How To Be An Open Source Gardener

by Steve Klabnik

HACKER MONTHLY

Issue 50 July 2014

Curator
Lim Cheng Soon

Contributors
Steve Klabnik
Derek Sivers
Alex Krupp
Nathan Kontny
Joe Savage
Jesper Louis Andersen
Hunter Loftis
Andrey Karpov
Callum Jefferies

Illustrators
Jaime G. Wong
Yana Dhyana

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: Yana Dhyana [yanadhyana.deviantart.com]

Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-50*

How To Be An Open Source Gardener

By STEVE KLABNIK

I DO A LOT of work on open source, but my most valuable contributions haven't been code. Writing a patch is the easiest part of open source. The truly hard stuff is all of the rest: bug trackers, mailing lists, documentation, and other management tasks. Here are some things I've learned along the way.

It was RailsConf 2012. I sat in on a panel discussion, and the number of issues open on rails/rails came up. There were about 800 issues at the time, and had been for a while. Inquiring minds wished to know if that number was ever going to drop, and how the community could help. It was brought up that there was an "Issues team," whose job would be to triage issues. I enthusiastically volunteered.

But what does "issue triage" mean, exactly? Well, on a project as large as Rails, there are a ton of issues that are incomplete, stale, need more information… and nobody was tending to them. It's kind of like a garden: you need someone to pull weeds, and do it often and regularly.

But before we talk about how to pull the weeds, let's figure out what kind of garden we even have on our hands!

What are Issues?

The very first thing your project needs to do is to figure out what Issues are supposed to be for. Each project is different. For example, in Rails, we keep Issues strictly for bugs only. Help questions go to Stack Overflow, and new feature discussion and requests go to the rails-core mailing list. For Rust, we have issues for feature requests, meta-issues… everything. For some repositories, closing all of the issues is not feasible, and for others, you're shooting for zero. (If you don't believe that this is even possible, check out Sequel [hn.my/sequel]. Issues are rarely even open for more than a few days!)

My personal favorite is to follow the Rails way. Ideally, you'd be at zero defects, and you can still have a place to discuss features. But really, having some plan is a necessary first step here.

Regular tending

So how do you tackle 800 issues? The only way I knew how: read all of them. Yep. Here's what I did: I took a Saturday (and a Sunday), and I went to the list of open Issues, then control-clicked on each one in turn to open them in a new tab. Finally, I also control-clicked on page 2. Then I closed this tab. Now I had 31 open tabs: 30 issues, and the next page. I read through the whole issue, including comments. When I got to the last tab, I was ready to repeat the process: open 30 issues, open page 3, click close. Next!

See, people think working on open source is glamorous, but it's actually not. Working on open source is reading 800 issues over the course of a weekend.

Anyway, once I read all of those issues, I was significantly more informed about the kinds of problems Rails was facing. I had a whole bunch of common questions, comments, and problems.

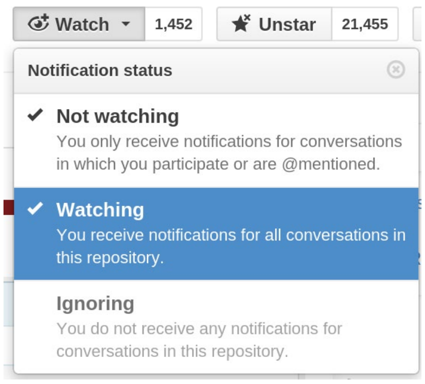
The next step was to do it all again.

Wait, again? Why? Well, now that I had a handle on things, I could actually take on the task of triage-ing the issues. If I'd tried to do it before I had the context, I might not have seen the duplicate issues, I wouldn't know what the normal kinds of comments were on issues, I wouldn't have known some common questions that maintainers had on pull requests, and in general, things would have just been worse.

This time, when reading the issue, I went through a little algorithm to sort them out. It looked a little like this:

1. Is this issue a feature request? If so, copy/paste an answer I wrote that pointed them to the mailing list, and click close.
2. Is this issue a request for help? If so, copy/paste an answer I wrote that pointed them to StackOverflow, and click close.
3. Was this issue for an older version of Rails than is currently supported? If so, copy/paste an answer I wrote that asks if anyone knows if this affects a supported version of Rails.
4. Did this issue provide enough information to reproduce the error? If no, copy/paste an answer I wrote that asks if they can provide a reproduction.
5. If the issue has a reproduction, and it wasn't on the latest Rails, try it against HEAD. If it still happened, leave a comment that it was still an issue.
6. If we got to this point, this issue was pretty solid. Leave a comment that I had triaged it, and cc the maintainer of that relevant sub-system of Rails, so they could find issues that pertain to the things they work on.

At the same time I did this, I clicked this button on the GitHub interface:

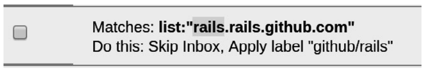


✓ **Not watching**
You only receive notifications for conversations in which you participate or are @mentioned.

✓ **Watching**
You receive notifications for all conversations in this repository.

Ignoring
You do not receive any notifications for conversations in this repository.

And then set up a Gmail filter to filter all of the emails into their own tag, and to skip my inbox:



Matches: **list:"rails.rails.github.com"**
Do this: Skip Inbox, Apply label "github/rails"

Why do this? Well, I didn't do all 800 immediately. I decided to do one page per day. This kept it a bit more manageable, rather than taking up entire days of my time. I need these emails and filters for the important second part of the process: tending to the garden regularly.

Each morning, before I go to work, I pour a cup of coffee and check my emails. I don't handle all of them before work, but I made an effort to tackle Rails' emails first. There would usually be about 20 or 25 new emails each morning, and since it was largely just one new comment, they'd be pretty fast to get through. 15 minutes later, I was back to current on all issues. At lunch, I'd do it again: ten minutes to handle the ten or so emails by lunch, and then, before I'd go to bed, I'd do it again: 15 more minutes to handle the next 20 notifications. Basically, I was spending a little under an hour each day, but by doing it every day, it never got out of hand.

Once I got through all of the issues, we were down to more like 600. A whole fourth of the issues shouldn't even have been open in the first place. Two weeks in is when the next big gain kicked in. Why two weeks? Well, two weeks is the grace period we decided before marking an issue as stale. Why two weeks? Well, that's kind of arbitrary, but two weeks feels like enough time for someone to respond if they're actively interested in getting an issue fixed. See, issues often need the help of the reporter to truly fix it, as there just isn't enough information in many bug reports to be able to reproduce and fix the problem.

So, after two weeks, I did one more thing each evening: I filtered by "least recently updated," and checked to see if any of those issues were stale. You just go back until they say "two weeks," and then, if you haven't heard from the reporter, mention that it's stale and give the issue a close. This is one of the other things I had to kind of let go of when working on a real project: closing an issue isn't forever. You can always re-open the issue later if it turns out you were wrong. So when trying to get a handle on 800 open issues, I defaulted to "guilty until proven innocent." Terminate issues with extreme prejudice. Leaving old, inconclusive issues doesn't help anyone. If it's a real bug that matters to someone, they'll come along and help reproduce it. If not, maybe someone else will later.

After a month or two, keeping on it, we got down to 450 or so issues. Members of the core team joked that they had to set up extra email filters from me, because they could tell exactly when I was doing triage. Slow and steady wins the race!

At this point, I knew enough about Rails to actually start writing some patches. And I happened to be familiar with basically every open bug. So it was easy to start picking some of them and try to reproduce them locally. So I'd do that, and then try to write a patch. If I couldn't, I'd at least upload my reproduction of the issue, and then leave a note on the Issue, pointing to my reproduction. That way, another team member could simply clone my repository and get to it. The only thing better than reproduction instructions are when those instructions say git clone.

But I managed to get a few patches in, and then a few more. Doing all of this janitorial work directly led the way towards attaining a commit bit on Rails. It was a long slog at first, but it just got easier the more I did it. A lot of work in open source is this way: it's really easy once you've done it a lot, but it is hard for newbies. I'm not yet sure how to tackle this problem…

I've since taken this approach on basically every repository I've worked on, and it's worked really well. But it only works if you keep at it: if you don't tend your garden, you'll get weeds. I haven't had as much time for Rails over the last few months, and it's back to 800 issues again. I'm not sure if these are real issues or not, as I've stopped tending. But without someone actively paying attention, it's only a matter of time before things get unseemly. If you're looking to help out an open source project, it's not a glamorous job, but all it takes is a little bit of work, and developing a habit. ■

Steve Klabnik is a Rails committer, Rust enthusiast, and works for Balanced Payments in San Francisco. He has authored "Designing Hypermedia APIs," "Rust for Rubyists," and "Rails 4 in Action." When he's not programming, he reads philosophy books and plays Android: Netrunner.

Illustration by Yana Dhyana.

HACK
ON YOUR
SEARCH ENGINE

and help change the future of search

duckduckhack.com

The Meaning of Life

By DEREK SIVERS

THERE'S A TRUE story about the student who showed up late to math class. He copied the problem that was already written on the board, assuming it was homework, and solved it that week. Only afterwards did he find out the teacher put it on the board as an example of an unsolvable problem.

This question — "What is the meaning of life?" — is the classic unsolvable problem. For thousands of years, people have been trying to figure it out. It's the punch line cliché of unanswerable questions.

But right now, let's be the naive ones that don't know it's considered unsolvable, and just figure out the meaning of life in under 20 minutes. OK?

LIFE IS ___________

What word do you think goes in that blank? Life is what? Any ideas?

Let's look at some of the different options that philosophers and smarties have said.

LIFE IS TIME

Some say life is time. Life is all about time. The definition of life is the time between when you're born and when you die. So the literal meaning of life is time.

So if life is time, the way to have a good life is to use time wisely.

How can you use time wisely? Five ways.

❶ Remember it's limited

If you find out tonight that you've only got one year left to live, you'll make the most of this next year. If you act like life is infinite, you won't.

To achieve great things, two things are needed: a plan, and not quite enough time.

Give yourself tight deadlines. Remember you could die at any time. Don't delay.

How can you use time wisely?

❷ Be mostly future-focused

Make most of your current actions serve your future self. Learn, practice, exercise, delay gratification, save and invest your money, and build towards your ideal future. People who do this are more successful and even happier.

But too much future focus leads to being a successful person on your 4th marriage, with no true friends. Too much future focus can take time away from important things that need you to be in the moment.

How can you use time wisely?

❸ Be somewhat present-focused

Sometimes, pull your head out of the future, and give your full attention to the present. Relationships, communication, and sex require this.

But too much present focus is hedonism: living only for immediate gratification with as much excitement and novelty as possible.

Too much present focus leads to an empty bank account and no impulse control.

Too much present focus robs you of the deeper happiness of delayed gratification, achieving long-term goals, and developing valuable expertise.

How can you use time wisely?

❹ Be somewhat past-focused

To remember your past is to live twice.

Keep your life in the context of the past, to see how far you've come.

Put aside time to re-interpret your past events, as a powerful reminder that you can re-interpret your present and future, too.

How can you use time wisely?

