests paying anything is better than an empty room. So the challenge is getting people to sleep in your beds. Over a years' time you need a lot of those people to make it work (that "average" hotel needs 50,000).

The difficulty with making this work from an OTA's perspective is how to allow people to make reservations at, for example, 200,000 properties over the next year. That is 14 billion potential room nights. Now the properties may be part of a large chain, like Marriott, that has a massive reservation system, or a mom and pop motor court operating with a fax machine. Each hotel has a certain number of rooms of different types (queen, king, etc) and these types may be broken down into different rates based on any number of parameters (free breakfast, mobile special rates, multi-night discounts, etc.). Somehow the details have to wind up at the OTA so it can provide them to potential customers. This is where ugly happens.

> **“Even with fancy reservations systems, ultimately an individual hotel manager is responsible for all the data and even the rates.”**



Note that even with fancy reservations systems, ultimately an individual hotel manager is responsible for all the data and even the rates. So each one of those properties has someone who decides what rates there will be, and how often they can change. Even at the large chains, individual managers may ignore or trump the chain's rules in order to maximize their potential sales. Now OTA's have what are generally called market managers (either employees or contractors), whose job it is to deal with the hotels, usually directly, to negotiate special rates or deals or simply sign them up. Some hotels and chains are exclusive to one OTA but many make deals with all of them. Sometimes the deals are complicated. OTA's can either negotiate a discount and sell the rooms themselves and collect the money, then pay the hotel or chain; sometimes they negotiate a commission and get paid later when the guest pays their bill after their stay; sometimes they will reserve actual rooms at a discount and hope to sell them all. The latter is more risky for the OTA since you can get stuck with the rooms, but you have the most flexibility on pricing.

In any case, the hotel is either paid immediately upon the guest making the reservation (which is often preferable) or they have to wait until the end of the stay and then send the commission later (usually much later). Both have advantages, but hotels generally like to get money as soon as possible, as does the OTA. But like all contracted things, the reality might be complicated.

Now if you decide you would rather avoid the OTA, you have to realize that is not so cut and dry either. Often a direct hotel reservation number may not go to the individual hotel, but to a chain reservation line, which is unlikely to give you any special pricing. Often hotels are franchises and are restricted in what they can offer, usually to avoid having related franchises try to kill each other in a local market. Hotels know people hope to get better deals direct and might sell you a room at what you imagine is a discount, except it isn't. Comparing rates between OTA's, chains and comparison sites is always a good idea for hotels (but rarely for flights, that's a much uglier can of worms for another day).

So how does a hotel search work? Firstly, OTAs have to get the hotel descriptions and room type information and prices from the hotels. This can range from a real-time connection to a full reservation system which is used by all the chain's properties all the way to a fax machine and a daily or even weekly update. Availability, which is what we call what rooms are available for a particular date or date range, is always based on cached data. If we had to query external systems to get information for searches we would never

return anything. Like any cached system, this creates the possibility for stale data. The staleness can be both availability (we say the hotel has a room) and price (we tell you it's $100). For searching to work we have to ask the hotel's system periodically for updates or even wait on a weekly fax, and then update the caches. Once you have done a search and have chosen a potential hotel, you are shown the available room types and rates, which can range from one type/rate to dozens at some properties. You then pick a room and express a desire to possibly book it. At this point the OTA system will query the real-time hotel system if available, or the "fax cache" and see if the room is actually available and what the current rate is. Now we will either tell you the room is not really available or note the real price. Sometimes if the room is not available you can choose a different room; sometimes there are no rooms available at all. It's also possible the hotel has rooms but is not making them available to the OTA.

Now you go ahead and either pay for the reservation or at least hold it (depending on the three types I mentioned above). At this point, assuming the payment is approved if we are collecting the money, we call the real-time system again and request an actual reservation, or at least mark the "fax cache" to fax the data. At this point it can still fail as perhaps the last room was reserved while you were filling out the form. The hotel system can also fail, or data connections fail, and you might not get the room either. We generally don't consider the reservation assured unless the hotel system tells us. Of course with the mom and pop hotel, the reservation might get lost or they had no rooms available or any number of problems might greet you when you show up. Always a good idea to call ahead and confirm.