❺ Get in the zone

You know the feeling of flow — where you're focused on work that's not too easy and not too hard - where the work itself has clear goals and is its own reward.

People at the end of their life who claimed to be the happiest with their life were the ones who had spent the most time in this state of flow.

For a good life, pursue the work that puts you in this state, and avoid the things that pull you from this state.

Let's say life is time. What do you think? Pretty good argument?

Let's look at another perspective.

LIFE IS CHOICE

Some say life is choice. Life is all about choice. You make a hundred little choices a day, and a hundred big choices in your life. These choices change your entire life. Your life is created by your choices. Therefore life is choice.

So if life is choice, the way to have a good life is to make good choices.

How can you make good choices? Four ways.

❶ Let instinct trump logic

The different parts of your brain started developing at different periods in evolution. The oldest part of your brain, the one that's been evolving since we were fish, deals with instincts, fears, and gut feelings. The newest part of your brain, the one that's pretty uniquely human, deals with logic, language, and predictions.

This newest part is still in beta. A $5 calculator can beat it at math. But this oldest part was launched a billion years ago, and has been in production and development ever since.

Everything you observe and learn is first processed by your logical brain, but then the results are permanently stored as instincts, fears, and gut feelings. Your instincts and emotions hold the culmination of everything you've ever observed and learned. So you'll make better choices if you listen to your instincts, instead of relying too much on your $5 calculator beta brain.

How can you make good choices?

❷ Stop at good enough

You now have more options than ever. You try to choose the best option, the best career, the best school, and the best boyfriend/girlfriend/partner/spouse.

But thinking this way makes you feel worse about the choices you've made. You're more aware than ever of all the options you didn't choose, and the benefits of each.

So don't seek the absolute best. Stop when you find an option that is good enough. You'll make an equally good choice, but more importantly, you'll feel much better about it. Happiness counts.

How can you make good choices?

❸ Set limits

Every choice you have to make causes a little bit of pain. Having choice in life is good, but having more choice is not always better.

You're happier when you let other people make some choices for you. If you're very sick, you want your doctor to choose what's best, not say, "There are dozens of good options. What do you want to do?" This is the appeal of religion. It gives you rules. It makes many of the choices for you.

So set limits to your choices in life. Cut off some options. Give yourself rules.

How can you make good choices?

❹ Choose important not urgent

You know the difference between what's long-term important versus short-term urgent.

What's urgent are emails, texts, tweets, calls, and news.

What's important is spending a thousand hours to learn a new skill that will really help you in your life or work. What's important is giving your full undistracted attention to the important people in your life. What's important is taking time to get exercise, or to collect and share what you've learned.

But none of these things will ever be urgent.

So you have to ignore the tempting cries of the urgent, and deliberately choose what you know is important.

So life is choice? What do you think? Pretty good argument? Let's try another.

LIFE IS MEMORY

Some say life is memory. The future doesn't exist. It's something we imagine. The present is gone in a millisecond, so everything we experience in life is a memory. You could live a long life, but without a lot of memories, you only experience a short life. If you don't remember your life, it's like it never happened. So life is memory.

So if life is memory, the way to have a good life is to make more memories.

How can you make memories?

Change routines. Break monotony. Move. Make a major change whenever you can. These are your chronological landmarks. These are the hooks where you'll hang your memories.

Document it. Blog it. Not in a company's walled garden, but in a format you can archive and look through in 50 years, or your grand-kids can look through in 100 years. Keep a private blog for your future self, and tell the tales of where you've been, what you did, and the quirky people you've met along the way. You'll be surprised how much you forget if you don't record it.

Socrates said the unexamined life is not worth living. What about the forgotten life?

So life is memory? What do you think? Want to do another?

LIFE IS LEARNING

Both my smart friends and my spiritual friends insist that the meaning life is learning - that the reason you're here is to learn. Not just for your own sake, but for everyone alive, and future generations, the meaning of your life is to learn.

So if life is learning, the way to have a good life is to learn a lot.

How can you learn a lot?

Instead of talking about learning techniques, let's talk about getting the right mindset, so you can learn more than you realize.

You've probably heard about the Fixed mindset and the Growth mindset.

The Fixed mindset says, "I am good at this" or "I am bad at this". This starts in childhood when your parents say, "You're so good at math!" You think, "I'm good at math!" But then when you do poorly on one test, you think, "They were wrong. I'm not good at math." Most people think this way. You can hear it when they say, "She's a great singer" or "I'm just no good at dancing."

The Growth mindset says, "Anyone can be good at anything. Skill comes only from practice."

Two impossibly hard tests were given to hundreds of children. After the first test, all of the students were praised, but half of the students were privately told these 6 words: "You must be good at this." The other half were privately told these 6 words: "You must have worked really hard."

When they were given the second test, the students who were told, "You must be good at this", did 20% worse on the 2nd test. Those 6 words encouraged a fixed mindset that made them feel there was no point in trying. You either are or you aren't.

The students who were told "You must have worked really hard," did 30% better on the 2nd test. Those 6 words encouraged a growth mindset that made them feel that working harder made all the difference.

So that's a +-50% difference in performance because of 6 quick words by one teacher.

Multiply that by all the people in your life, all the days you hear feedback, and all the things you tell yourself, and you can see how this simple difference in mindset can make or break a life of learning.

Parents, pay attention to this. You may be harming your kids when you tell them they're good at things.

Successful people, pay attention to this. You may be harming yourself if you believe the praise that people give you. People tell you you're great at what you do, never just that you must have worked hard.

So... life is learning? What do you think?

Something else?

- Should we look at the Buddhist idea that life is SUFFERING? Nah, that's no fun.
- Life is LOVE? Too ambiguous.
- Life is NOTHING BUT REPLICATING DNA? Too accurate.

Let's change the subject.

Chinese

A few years ago, I started learning Chinese. I'm fascinated with the writing. I'm trying to memorize how to write these characters.

Chinese characters look complicated, but they're mostly made up of smaller simpler characters, the way that English words are made up of Latin roots and such. So you can remember the meaning of each character by knowing the meaning of its ingredients. For example:

语 *language = words* 讠 *+ five* 五*+ mouth* 口

So… Language is words that at least five mouths speak? Brilliant!

谢 *thank you = words* 讠 *+ body* 身 *+ inch* 寸

Hmmm... This one is not so obvious. Maybe the idea is that when you say thanks, you speak words that give a body an inch of respectful space? That's interesting.

名 *name = evening* 夕 *+ mouth* 口

So your real name is what's spoken by a mouth in the evening? That's kind of romantic.

I get so curious about the historical or cultural meaning behind each one.

Let's change the subject.

Talking Heads

Talking Heads were a great band from the late-70s to mid-80s. Their lyrics were really evocative and mysterious. They made you wonder what they were really about.

Then I read an interview with the Talking Heads where they said that many of their lyrics were just random. They would write evocative phrases onto little pieces of paper, then throw them into a bowl, and shuffle them up. Then they'd pull them out, and put them into the song in that order. They did this because they liked how the listener creates meaning that wasn't intended.

We assume that if someone writes a song, then sings it on stage into a microphone, that it must have meaning to them.

But nope. It was just random. Any meaning you think it contains was put there by you, the listener, not the writer. Like a Rorschach test.

Back to Chinese.

I got so curious about the historical meaning of these Chinese characters that I got a Chinese etymological dictionary that tells the full history behind every one.

I looked up the examples I gave here, and found out those characters were just phonetic! Those composite character bits were NOT chosen for their meaning at all, just their sound!

So it seems I've just been putting the meanings into them, myself. They actually had no meaning at all!

It blew my mind. I had been memorizing hundreds of characters for months, reading all kinds of meaning into the ingredients of each one.

After recovering from that, I thought: How many other things in life really have no meaning? What else have I been putting my own meaning into, thinking it was true?

Wired

I know that we're wired to do it. I know we survived on the savannah for eons because we evolved to look for patterns. Our ancestors are the ones who noticed the patterns of the tiger stripes or the lion face in the grass.

A moth is so deeply wired to fly towards the light that it may never accept that your light bulb is not the moon.

We are so deeply wired to find patterns that we may never accept that many things are just random.

We should have the same sympathy for our faulty wiring as we do for the moth. Evolution taught us to do this thing, but didn't teach us to stop.

Give us some dots and a line, and we'll see a face. Burn some toast and we'll find Elvis in it.

A carrot from my garden looks like Jesus. What does it mean?

A black cat crossed my path as I walked under a ladder on Friday the 13th. What does it mean?

An old friend calls just a minute after I was thinking about them. What does it mean?

What does it mean that you went to a prestigious well-known school? What does it mean that you didn't?

What does it mean that your good friend died? What does it mean that you're tall?

What does it mean that you have a lot of followers online? What does it mean that you don't?

What does it mean that you're female? What does it mean that you're male?

What does it mean that you're an entrepreneur? What does it mean that you're not?

What does it mean that all of your previous attempts at something have failed?

Nothing! Nothing at all.

Nothing has inherent meaning. Everything is only what it is and that's it.

So let's get back to our original question and wrap this up.

Life Is ______

What is the meaning of life? LIFE IS ______

TIME?

CHOICE?

MEMORY?

LEARNING?

SUFFERING?

LOVE?

REPLICATING DNA?

You can tell by the variety of answers that they are just projected meanings.

You can choose to project one of these meanings onto your life, if it makes you feel good, or improves your current actions.

But you know the real answer is clear and obvious now.

LIFE IS (just) LIFE. IT DOESN'T MEAN ANYTHING.

Erase any meaning you put into past events. Erase any meaning that's holding you back. Erase those times where people said that this means that. None of it is real.

Life has no inherent meaning. Nothing has inherent meaning.

Life is a blank slate.

You're free to project any meaning that serves you.

You're free to do with it, anything you want.

Thank you. ■

Originally a professional musician and circus clown, Derek Sivers created CD Baby in 1998. He is a frequent speaker at the TED Conference, with over 5 million views of his talks. His new company is Wood Egg, publishing annual guides to 16 countries in Asia.

Illustration by Jaime G. Wong.

BETTER SENDING, BETTER INSIGHTS

ANALYZE, REACT, ENGAGE

NEW

REST API

Parse API

Real-Time event API

features coming soon!

mailjet™

The Most Important Tech Job That Doesn’t Actually Exist

By ALEX KRUPP

Yesterday I asked a prominent VC a question:

“Why is it that, despite the fact that so many successful startup ideas come from academic research, on the investment side there doesn’t seem to be anyone vetting companies on the basis of whether or not what they’re doing is consistent with the relevant research and best practices from academia?”

His response was that, unlike with startups in other sectors (e.g. biotech, cleantech, etc.), most tech startups don’t come out of academia, but rather are created to fill an unmet need in the marketplace. And that neither he nor many of his colleagues spent much time talking with academics for this reason.

This seems to be the standard thinking across the industry right now. But despite having nothing but respect for this investor, I think the party line here is unequivocally wrong.

Let’s start with the notion that most tech startups don’t come out of academia. While this may be true if you consider only the one-sentence pitch, once you look at the actual design and implementation choices these startups are making there is typically quite a lot to work with.