Once you have your reservation, you assume everything will be smooth, and it usually is. Booking a hotel via an OTA usually means there is a hotel reservation number that you will receive in the confirmation or perhaps in a later email. Still, even if a major hotel chain gives you one, it's still possible for the local hotel to lose things or perhaps their local system crashed or their inventory is not exactly up to date. Hotels can also have fires and other issues which might make a reservation become unavailable.

Now, the price you pay is clearly a highly variable thing. We try to negotiate with hotels for special rates; sometimes they might favor one OTA over another. Of course, hotels are competing with each other. Even franchise or chain hotels will often ignore their franchise or chain rules and price things themselves. It's a complicated game of trying to get more people in their beds. Remember a paying customer at any rate is better than an empty room. Managers will do almost anything to improve their bookings.

Hotels are the only thing (maybe cruises) where an OTA makes real money. Cars and flights pay very little and the price differences there are fairly minimal. Billions of room nights make for an appealing marketplace, but also a challenging one to manage. Even a small hotel can make a lot of money if it can attract enough customers, since the supply is fixed and their cost is basically fixed as wel;l the difference is filling the rooms. OTA's can make a lot of money as well, but at the cost of a complicated mass of connected systems of various levels of quality. Now add in multiple countries with all sorts of different rules, mix in contracted market managers who may have their own agendas (which is what it sounds like in Cancun) and hotels desperate to fill their rooms plus all the competing interests like OTA's trying to book your reservations and you have a volatile mix of players.

I work on the customer end (mobile) so some of this is way out of my area, but I've learned enough about the back end to understand how complicated it can be.

This is nothing at all compared to flights, which is mighty ugly stuff. But that's another story. ■

In 3 decades of programming Andrew has worked on almost every kind of software. Currently he works in mobile at a well known travel brand and writes in his blog, *thecodist.com*

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

What It's Like To Die

By SASH MACKINNON

Six months ago, I died.

I have no recollection of the event, but I've heard the story retold so many times that I may as well have seen it all. I was at the gym in my apartment complex with my roommate, Sam. I was running on the treadmill when I turned and told him I was going to faint. I collapsed and fell onto the still-moving belt, which tore the skin off my knee and pushed me onto the floor. Sam was shocked. He called for help. A personal trainer and her client ran over, called an ambulance, and assisted Sam in giving me CPR while my body slowly drained of color.

My heart had gone into ventricular fibrillation. "Vfib," as I heard numerous doctors call it, is a type of arrhythmia — a series of irregular electrical signals in the ventricle chamber of the heart. Instead of beating normally, the walls quiver erratically, like they're having a seizure. The heart quickly becomes unable to pump blood to other organs. I had suffered from what is officially, and somewhat morbidly, termed "Sudden Cardiac Death."

The paramedics arrived and walked slowly down the length of the pool to the gym. This was procedure, they later told me; they didn't want to run and cause alarm. When they reached me, they defibrillated my heart by strapping patches to my abdomen and running a strong electrical current through my body. I was told that after the first administration my heart had remained in arrhythmia. After the second, it started beating regularly.

For those 4 minutes and 30 seconds, I was clinically dead.

I spent the next two days in a coma while the doctors cooled my body to 32 degrees in order to avoid brain damage. During this time I developed a pulmonary embolism and pneumonia. Whenever I visit a doctor now they are always surprised — "Each of those alone could have killed you. It's a miracle you survived all three!" I survived by sitting through hours of MRIs with oxygen in my nose, three IVs in my arm and ten pills a day for weeks. Sam and my two mothers, Laurie and Kerrie, rarely left my side.

The stories you hear about people dying usually end with tunnels, lights, flashbacks, God, and big epiphanies. That isn't what happened to me.

After finally regaining enough consciousness to understand my situation, I sat for hours staring at the hospital walls. I didn't have any life changing realizations. I wasn't regretful. In fact, I couldn't think of anything in my life I wanted to change at all. Being trapped alone in that sterile room with wires hanging off my chest only made me think about everything in my life I wanted back.

Most people I tell this story to think I'm unlucky because I had a cardiac arrest at 21 years old. But I don't think so. Only five percent of people who suffer ventricular fibrillation out of the hospital survive. Of those that do survive, more than half of them have brain damage. That means only two and a half percent fully recover. Not only did I fully recover, but I did so in the company of the people closest to me.