For example, there is a startup I recently looked at that works to match mentors with mentees. Though one might not be aware of it, there is actually a wealth of research into best practices:

- What factors should be used when matching mentors with mentees?
- How should the relationship between the mentor and mentee be structured?
- What kind of training, if any, should be given to the participants?

That’s not to say that a startup that’s doing something outside the research, or even contraindicated by the research, is in any way suspect. But it does raise some questions: Does the startup have a good reason for what they’re doing? Are they aware of the relevant research? Is there something they know that we don’t?

If the entrepreneurs have good answers to these questions then it’s all the more reason to take them seriously. But if they don’t then this should raise a few red flags. And it’s not only niche startups in wonky areas where this is an issue.

For example, I rarely post to Facebook anymore, but people who follow me can still get a good idea of what I’m up to. Why? Because Facebook leverages the idea of behavioral residue to figure out what I’m doing (and let my friends know) without me having to explicitly post updates. It does this by using both interior behavioral residue, e.g. what I’m reading and clicking on within the site, and exterior behavioral residue, e.g. photos of me taken outside of Facebook.

To understand why leveraging behavioral residue is so important for social networks, consider that of people who visit the typical website only about 10% will make an account. Of those who do, about 10% will make at least one content contribution, and of those about 10% will become core contributors. So if you consider your typical user with a couple hundred friends, this

“Have we really become so myopic as to place zero value on knowing whether or not a startup is congruent or contraindicated by the last 80+ years of research?”

translates into seeing content from only a tiny handful of other people on a regular basis.

In contrast with Facebook, one of the reason why FourSquare has yet to succeed is due to significant problems with their initial design decisions:

- The only content on the site comes from users who manually check into locations and post updates. This means that of my 150 or so friends, I'm only seeing what one or two of them are actually doing, so what's the value?
- The heavy use of extrinsic motivation (e.g. badges) has been shown time and again that extrinsic motivation undermines intrinsic motivation.

The latter especially is a good example of why investing on traction alone is problematic: many startups that leverage extrinsic rewards are able to get a good amount of initial traction, but almost none of them are able to retain users or cross the chasm into the mainstream. Why isn't it anyone's job to know this, even though the research is readily available for any who wants to read it? And why is it so hard to go to any major startup event without seeing VCs showering money on these sorts of startups that are so contraindicated by the research that they have almost no realistic chance of succeeding?

This same critique of investors applies equally to the startups themselves. You probably wouldn't hire an attorney who wasn't willing to familiarize himself with the relevant case law before going to court. So why is it that the vast majority of people hired as community managers and growth marketers have never read Robert Kraut? And the vast majority of people hired to create mobile apps have never heard of Mizuko Ito?

A lot of people associate the word design with fonts, colors, and graphics, but what the word actually means is fate — in the most existential sense of the word. That is, good design literally makes it inevitable that the user will take certain actions and have certain subjective experiences. While good UX and graphic design are essential, they're only valuable to the extent that the person doing them knows how to create an authentic connection with the users and elicit specific emotional and social outcomes. So why are we hiring designers mainly on their Photoshop skills and maybe knowing a few tricks for optimizing conversions on landing pages? What a waste.

Of all the social sciences, the following seem to be disproportionately valuable in terms of creating and evaluating startups:

- Psychology / Social Psychology
- Internet Psychology / Computer Mediated Communication
- Cognitive Development / Early Childhood Education
- Organizational Behavior
- Sociology
- Education Research
- Behavioral Economics

And yet not only is no one hiring for this, but having expertise in these areas likely won't even get you so much as a nominal bonus. I realize that traction and team will always be the two biggest factors in determining which startups get funded, but have we really become so myopic as to place zero value on knowing whether or not a startup is congruent or contraindicated by the last 80+ years of research?

So should you invest in (or work for) the startup that sends text messages to people reminding them to take their medicine? How about the one that lets you hire temp laborers using cell phones? Or the app for club owners that purports to increase the amount of money spent on drinks? In each of these cases there is a wealth of relevant literature that can be used to help figure out whether or not the founders have done their homework and how likely they are to succeed. And it seems like if you don't have someone who's willing to invest a few hours to read the literature then you're playing with a significant handicap.

Investors often wait months before investing in order to let a little more information surface, during which time the valuation can (and often does) increase by literally millions. Given that the cost of doing the extra research for each deal would be nominal in the grand scheme of things, and given the fact that this research can benefit not only the investors but also the portfolio companies themselves, does it really make sense to be so confident that there's nothing of value here?

What makes the web special is that it's not just a technology or a place, but a set of values. That's what we were all originally so excited about. But as startups become more and more prosaic, these values are largely becoming lost. As Howard Rheingold once said, "The "killer app" of tomorrow won't be software or hardware devices, but the social practices they make possible." You can't step in the same river twice, but I think there's something to be said for startups that make possible truly novel and valuable social practices, and for creating a larger ecosystem that enables them. ■

Alex studied marketing as part of Seth Godin's six-month alternative MBA program, and was the first single non-technical founder ever accepted into Y Combinator. He then taught himself to code, and now specializes in frontend web development and Big Data analytics.

How To Get Business Ideas: Remove Steps

By NATHAN KONTNY

I see so many aspiring entrepreneurs stressed out hoping to find some spark of a business idea, but the common complaint is:

Everything good has already been done.

I used to feel like that too, but it doesn't have to be hard. It can be as simple as:

1. Find a job people have.
2. List out every step people take to complete that job.
3. Remove as many steps as you can.

Look at measuring cups.

People have been using cups to measure things for thousands of years. And standardized measuring cups have been around since Fannie Farmer invented them in 1896.

You can't possibly come up with a new idea for a measuring cup. Right?

But then someone does. A guy named Steve Hoeting was trying to come up with a new recipe for brownies, and realized he spent too many steps in the process reading the level of stuff on the side of his measuring cup. Could he shave steps from that process, by putting the ruler at an angle to read it from above? A couple prototypes later, he had a crude design, which went onto become the wild success of the OXO measuring cup.

What about S'mores?

You can hardly come up with a simpler dessert: graham crackers + chocolate + marshmallow, a recipe that's been around for at least 90 years when it first appeared recorded in a Girl Scouts' book in 1927.

How can you simplify this into a business idea?

Well, three college kids had to come up with a last-minute idea for an entrepreneurial class project a day before it was due. Realizing the chocolate step of S'mores was a small nuisance and that the chocolate is hard to get consistently melted in your recipe, they created Stuff n' Mallows - marshmallows with the chocolate already stuffed inside. [stuffnmallows.com]

They shaved off the steps of buying chocolate bars, and adding it to your S'more. Now, just heat the Stuff n' Mallows for your S'more, and you've already accomplished the chocolate part.

Simple. I bet someone reading this even said to themselves, "That's a trivial idea. It's dumb."

Now, just a year later, their product is in over 40 stores across the United States, and they're winning innovation awards from one of the oldest trade associations in the world, the National Confectioners Association.

If Steve Hoeting and these college kids successfully simplified what already appeared so simple and had been around for hundreds of years, imagine how many things you can find to improve.

****

Want a good place to start looking for jobs to simplify? One of my favorite overlooked resources for business ideas is For Dummies books. [dummies.com]

Make everything easier in your life with step-by-step instruction.
– For Dummies website description

For Dummies has already done a ton of the work laying out job after job and the steps involved. Grab a Dummies book and use it as a starting point to find steps to try and remove. ■

Nathan Kontny Founder of two YC companies: Inkling & Citypsoh. Engineer for President Obama's re-election campaign. Now working on Draft.

Creating a Bare Bones Bootloader

By JOE SAVAGE

Upon hearing the word "bootloader", many people cower in fear. It seems like a scary low-level thing that's very easy to mess up. To dispel some of this fear, I'm going to walk through making a super simple bootloader/OS, assuming only a basic level of familiarity with assembly language and how computers work.

The Plan

The first thing to understand or research if we're looking to write something that runs when the computer starts up, is exactly what the boot process is. Typically computers will first boot into the BIOS (Basic Input/Output System) firmware, which performs some tests (often while showing a logo of sorts onscreen) and then boots into an operating system. There's also a new standard, UEFI, coming into play as an alternative to BIOS, but we aren't going to talk about that here.

```
                 F2 = System Setup
             F10 = System Services
      Entering BIOS Boot Manager
                    F12 = PXE Boot

         www.dell.com

         PowerEdge T710
       BIOS Revision 6.1.0
```

A typical Dell BIOS boot screen

More precisely, the standard BIOS boot process is as follows:

- The computer boots into the BIOS.
- The BIOS performs a Power-on Self-test (POST).
- Using information from POST and BIOS configuration details, possible boot devices are selected.
- For disk drives (or similar devices), the first 512 bytes of the disk - termed the "boot sector" — is considered for booting. If the sector can be read and the standard boot signature is present in the last two bytes (0x55 0xAA), the device is considered bootable. Otherwise, the next device in the list of candidates is checked.
- Assuming the disk drive is bootable, the 512 byte boot sector is copied to address 0x007C00 at which point the BIOS transfers control to the loaded sector through a jump instruction to 0x007C00.

When booting from partitioned devices, the boot sector of the drive will commonly be a Master Boot Record (MBR), in which case the boot sector generally consists of a bootstrap which identifies a certain partition on the drive — usually via a partition table (which may also reside in the first sector of the drive) — and proceeds to load the boot sector of that particular partition, sometimes called a Volume Boot Record (VBR). The structure of a number of modern MBR boot sectors is as follows:

```
0x7C00: Bootstrap
0x????: Disk Information
0x????: Partition Table
0x7DFE: Boot Signature (0x55
0xAA)
0x7E00:
```

In our case, however, dealing with disk partitions is a bit fancy and overcomplicated. Since complete control is handed to us when the BIOS jumps to our boot sector in RAM, we can just work with an unpartitioned disk and go straight into our Volume Boot Record in the first 512 bytes (the boot sector) of our disk image. Technically that means we're not really creating a bootloader, more of a really really really small operating system. Realistically though, what we're creating is exactly like a bootloader - just without the loading of and jumping into some other portion of code on the disk. And so I'll continue referring to our bootloader as such. Thus the structure of our boot sector will be the code we want to execute, followed by the standard boot signature.

So the plan is that the BIOS will copy our boot sector to 0x007C00 and transfer control to this point, and then we... do something. Since we're just creating a bare bones bootloader, let's make our goal to output some text. Note that in the x86 BIOS process, the BIOS will transfer control while the processor is still operating in real-mode. This has its advantages and disadvantages, but to keep things simple here we'll just keep things in real-mode (keeping it real, bro) for the entirety of our bootloader.

With all this research and theory out of the way, let's actually write some code!

The Code

We're going to be using NASM here, because it happens to be the x86 assembler I'm most familiar with. Since x86 real-mode defaults to using 16-bit instructions, we want the assembler to output instructions as such. We can still use 32-bit registers if we so wish, but as we're operating in 16-bit mode we need to prefix any instructions that do this with the "Operand Size Override Prefix" (0x66). All of this is accomplished by using the "BITS" NASM directive - in this case: `BITS 16`.

Addresses in x86 assembly are determined from the segment address plus an offset. As both real and protected mode of the x86 processor use 16-bit segment registers, we should ensure that our segments - in particular the stack segment (`ss`) and data segment (`ds`) — refer to sensible 64K regions. Since real-mode limits absolute addresses/offsets to 16-bits in length, segments can help us access more than one 64K portion of memory.

Regardless of whether we'll make explicit use of the stack segment in this example, it seems good practice to set up a stack — especially given that some instructions make implicit use of segments (e.g. `push`, which uses the Stack Segment with a Stack Pointer offset), and it would be irresponsible not to define a stack if the bootloader were to be expanded beyond our basis of outputting some text.

In this case, I'm going to structure our bootloader such that it has a 4K stack just after the location of the boot sector in memory. Remember that the boot sector is loaded into 0x7C00 and is 0x200 (512) bytes long, and so we want our stack to reside just after this. Due to the way that segments work in x86 assembly (they refer to 64K chunks of memory, not specific locations, as noted previously), we actually want to divide both of those values by 16 before assigning the final location to the stack segment — `Effective Address = Segment*16 + Offset`. So the code to set up our stack segment is as follows:

```
mov ax, 07C0h   ; Set 'ax' equal to the location
of this bootloader divided by 16
add ax, 20h     ; Skip over the size of the
bootloader divided by 16
mov ss, ax      ; Set 'ss' to this location (the
beginning of our stack region)
```

Since space is often precious in a mere 512 bytes, assuming we don't get the assembler to optimise this code it might be more appropriate to save a few bytes by doing the `add` instruction manually, and simply changing the `mov` to `mov ax, 07E0h`. I'll keep things as they are for now though since keeping the add separate offers more clarity to what we're actually doing.

As noted previously, operations involving the stack involve the stack pointer offset from the stack segment: `SS:SP`. As the stack pointer grows downwards in memory address (i.e. towards 0x00) by default, we want to define the stack pointer to point to the bottom of the stack which is 4K away from our stack segment: `mov sp, 4096`. At this point our bootloader memory footprint should look like the following:

```
        |^^^^^^^^^^^^^^^^^|
0x7C00: |-----------------|
        |   Boot Sector   |
0x7E00: |-----------------| <-SS
        |                 |
        |                 |
        |   Stack Space   |
        |                 |
        |                 |
0x8E00: |-----------------| <-SS:SP
        |                 | (grows upwards)
         ˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇ
```

With regards to the data segment, we can just set the segment to the start of our bootloader code. Remember that the real purpose of segments in real-mode is to allow us to access more than a single 64K of data (in real-mode, there is a little over 1MB of addressable memory — `0xFFFF0 + 0xFFFF` bytes to be exact), and any static or global data in our little bootloader will most definitely live within our little 512 codebase. So with the `ds` pointed towards the start of our bootloader code, we can definitely access all of the data we might need to in Segment:Offset form. Thus the code to initialise our data segment:

```
mov ax, 07C0h   ; Set 'ax' equal to the location
of this bootloader divided by 16
mov ds, ax      ; Set 'ds' to the this location
```

From here, we can actually go ahead and do something interesting before we tie up. As we're in real-mode and haven't specified any custom interrupt handling, we can make use of some standard BIOS interrupts to accomplish tasks. Looking at the list on Wikipedia [hn.my/biosi], the "Video Services" interrupt 10h seems interesting. Upon closer inspection, the interrupt when `AH` is 0Eh seems particularly interesting as it provides teletype output. Looking at the particular parameters for this interrupt it seems that `AL` should contain the character to be printed to the screen, `BH` should contain the page number, and `BL` should contain the colour (only in graphic mode). Thus writing assembly to print a string using `int 10h` from here is trivial.

Perhaps in the context of our bootloader it makes sense to write a proper routine for writing strings which expects a single parameter — the address of a null-terminated string — through the `si` (Source Index) register. The next portion in our bootloader code will hence be to call this routine (which we'll name "print"), and stop execution such that our string remains on the screen and nothing else happens:

```
mov si, message ; Put address of the null-termi-
nated string to output into 'si'
call print      ; Call our string-printing routine
cli             ; Clear the Interrupt Flag (dis-
able external interrupts)
hlt             ; Halt the CPU (until the next
external interrupt)
message db 'This was outputted by a basic boot-
loader!', 0
```

Now we just need to define our print function to make use of the standard BIOS interrupt 10h previously described:

```
; Routine for outputting string in 'si' register
to screen
print:
       mov ah, 0Eh     ; Specify 'int 10h'
'teletype output' function
                       ; [AL = Character, BH =
Page Number, BL = Colour (in graphics mode)]
.printchar:
       lodsb           ; Load byte at address SI
into AL, and increment SI
       cmp al, 0
       je .done        ; If the character is
zero (NUL), stop writing the string
       int 10h         ; Otherwise, print the
character via 'int 10h'
       jmp .printchar  ; Repeat for the next
character
.done:
       ret
```

Now that we've finished writing the main bulk of the code, we just need to pad the code out with 0s to byte 510, and then use bytes 511 and 512 for the standard byte signature (as previously mentioned, 0x55 and 0xAA — which we'll specify backwards in our code due to little endian byte order). Using the NASM `times` directive for the padding, we can do all of this with the following:

```
; Pad to 510 bytes (boot sector size minus 2)
with 0s, and finish with the two-byte standard
boot signature
times 510-($-$$) db 0
dw 0xAA55           ; => 0x55 0xAA (little
endian byte order)
```

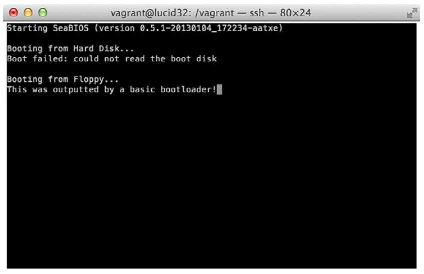
And we're done! We finished writing our little bootloader — the final product of which, "`bootloader.asm`", is broadly as follows:

```
       BITS 16

start:
       ; Set up 4K stack after this bootloader
       ; [Remember: Effective Address = Segment*16 + Offset]
       mov ax, 07C0h    ; Set 'ax' equal to the location of this bootloader divided by 16
       add ax, 20h      ; Skip over the size of the bootloader divided by 16
       mov ss, ax       ; Set 'ss' to this location (the beginning of our stack region)
       mov sp, 4096     ; Set 'ss:sp' to the top of our 4K stack

       ; Set data segment to where we're loaded so we can implicitly access all 64K from here
       mov ax, 07C0h    ; Set 'ax' equal to the location of this bootloader divided by 16
       mov ds, ax       ; Set 'ds' to the this location

       ; Print our message and stop execution
       mov si, message ; Put address of the null-terminated string to output into 'si'
       call print       ; Call our string-printing routine
       cli              ; Clear the Interrupt Flag (disable external interrupts)
       hlt              ; Halt the CPU (until the next external interrupt)

data:
       message db 'This was outputted by a basic bootloader!', 0

; Routine for outputting string in 'si' register to screen
print:
       mov ah, 0Eh      ; Specify 'int 10h' 'teletype output' function
                        ; [AL = Character, BH = Page Number, BL = Colour (in graphics mode)]
.printchar:
       lodsb            ; Load byte at address SI into AL, and increment SI
       cmp al, 0
       je .done         ; If the character is zero (NUL), stop writing the string
       int 10h          ; Otherwise, print the character via 'int 10h'
       jmp .printchar   ; Repeat for the next character
.done:
       ret

; Pad to 510 bytes (boot sector size minus 2) with 0s, and finish with the two-byte standard boot
signature
times 510-($-$$) db 0
dw 0xAA55              ; => 0x55 0xAA (little endian byte order)
```

This code, along with the bits I use to compile and test the bootloader can be found at the associated GitHub repo, here. [hn.my/bootcode]

Testing The Code

Now we've actually finished writing our bootloader, we need some way to test it. I use Vagrant with a "lucid32" box for all my OS testing, but in this case you might be able to get by in your local environment. You can first compile the NASM via something along the lines of `nasm -f bin -o bootloader.bin bootloader.asm`, and you can then use a utility such as "dd" to copy the data to a floppy disk image file (or to a real floppy disk!): `dd conv=notrunc bs=512 count=1 if=bootloader.bin of=bootloader.flp`. Using QEMU, you could then boot the floppy disk image in a virtual machine with something like qemu `-fda bootloader.flp -curses`. And voilà, it works.



```
Starting SeaBIOS (version 0.5.1-20130104_172234-aatxe)

Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
This was outputted by a basic bootloader!
```

Our bootloader in action!

Now, I appreciate that this seems like an awful lot of work for a single line of text. There certainly is a fair bit of theory behind this, but at the end of the day the resulting assembly is actually relatively simple, and we've created a disk sector that a real x86 BIOS could boot into. If you wanted to actually test this "properly", you could apply the disk image in the correct manner to a USB stick, floppy disk, or CD, and actually boot your computer into our little bootloader. Into our little world from which we have full CPU control and a number of powerful extensions could easily be added. If you're looking for some cool extensions you could add right now (still in real-mode), just take a look at some of the other standard BIOS interrupts, or alternatively you could start looking at other more complex bootloader goals.

Generally if you're interested in writing the core of an operating system, I'd actually suggest that you start out by using GRUB, which takes care of the process of taking control and switching into protected mode for you in a very standardised way (this is a pretty nice tutorial for getting started writing a very basic kernel which boots from GRUB). There's something interesting to me, however, about being able to write the code to actually take control — from the very first stage — all on our own, and that's exactly what we've achieved here. ■

Joe Savage is a computer science student and software developer from England, interested in a wide variety of areas ranging from reverse engineering to game development. He writes about technical topics on his blog at *reinterpretcast.com*

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

Erlang and Code Style

Musings On Mostly Defensive Programming Styles

By JESPER LOUIS ANDERSEN

CORRECT ERLANG USAGE mandates that you do not write any kind of defensive code. This is called *intentional programming*. You write code for the intentional control flow path which you expect the code to take. And you don't write any code for the paths which you think are not possible. Furthermore, you don't write code for data flow which was not the intention of the program.

It is an effect, silly

If an Erlang program goes wrong, it crashes. Say we are opening a file. We can *guard* the file open call like so:

```
{ok, Fd} = file:open(Filename, [raw, binary,
read, read_ahead]),
```

What happens if the file doesn't exist? Well, the process crashes. But note we did not have to write any code for that path. The default in Erlang is to crash when a match isn't valid. We get a badmatch error with a reason as to why we could not open the file.

A process crashing is not a problem. The *program* is still operating, and supervision — an important fault-tolerance concept in Erlang — will make sure that we try again in a little while. Say we have introduced a race condition on the file open, by accident. If it happens rarely, the program would still run, even if the file open fails from time to time.

You will often see code that looks like:

```
ok = foo(...),
ok = bar(...),
ok = ...
```

which then asserts that each of these calls went well, making sure code crashes if the control and data flow is not what is expected.