If there is one lesson I took away from the experience, it is not to "live life to the fullest" or "have no regrets." It is to feel lucky. Feeling lucky means you are appreciating the things in your life that sometimes go unnoticed. It means you are achieving more than think you deserve. Feeling lucky requires a certain humility we often lose sight of.

For me, it took losing everything to remember how lucky I am. ■

Sash Mackinnon is an Australian who moved to Silicon Valley to make games. He worked at Zynga as Mark Pincus' technical assistant for a year before joining MinoMonsters. Also he died.

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

McDonald's Theory Of Bad Ideas

By JON BELL

I USE A TRICK with co-workers when we're trying to decide where to eat for lunch and no one has any ideas. I recommend McDonald's.

An interesting thing happens. Everyone unanimously agrees that we can't possibly go to McDonald's and better lunch suggestions emerge. Magic!

It's as if we've broken the ice with the worst possible idea, and now that the discussion has started, people suddenly get very creative. I call it the McDonald's Theory:

People are inspired to come up with good ideas to ward off bad ones.

This is a technique I use a lot at work. Projects start in different ways. Sometimes you're handed a formal brief. Sometimes you hear a rumor that something might be coming so you start thinking about it early. Other times you've been playing with an idea for months or years before sharing with your team. There's no defined process for all creative work, but I've come to believe that all creative endeavors share one thing: the second step is easier than the first. Always.

Anne Lamott advocates "shitty first drafts," Nike tells us to "Just Do It," and I recommend McDonald's just to get people so grossed out they come up with a better idea. It's all the same thing. Lamott, Nike, and the McDonald's Theory are all saying that the first step isn't as hard as we make it out to be. Once, I got an email from Steve Jobs and it was just one word: "Go!" Exactly. Dive in. Do. Stop overthinking it.

The next time you have an idea rolling around in your head, find the courage to quiet your inner critic just long enough to get a piece of paper and a pen, then just start sketching it. "But I don't have a long time for this!" you might think. Or, "The idea is probably stupid," or, "Maybe I'll go online and click around for — "

No. Shut up. Stop sabotaging yourself.

The same goes for groups of people at work. The next time a project is being discussed in its early stages, grab a marker, go to the board, and throw something up there. The idea will probably be stupid, but that's good! The McDonald's Theory teaches us that it will trigger the group into action.

It takes a crazy kind of courage, of focus, of foolhardy perseverance to quiet all those doubts long enough to move forward. But it's possible — you just have to start. Bust down that first barrier and just get things on the page. It's not the kind of thing you can do in your head; you have to write something, sketch something, do something, and then revise off it.

Not sure how to start? Sketch a few shapes, then label them. Say, "This is probably crazy, but what if we…." and try to make your sketch fit the problem you're trying to solve. Like a magic spell, the moment you put the stuff on the board, something incredible will happen. The room will see your ideas, will offer their own, will revise your thinking, and by the end of 15 minutes, 30 minutes, an hour, you'll have made progress.

That's how it's done. ■

Jon Bell is a designer living in Seattle. He writes more about himself on *lot23.com*

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

```
{
    join: 'Intensive Online Bootcamp',
    learn: 'Web Development',
    goto: 'http://www.gotealeaf.com'
}
```

Tealeaf Academy
an online school for developers

Learn Ruby on Rails | Level up Skills | Launch Products | Get a Job

MEMSET® HOSTING

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

MINISERVER™ CLOUD COMPUTE

From $0.020/hour
to 4 x 2.9 GHz Xeon cores
31 GBytes RAM
2.5TB RAID(1) disk

MEMSTORE™ CLOUD STORAGE

$0.091/GByte/month or less
99.999999% object durability
99.995% availability guarantee
RESTful API, FTP/SFTP and CDN Service

SCAN THE CODE FOR MORE INFORMATION

MEMSET® HOSTING

Find out more about us at
www.memset.com

or chat to our sales team on
0800 634 9270.

CarbonNeutral® hosting

THE BRITISH ASSESSMENT BUREAU ISO9001

BSI™ UKAS INFORMATION SECURITY MANAGEMENT 003

THE BRITISH ASSESSMENT BUREAU ISO14001