Notice the complete lack of error handling. We don't write

```
case foo(...) of
    ok -> case bar(...) of ... end;
    {error, Reason} -> throw({error, Reason})
end,
```

Nor do we fall into the trap of the Go programming language and write:

```
res, err := foo(...)
if err != nil {
    panic(...)
}
res2, err := bar(...)
if err != nil {
    panic(...)
}
```

because this is also plain silly, tedious, and cumbersome to write.

The key is that we have a crash-effect in the Erlang interpreter which we can invoke where the *default* is to crash the process if something goes wrong. And have another process clean up. Good Erlang code abuses this fact as much as possible.

Intentional?

Note the word intentional. In some cases, we *do* expect calls to fail. So we just handle it like everyone else would, but since we can emulate sum-types in Erlang, we can do better than languages with no concept of a sum-type:

```
case file:open(Filename, [raw, read, binary]) of
    {ok, Fd} -> ...;
    {error, enoent} -> ...
end,
```

Here we have written down the intention that the file might not exist. However:

- We *only* worry about nonexistence.
- We crash on `eaccess`, which means an access error due to permissions.
- Likewise for `eisdir, enotdir, enospc`.

Why?

Leaner code, that's why.

We can skip lots of defensive code which often more than halves the code size of projects. There is much less code to maintain so when we refactor, we need to manipulate less code as well.

Our code is not littered with things having nothing to do with the "normal" code flow. This makes it far easier to read code and determine what is going on.

Erlang process crashes give lots of information when something dies. For a proper OTP process, we get the State of the process before it died and what message was sent to it that triggered the crash. A dump of this is enough in about 50% of all cases, and you can reproduce the error just by looking at the crash dump. In effect, this eliminates a lot of silly logging code.

Data flow defensive programming

Another common way of messing up Erlang programs is to mangle incoming data through pattern matching. Stuff like the following:

```
convert(I) when is_integer(I) -> I;
convert(F) when is_float(F) -> round(F);
convert(L) when is_list(L) ->
list_to_integer(L).
```

The function will convert "anything" to an integer. Then you proceed to use it:

```
process(Anything) -> I = convert(Anything),
...I...
```

The problem here is not with the **process** function, but with the call-sites of the **process** function. Each call-site has a different opinion on what data is being passed in this code. This leads to a situation where every subsystem handles conversions like these.

There are several disguises of this anti-pattern. Here is another smell:

```
convert({X, Y}) -> {X, Y};
convert(B) when is_binary(B) ->
    [X, Y] = binary:split(B, <<"-">>),
    {X, Y}.
```

This is stringified programming where all data are pushed into a string and then manually deconstructed at each caller. It leads to a lot of ugly code with little provision for extension later.

Rather than trying to handle different types, enforce the invariant early on the API:

```
process(I) when is_integer(I) -> ...
```

And then *never* test for correctness inside your subsystem. The dialyzer is good at inferring the use of `I` as an integer. Littering your code with `is_integer` tests is not going to buy you anything. If something is wrong in your subsystem, the code will crash, and you can go handle the error.

There is something to be said about static typing here, which will force you out of this unityped world very easily. In a statically typed language, I could still obtain the same thing, but then I would have to define something along the lines of (* Standard ML code follows *)

```
    datatype anything = INT of int
                      | STRING of string
                      | REAL of real
```

and so on. This quickly becomes hard to write pattern matches for, so hence people only define the *anything* type if they really need it.

The scourge of undefined

Another important smell is that of the *undefined* value. The story here is that undefined is often used to program an Option/Maybe monad. That is, we have the type:

```
-type option(A) :: undefined | {value, A}.
```

It is straightforward to define reflection/reification into an exception-effect for these. Jakob Sievers stdlib2 library already does this, as well as define the monadic helper called do (though the monad is of the Error-type rather than Option).

But I've seen:

```
-spec do_x(X) -> ty() | undefined
  when X :: undefined | integer().
do_x(undefined) -> undefined;
do_x(I) -> ...I....
```

which leads to complicated code. You need to be 100% in control of what values can fail and what values cannot. Constructions like the above silently pass undefined on. This has its uses — but be wary when you see code like this. The *undefined* value is essentially a *NULL*. And those were C.A.R Hoare's billion dollar mistake.

The problem is that the above code is *nullable*. The default in Erlang is that you never have NULL-like values. Introducing them again should be used sparingly. You will have to think long and hard because once a value is nullable, it is up to you to check this all the time. This tends to make code convoluted and complicated. It is better to test such things up front and then leave it out of the main parts of the code base as much as possible.

"Open" data representations

Whenever you have a data structure, there is a set of modules which knows about and operates on that data structure. If there is only a single module, you can emulate a common pattern from Standard ML or OCaml where the concrete data structure representation is abstract for most of the program and only a single module can operate on the abstract type.

This is not entirely true in Erlang, where anyone can introspect any data. But keeping the illusion is handy for maintainability.

The more modules that can manipulate a data structure, the harder it is to alter that data structure. Consider this when putting a record in a header file. There are two levels of possible creeping insanity:

- You put the record definition in a header file in `src`. In this case only the application itself can see the records, so they don't leak out.
- You put the record definition in a header file in `include`. In this case the record can leak out of the application and often will.

A good example is the HTTP server `cowboy` where its request object is manipulated through the `cowboy_req` module. This means the internal representation can change while keeping the rest of the world stable on the module API.

There are cases where it makes sense to export records. But think before doing so. If a record is manipulated by several modules, chances are that you can win a lot by re-thinking the structure of the program.

The values "true" and "false" are of type atom()

As a final little nod, I see too much code looking like

```
  f(X, Y, true, false, true, true),
```

Which is hard to read. Since this is Erlang, you can just use a better name for the true and false values. Just pick an atom which makes sense and then produce that atom. It also has the advantage to catch more bugs early on if arguments get swapped by accident. Also note you can bind information to the result, by passing tuples. There is much to be said about the concept of *boolean blindness*, which in typical programs means to rely too much on `boolean()` values. The problem is that if you get a true say, you don't know why it was true. You want evidence as to its truth. And this can be had by passing this evidence in a tuple. As an example, we can have a function like this:

```
case api:resource_exists(ID) of
  true -> Resource = api:fetch_resource(ID), ...;
  false -> ...
end.
```

But we could also write it in a more direct style:

```
case api:fetch_resource(ID) of
    {ok, Resource} -> ...;
    not_found -> ...
end.
```

which in the long run is less error prone. We can't by accident call the `fetch_resource` call, and if we look up the resource, we also get hold of the evidence of what the resource is. If we don't really want to use the resource, we can just throw it away.

Closing remarks

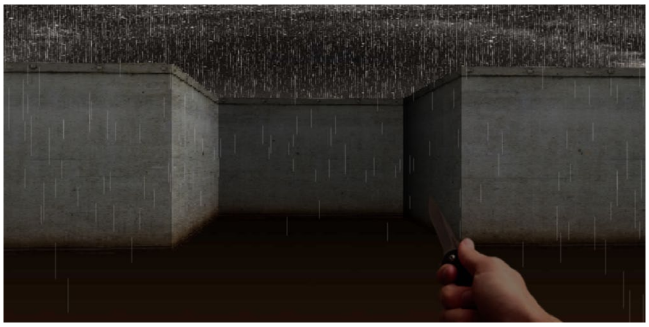
Rules of thumb exists to be broken. So once in a while they must be broken. However, I hope you learned something or had to stop and reflect on something if you happened to get here. ■

Jesper Louis Andersen is a functional programmer, mostly working in Erlang, Standard ML and OCaml. He has a keen interest in concurrent and distributed programming and their application to old and new problems.

A First-Person Engine in 265 Lines

By HUNTER LOFTIS

TODAY, LET'S DROP into a world you can reach out and touch. In this article, we'll compose a first-person exploration from scratch, quickly and without difficult math, using a technique called raycasting. You may have seen it before in games like Daggerfall and Duke Nukem 3D, or more recently in Notch Persson's ludum dare entries. If it's good enough for Notch, it's good enough for me!


Demo [demos.playfuljs.com/raycaster]

Raycasting feels like cheating, and as a lazy programmer, I love it. You get the immersion of a 3D environment without many of the complexities of "real 3D" to slow you down. For example, raycasts run in constant time, so you can load up a massive world and it will just work, without optimization, as quickly as a tiny world. Levels are defined as simple grids rather than as trees of polygon meshes, so you can dive right in without a 3D modeling background or mathematics PhD.

It's one of those techniques that blows you away with simplicity. In fifteen minutes you'll be taking photos of your office walls and checking your HR documents for rules against "building workplace gunfight simulations."

The Player

Where are we casting rays from? That's what the player is all about. We need just three properties: x, y, and direction.

```
function Player(x, y, direction) {
  this.x = x;
  this.y = y;
  this.direction = direction;
}
```

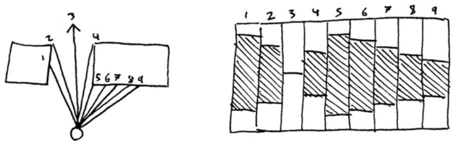
The Map

We'll store our map as a simple two-dimensional array. In this array, 0 represents no wall and 1 represents wall. You can get a lot more complex than this. For example, you could render walls of arbitrary heights, or you could pack several "stories" of wall data into the array, but for our first attempt 0-vs-1 works great.

```
function Map(size) {
  this.size = size;
  this.wallGrid = new Uint8Array(size * size);
}
```

Casting a ray

Here's the trick: a raycasting engine *doesn't draw the whole scene at once*. Instead, it divides the scene into independent columns and renders them one-by-one. Each column represents a single ray cast out from the player at a particular angle. If the ray hits a wall, it measures the distance to that wall and draws a rectangle in its column. The height of the rectangle is determined by the distance the ray traveled. More distant walls are drawn shorter.



The more rays you draw, the smoother the result.

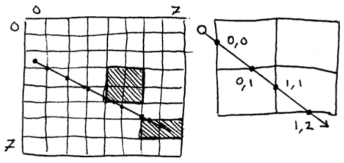
❶ Find each ray's angle

First, we find the angle at which to cast each ray. The angle depends on three things: the direction the player is facing, the focal length of the camera, and which column we're currently drawing.

```
var x = column / this.resolution - 0.5;
var angle = Math.atan2(x, this.focalLength);
var ray = map.cast(player, player.direction +
angle, this.range);
```

❷ Follow each ray through the grid

Next, we need to check for walls in each ray's path. Our goal is to end up with an array that lists each wall the ray passes through as it moves away from the player.



Starting from the player, we find the nearest horizontal (stepX) and vertical (stepY) gridlines. We move to whichever is closer and check for a wall (inspect). Then we repeat until we've traced the entire length of each ray.

```
function ray(origin) {
  var stepX = step(sin, cos, origin.x, origin.y);
  var stepY = step(cos, sin, origin.y, origin.x,
true);
  var nextStep = stepX.length2 < stepY.length2
    ? inspect(stepX, 1, 0, origin.distance,
stepX.y)
    : inspect(stepY, 0, 1, origin.distance,
stepY.x);

  if (nextStep.distance > range) return [origin];
  return [origin].concat(ray(nextStep));
}
```

Finding grid intersections is straightforward: just look for whole numbers of x (1, 2, 3, etc.). Then, find a matching y by multiplying by the line's slope (rise / run).

```
var dx = run > 0 ? Math.floor(x + 1) - x : Math.
ceil(x - 1) - x;
var dy = dx * (rise / run);
```

Did you notice what's awesome about this part of the algorithm? We don't care how big the map is! We're only looking at specific points on the grid — approximately the same number of points each frame. Our example map is 32 x 32, but a map that's 32,000 x 32,000 would run just as quickly!

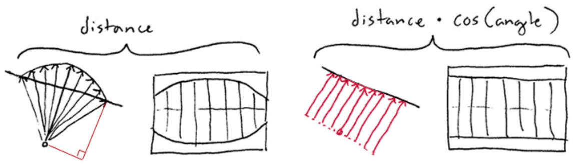
❸ Draw a column

Once we've traced a ray, we need to draw any walls that it found in its path.

```
  var z = distance * Math.cos(angle);
  var wallHeight = this.height * height / z;
```

We determine the height of each wall by dividing its maximum height by z. The further away a wall is, the shorter we draw it.

Oh damn, where did this *cosine* come in? If we just use the raw distance from the player, we'll end up with a fisheye effect. Why? Imagine that you're facing a wall. The edges of the wall to your left and right are further away from you than the center of the wall. But you don't want straight walls to bulge out in the middle! To render flat walls as we really see them, we build a triangle out of each ray and find the perpendicular distance to the wall with cosine. Like this:



And I promise, that's the hardest math in this whole thing.

Render the damn thing!

Let's use a Camera object to draw the map each frame from the player's perspective. It will be responsible for rendering each strip as we sweep from the left to the right of the screen.

Before it draws the walls, we'll render a skybox, just a big picture in the background with stars and a horizon. After the walls are done we'll drop a weapon into the foreground.

```
Camera.prototype.render = function(player, map)
{
  this.drawSky(player.direction, map.skybox,
map.light);
  this.drawColumns(player, map);
  this.drawWeapon(player.weapon, player.paces);
};
```

The camera's most important properties are resolution, focal length, and range.

- *Resolution* determines how many strips we draw each frame: how many rays we cast.
- *Focal length* determines how wide of a lens we're looking through: the angles of the rays.
- *Range* determines how far away we can see: the maximum length of each ray.

Putting it all together

We'll use a `Controls` object to listen for arrow keys (and touch events) and a `GameLoop` object to call `requestAnimationFrame`. Our simple `gameloop` is just three lines:

```
loop.start(function frame(seconds) {
  map.update(seconds);
  player.update(controls.states, map, seconds);
  camera.render(player, map);
});
```

The details

Rain

Rain is simulated with a bunch of very short walls in random places.

```
var rainDrops = Math.pow(Math.random(), 3) * s;
var rain = (rainDrops > 0) && this.project(0.1,
angle, step.distance);

ctx.fillStyle = '#ffffff';
ctx.globalAlpha = 0.15;
while (--rainDrops > 0) ctx.fillRect(left, Math.
random() * rain.top, 1, rain.height);
```

Instead of drawing the walls at their full width, we draw them one pixel wide.

Lighting and lightning

The lighting is actually shading. All walls are drawn at full brightness, and then covered with a black rectangle of some opacity. The opacity is determined by distance as well as by the wall's orientation (N/S/E/W).

```
ctx.fillStyle = '#000000';
ctx.globalAlpha = Math.max((step.distance +
step.shading) / this.lightRange - map.light, 0);
ctx.fillRect(left, wall.top, width, wall.height);
```

To simulate lightning, `map.light` randomly spikes to 2 and then quickly fades down again.

Collision detection

To prevent the player from walking through walls, we just check his future position against our map. We check x and y independently so the player can slide along a wall:

```
Player.prototype.walk = function(distance, map)
{
  var dx = Math.cos(this.direction) * distance;
  var dy = Math.sin(this.direction) * distance;
  if (map.get(this.x + dx, this.y) <= 0) this.x
+= dx;
  if (map.get(this.x, this.y + dy) <= 0) this.y
+= dy;
};
```

Wall textures

The walls would be pretty boring without a texture. How do we know which part of the wall texture to apply to a particular column? It's actually pretty simple: we take the remainder of our intersection point.

```
step.offset = offset - Math.floor(offset);
var textureX = Math.floor(texture.width * step.
offset);
```

For example, an intersection with a wall at (10, 8.2) has a remainder of 0.2. That means that it's 20% from the left edge of the wall (8) and 80% from the right edge (9). So we multiply 0.2 * texture.width to find the x-coordinate for the texture image.

What's next?

Because raycasters are so fast and simple, you can try lots of ideas quickly. You could make a dungeon crawler, first-person shooter, or a grand-theft-auto style sandbox. Hell, the constant-time makes me want to build an oldschool MMORPG with a massive, procedurally generated world.

Fork the code! [hn.my/playfuljs]

Here are a few challenges to get you started:

- Immersion. This example is begging for full-screen mouse-lock with a rainy background and thunderclaps synchronized to the lightning.
- Optimization. Lots of speedups possible here, starting with caching the identical Math.atan2 and Math.cos calls we make hundreds of times each frame.
- An indoors level. Replace the skybox with a symmetric gradient or, if you're feeling plucky, try rendering floor and ceiling tiles (think of it this way: they're just the spaces between the walls you're already drawing!)
- Lighting objects. We already have a fairly robust lighting model. Why not place lights in the world and compute wall lighting based on them? Lights are 80% of atmosphere.
- Good touch events. I've hacked in a couple of basic touch controls so folks on phones and tablets can try out the demo, but there's huge room for improvement.
- Camera effects. For example, zooming, blurring, drunk mode, etc. With a raycaster this are surprisingly simple. Start by modifying camera.fov in the console.

As always, if you build something cool, or have related work to share, link me to it via email [hunter@hunterloftis.com] or twitter [@hunterloftis] and I'll shout it from the rooftops. ■

Hunter Loftis is a full-stack JavaScript junkie based in San Francisco. He's worked for a decade as an illustrator, web designer, Flash animator, and app developer. Hunter shares fun programming techniques in the world's most popular language at *playfuljs.com*

The Last Line Effect

By ANDREY KARPOV

I HAVE STUDIED THE number of errors caused by using the Copy-Paste method and can assure you that programmers most often tend to make mistakes in the last fragment of a homogeneous code block. I have never seen this phenomenon described in books on programming, so I decided to write about it myself. I called it the "last line effect."

Introduction

My name is Andrey Karpov, and I do an unusual job: I analyze program code of various applications with the help of static analyzers and write descriptions of errors and defects I find. I do this for pragmatic and mercenary reasons because what I do is the way our company advertises its tools PVS-Studio and CppCat. The scheme is very simple. I find bugs. Then I describe them in an article. The article attracts our potential customers' attention. Profit. But today's article is not about the analyzers.

When carrying out analysis of various projects, I save bugs I find and the corresponding code fragments in a special database. By the way, anyone interested can take a look at this database. We convert it into a collection of html-pages and upload them to our website in the "Detected errors" section. [viva64.com/en/examples]

This database is unique indeed! It currently contains 1500 code fragments with errors and is waiting for programmers to study it and reveal certain regularity patterns among these errors. That may serve as a useful basis for many future research studies, manuals, and articles.

I have never carried out any special investigation of the material gathered. One pattern, however, is showing up so clearly that I decided to investigate it a bit deeper. You see, in my articles I have to write the phrase "note the last line" pretty often. It occurred to me that there had to be some reason behind it.

Last line effect

When writing program code, programmers often have to write a series of similar constructs. Typing the same code several times is boring and inefficient. That's why they use the Copy-Paste method: a code fragment is copied and pasted several times with further editing. Everyone knows what is bad about this method: you risk easily forgetting to change something in the pasted lines and thus giving birth to errors. Unfortunately, there is often no better alternative to be found.

Now let's speak of the pattern I discovered. I figured out that mistakes are most often made in the last pasted block of code.

Here is a simple and short example:

```
inline Vector3int32& operator+=(const Vecto-
r3int32& other) {
  x += other.x;
  y += other.y;
  z += other.y;
  return *this;
}
```

Note the line "z += other.y;". The programmer forgot to replace "y" with "z" in it.

You may think this is an artificial sample, but it is actually taken from a real application. Further in this article, I am going to convince you that this is a very common issue. This is what the "last line effect" looks like. Programmers most often make mistakes at the very end of a sequence of similar edits.

I heard somewhere that mountain-climbers often fall off at the last few dozens of meters of ascent. Not because they are tired; they are simply too joyful about almost reaching the top — they anticipate the sweet taste of victory, get less attentive, and make some fatal mistake. I guess something similar happens to programmers.

Now a few figures.

Having studied the bug database, I singled out 84 code fragments that I found to have been written through the Copy-Paste method. Out of them, 41 fragments contain mistakes somewhere in the middle of copied-and-pasted blocks. For example:

```
strncmp(argv[argidx], "CAT=", 4) &&
strncmp(argv[argidx], "DECOY=", 6) &&
strncmp(argv[argidx], "THREADS=", 6) &&
strncmp(argv[argidx], "MINPROB=", 8)) {
```

The length of the "THREADS=" string is 8 characters, not 6.

In other 43 cases, mistakes were found in the last copied code block.

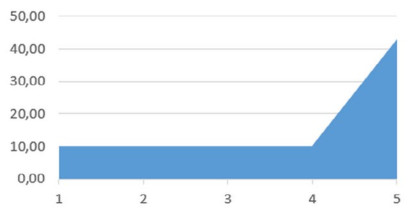
Well, the number 43 looks just slightly bigger than 41. But keep in mind that there may be quite a lot of homogeneous blocks, so mistakes can be found in the first, second, fifth, or even tenth block. So we get a relatively smooth distribution of mistakes throughout blocks and a sharp peak at the end.

I accepted the number of homogeneous blocks to be 5 on the average.

So it appears that the first 4 blocks contain 41 mistakes distributed throughout them; that makes about 10 mistakes per block.

And 43 mistakes are left for the fifth block!

To make it clearer, here is a rough diagram:



A rough diagram of mistake distribution in five homogeneous code blocks.

So what we get is the following pattern:

The probability of making a mistake in the last pasted block of code is 4 times higher than in any other block.

I don't draw any grand conclusions from that. It's just an interesting observation that may be useful to know about for practical reasons so that you stay alert when writing the last fragments of code.

Examples

Source Engine SDK

```
inline void Init( float ix=0, float iy=0,
                  float iz=0, float iw = 0 )
{
  SetX( ix );
  SetY( iy );
  SetZ( iz );
  SetZ( iw );
}
```

The `SetW()` function should be called at the end.

Chromium

```
if (access & FILE_WRITE_ATTRIBUTES)
  output.append(ASCIIToUTF16("\tFILE_WRITE_
ATTRIBUTES\n"));
if (access & FILE_WRITE_DATA)
  output.append(ASCIIToUTF16("\tFILE_WRITE_
DATA\n"));
if (access & FILE_WRITE_EA)
  output.append(ASCIIToUTF16("\tFILE_WRITE_
EA\n"));
if (access & FILE_WRITE_EA)
  output.append(ASCIIToUTF16("\tFILE_WRITE_
EA\n"));
break;
```

The last block and the one before it are identical.

ReactOS

```
if (*ScanString == L'\"' ||
    *ScanString == L'^' ||
    *ScanString == L'\"')
```

Multi Theft Auto

```
class CWaterPolySAInterface
{
public:
    WORD m_wVertexIDs[3];
};
CWaterPoly* CWaterManagerSA::CreateQuad (....)
{
  ....
  pInterface->m_wVertexIDs [ 0 ] = pV1->GetID
();
  pInterface->m_wVertexIDs [ 1 ] = pV2->GetID
();
  pInterface->m_wVertexIDs [ 2 ] = pV3->GetID
();
  pInterface->m_wVertexIDs [ 3 ] = pV4->GetID
();
  ....
}
```

The last line was pasted mechanically and is redundant. There are only 3 items in the array.

Source Engine SDK

```
intens.x=OrSIMD(AndSIMD(BackgroundColor.x,no_
hit_mask),
             AndNotSIMD(no_hit_mask,intens.x));
intens.y=OrSIMD(AndSIMD(BackgroundColor.y,no_
hit_mask),
             AndNotSIMD(no_hit_mask,intens.y));
intens.z=OrSIMD(AndSIMD(BackgroundColor.y,no_
hit_mask),
             AndNotSIMD(no_hit_mask,intens.z));
```

The programmer forgot to replace "BackgroundColor.y" with "BackgroundColor.z" in the last block.

Trans-Proteomic Pipeline

```
void setPepMaxProb(....)
{
  ....
  double max4 = 0.0;
  double max5 = 0.0;
  double max6 = 0.0;
  double max7 = 0.0;
  ....
  if ( pep3 ) { ... if ( use_joint_probs && prob
> max3 ) ... }
  ....
  if ( pep4 ) { ... if ( use_joint_probs && prob
> max4 ) ... }
  ....
  if ( pep5 ) { ... if ( use_joint_probs && prob
> max5 ) ... }
  ....
  if ( pep6 ) { ... if ( use_joint_probs && prob
> max6 ) ... }
  ....
  if ( pep7 ) { ... if ( use_joint_probs && prob
> max6 ) ... }
  ....
}
```

The programmer forgot to replace "prob > max6" with "prob > max7" in the last condition.

SeqAn

```
inline typename Value<Pipe>::Type const & opera-
tor*() {
  tmp.i1 = *in.in1;
  tmp.i2 = *in.in2;
  tmp.i3 = *in.in2;
  return tmp;
}
```

SlimDX

```
for( int i = 0; i < 2; i++ )
{
  sliders[i] = joystate.rglSlider[i];
  asliders[i] = joystate.rglASlider[i];
  vsliders[i] = joystate.rglVSlider[i];
  fsliders[i] = joystate.rglVSlider[i];
}
```

The rglFSlider array should have been used in the last line.

Qt

```
if (repetition == QStringLiteral("repeat") ||
    repetition.isEmpty()) {
  pattern->patternRepeatX = true;
  pattern->patternRepeatY = true;
} else if (repetition == QStringLiteral("repeat-
x")) {
  pattern->patternRepeatX = true;
} else if (repetition == QStringLiteral("repeat-
y")) {
  pattern->patternRepeatY = true;
} else if (repetition == QStringLiteral("no-
repeat")) {
  pattern->patternRepeatY = false;
  pattern->patternRepeatY = false;
} else {
  //TODO: exception: SYNTAX_ERR
}
```

"patternRepeatX" is missing in the very last block. The correct code looks as follows:

```
pattern->patternRepeatX = false;
pattern->patternRepeatY = false;
```

ReactOS

```
const int istride = sizeof(tmp[0]) /
sizeof(tmp[0][0][0]);
const int jstride = sizeof(tmp[0][0]) /
sizeof(tmp[0][0][0]);
const int mistride = sizeof(mag[0]) /
sizeof(mag[0][0]);
const int mjstride = sizeof(mag[0][0]) /
sizeof(mag[0][0]);
```

The "mjstride" variable will always be equal to one. The last line should have been written like this:

```
const int mjstride = sizeof(mag[0][0]) /
sizeof(mag[0][0][0]);
```

Mozilla Firefox

```
if (protocol.EqualsIgnoreCase("http") ||
    protocol.EqualsIgnoreCase("https") ||
    protocol.EqualsIgnoreCase("news") ||
    protocol.EqualsIgnoreCase("ftp") ||
<<<---
    protocol.EqualsIgnoreCase("file") ||
    protocol.EqualsIgnoreCase("javascript") ||
    protocol.EqualsIgnoreCase("ftp")) {
<<<---
```

A suspicious string "ftp" at the end. It has already been compared to.

Quake-III-Arena

```
if (fabs(dir[0]) > test->radius ||
    fabs(dir[1]) > test->radius ||
    fabs(dir[1]) > test->radius)
```

The value from the dir[2] cell is left unchecked.

Clang

```
return (ContainerBegLine <= ContaineeBegLine &&
        ContainerEndLine >= ContaineeEndLine &&
        (ContainerBegLine != ContaineeBegLine ||
         SM.getExpansionColumnNumber(ContainerR
Beg) <=
         SM.getExpansionColumnNumber(ContaineeR
Beg)) &&
        (ContainerEndLine != ContaineeEndLine ||
         SM.getExpansionColumnNumber(ContainerR
End) >=
         SM.getExpansionColumnNumber(ContainerR
End)));
```

At the very end of the block, the "SM.getExpansionColumnNumber(ContainerREnd)" expression is compared to itself.

MongoDB

```
bool operator==(const MemberCfg& r) const {
  ....
  return _id==r._id && votes == r.votes &&
         h == r.h && priority == r.priority &&
         arbiterOnly == r.arbiterOnly &&
         slaveDelay == r.slaveDelay &&
         hidden == r.hidden &&
         buildIndexes == buildIndexes;
}
```

The programmer forgot about "r." in the last line.

Unreal Engine 4

```
static bool PositionIsInside(....)
{
  return
    Position.X >= Control.Center.X - BoxSize.X *
0.5f &&
    Position.X <= Control.Center.X + BoxSize.X *
0.5f &&
    Position.Y >= Control.Center.Y - BoxSize.Y *
0.5f &&
    Position.Y >= Control.Center.Y - BoxSize.Y *
0.5f;
}
```

The programmer forgot to make 2 edits in the last line. Firstly, ">=" should be replaced with "<="; secondly, minus should be replaced with plus.

Qt

```
qreal x = ctx->callData->args[0].toNumber();
qreal y = ctx->callData->args[1].toNumber();
qreal w = ctx->callData->args[2].toNumber();
qreal h = ctx->callData->args[3].toNumber();
if (!qIsFinite(x) || !qIsFinite(y) ||
    !qIsFinite(w) || !qIsFinite(w))
```

In the very last call of the function qIsFinite, the "h" variable should have been used as an argument.

OpenSSL

```
if (!strncmp(vstart, "ASCII", 5))
  arg->format = ASN1_GEN_FORMAT_ASCII;
else if (!strncmp(vstart, "UTF8", 4))
  arg->format = ASN1_GEN_FORMAT_UTF8;
else if (!strncmp(vstart, "HEX", 3))
  arg->format = ASN1_GEN_FORMAT_HEX;
else if (!strncmp(vstart, "BITLIST", 3))
  arg->format = ASN1_GEN_FORMAT_BITLIST;
```

The length of the "BITLIST" string is 7, not 3 characters.

Let's stop here. I hope the examples I have demonstrated are more than enough.

Conclusion

From this article you have learned that with the Copy-Paste method making a mistake in the last pasted block of code is 4 times more probable than in any other fragment.

It has to do with the specifics of human psychology, not professional skills. I have shown you in this article that even highly-skilled developers of such projects as Clang or Qt tend to make mistakes of this kind.

I hope my observation will be useful for programmers and perhaps urge them to investigate our bug database. I believe it will help reveal many regularity patterns among errors and work out new recommendations for programmers. ■

Andrey Karpov is technical director of the OOO "Program Verification Systems" company where his task is to develop source code static analyzers. He has worked for several years in the "CFD Software Group" Scientific Center where he has acquired an exceptional experience of resource-intensive software development in the sphere of computational modeling and visualization. It was there that he noticed an insufficient set of tools for detecting defects in 64-bit software handling large memory amounts. It became the starting point in creation of the Viva64 static analyzer and later the PVS-Studio package.

Reprinted with permission of the original author.
First appeared in *hn.my/lastline* (viva64.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.

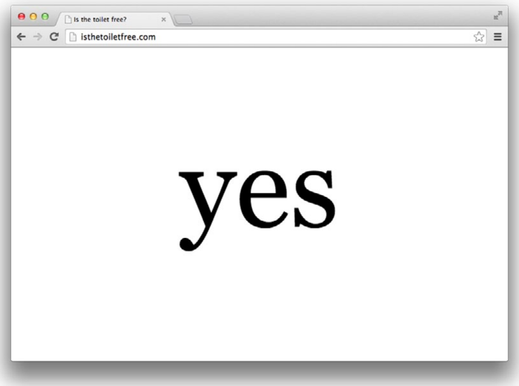
MANDRILL.COM

Is the Toilet Free?

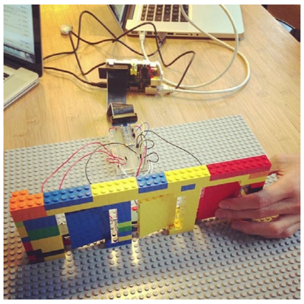
By CALLUM JEFFERIES

A FEW OF US Many have been working on a side project that we've aptly named Is the Toilet Free? Its purpose: to provide an at-desk indication of whether a toilet is free in an effort to remedy that laborious walk to the loo only to find that they're all engaged. Be gone queues and awkward pre-poo chit-chat. (Some of many struggles at MxM.)

It was an idea communicated simply as a website that could do this:

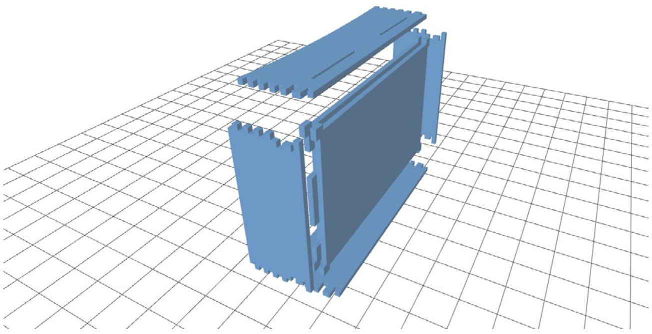

Is the toilet free?
isthetoiletfree.com
yes

Fiona and Raffi began fiddling with a Raspberry Pi to see if they could come up with the bare essential hardware and software. From there (with my near-non-existent knowledge of electronics) I was able to extrapolate on what they had created to add a few more switches and LEDs. We soon had a circuit that resembled the end result. We prototyped…

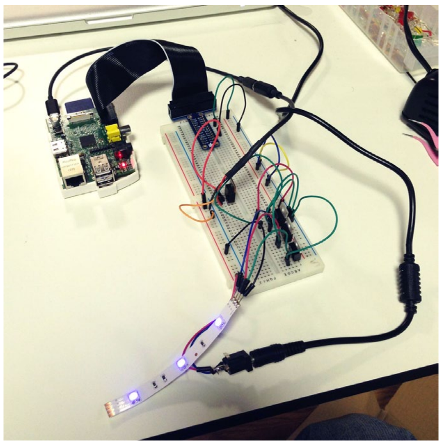


And we had some software that updated the website. It worked brilliantly.

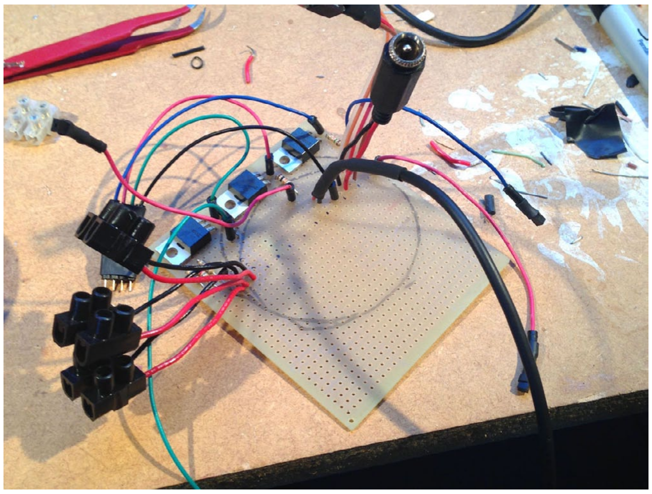
I designed a box to house the Pi and some strip LEDs that would sit on the wall outside the loo.


Source: *hn.my/modelstl*

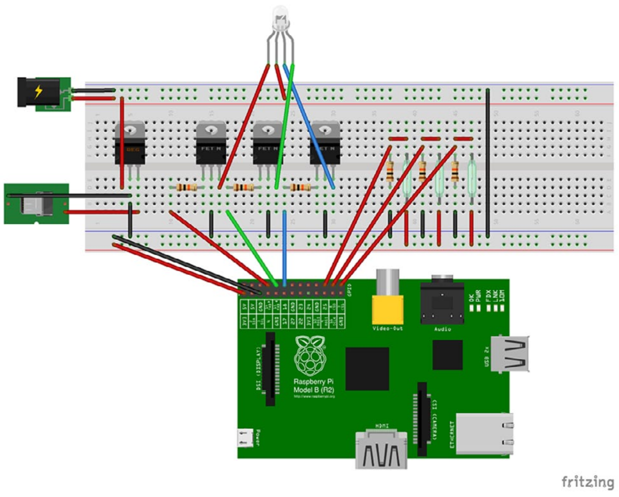
Thus far it had all been hacked-together prototypes, so we needed to formalize what we'd done. I wrote some new software that in principle was: show a green light if there is at least one toilet free, otherwise show red. Then when a change is seen, log it and update the website. Ben and I ordered some components to create the V2 circuit, and tested it on a breadboard:

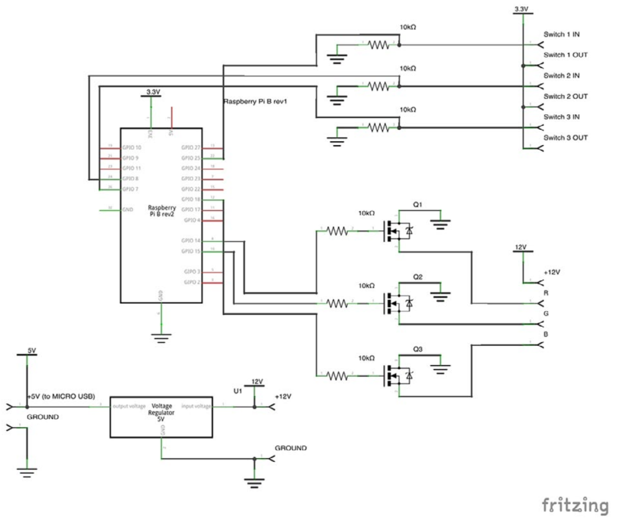


Then transferred and soldered it to some stripboard:



The circuit diagram:





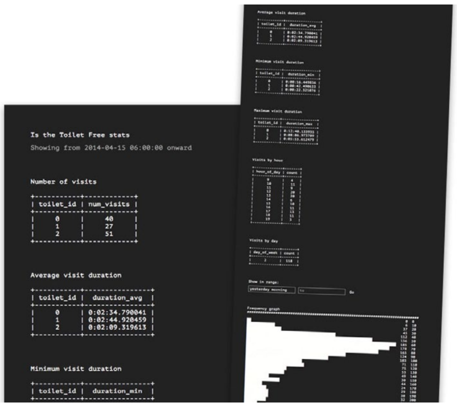
We fit it all in the box, and the DIY man had come to fit some reed switches on the toilet doors. He made a pickle of it. (Sorry, DIY man, if you're reading this.)

Along the way we'd thought about what we could do with the data being logged. We knew in and out times, why not make some charts out of it? To debunk any upfront data uneasiness — because you might be asking "Why am I being recorded going to the toilet?" — privacy was our core concern; the light collectively represents the toilets' state to prevent distinguishing one toilet from another. It doesn't know who you are, and it's not measuring your deposits or anything similarly absurd.

Moving on though, we've been collecting data (rather inconsistently) for about 3 weeks, and composed some (hopefully reliable) SQL queries. Thanks to Dan and Raffi we can tell:

- If the toilets are free
- The total number of visits
- Minimum visit duration
- Maximum visit duration
- Average visit duration
- Total visits by hour
- Total visits by day
- From which we can infer:
- The favorite toilet
- Peak times
- Off-peak times
- An estimated wait time

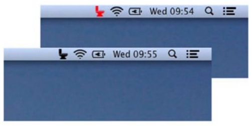
I created a command-line inspired stats page for the above:



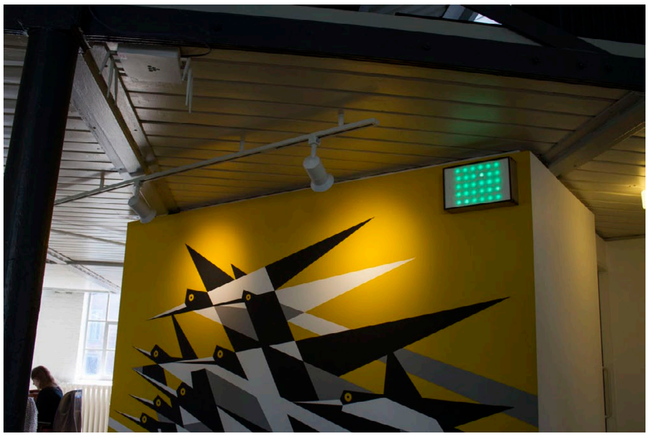
```
Is the Toilet Free stats
Showing from 2014-04-15 06:00:00 onward

Number of visits

+-----------+------------+
| toilet_id | num_visits |
+-----------+------------+
|     0     |     40     |
|     1     |     27     |
|     2     |     51     |
+-----------+------------+

Average visit duration

+-----------+----------------+
| toilet_id |  duration_avg  |
+-----------+----------------+
|     0     | 0:02:34.790041 |
|     1     | 0:02:44.920459 |
|     2     | 0:02:09.319613 |
+-----------+----------------+

Minimum visit duration

+-----------+----------------+
| toilet_id |  duration_min  |
+-----------+----------------+
```

```
Minimum visit duration

+-----------+----------------+
| toilet_id |  duration_min  |
+-----------+----------------+
|     0     | 0:00:16.449836 |
|     1     | 0:00:42.490633 |
|     2     | 0:00:22.321076 |
+-----------+----------------+

Maximum visit duration

+-----------+----------------+
| toilet_id |  duration_max  |
+-----------+----------------+
|     0     | 0:12:40.133955 |
|     1     | 0:08:06.973709 |
|     2     | 0:03:33.612479 |
+-----------+----------------+

Visits by hour

+-------------+-------+
| hour_of_day | count |
+-------------+-------+
|      9      |   4   |
|     10      |  11   |
|     11      |   9   |
|     12      |  20   |
|     13      |  20   |
|     14      |   6   |
|     15      |  10   |
|     16      |  11   |
|     17      |  13   |
|     18      |  11   |
|     19      |   3   |
+-------------+-------+

Visits by day

+-------------+-------+
| day_of_week | count |
+-------------+-------+
|      2      |  118  |
+-------------+-------+

Show in range:
[yesterday morning] [to] Go

Frequency graph
```

Garold's also made something that lives in a Mac OS menu bar:



It's rather pointless but it's all in the name of fun.



What's next?

For now we'll keep collecting data to see what other worthless knowledge we can assimilate. The software has gone as far as seems sensible, but it'd be brilliant to develop the hardware. The sign looks a bit pathetic (that bloody LED that won't glue down!) and it's all far from ideal. I'm trying to document as much of what's been done so far however, because I've learnt a great deal and it makes sense to capture that and share it. The software is on GitHub [hn.my/toiletrepo] should you want to look at it. There's also a wiki that I hope will grow into a guide for making your own.

Raspberry Pis are brilliant; we were able to create Internet-connected hardware with very little. It's empowering to say the least, and the barrier to entry seemed so small. For a project intended as a bit of fun, it's certainly delivered. I've thoroughly enjoyed all of it. It's been an excellent opportunity to explore new things that I'll without-a-doubt pursue further. Personally I would love to see others' attempts at something like this — you're sure to have as much fun. We have a new project in the pipeline that you'll be glad to know doesn't involve fecal matter. Think solenoids and severed fingers. ■

Callum Jefferies is an interaction designer and developer in London. He is currently working with Made by Many. [madebymany.com]

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

Metrics and monitoring for people who know what they want

We know from experience that monitoring your servers and applications can be painful, so we built the sort of service that we would want to use. Simple to set up, responsive support from people who know what they're talking about, and reliably fast metric collection and dashboards.

Dashboards **StatsD** **Happiness**

Now with Grafana!

Why Hosted Graphite?

• **Hosted metrics and StatsD:** Metric aggregation without the setup headaches

• **High-resolution data:** See everything like some glorious mantis shrimp / eagle hybrid*

• **Flexible:** Lots of sample code, available on Heroku

• **Transparent pricing:** Pay for metrics, not data or servers

• **World-class support:** We want you to be happy!

Promo code: **HACKER**

Grab a free trial at http://www.hostedgraphite.com

*Hosted Graphite's mantis shrimp / eagle breeding program has been unsuccessful thus far

HOSTEDGRAPHITE

You **push** it
we **test** it
& **deploy** it

circle**ci**

Get 50% off your first 6 months
circleci.com/?join=hm