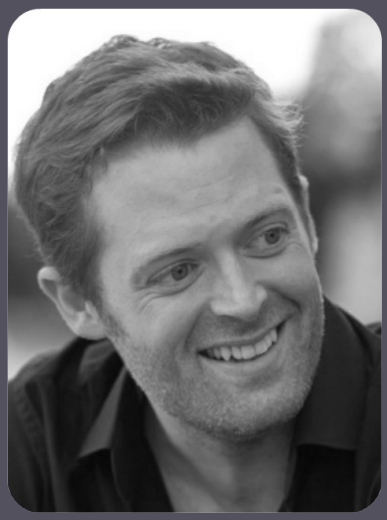
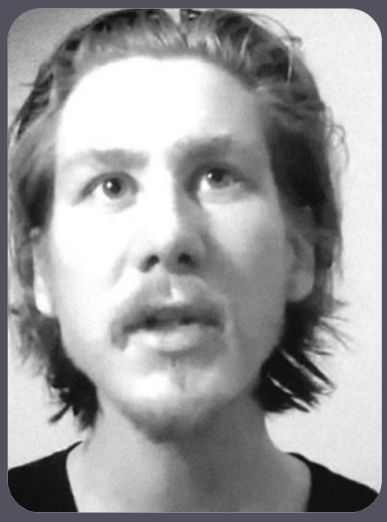
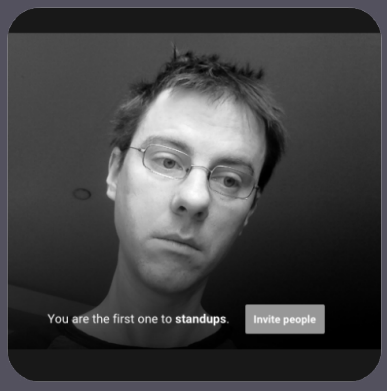
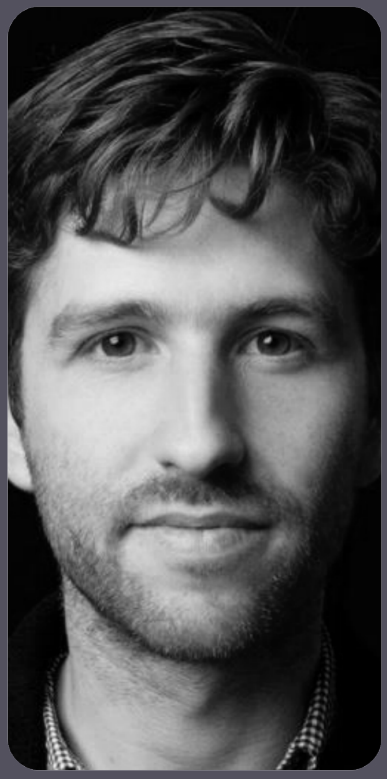




hacker bits

July 2016

new bits

Greetings from Redmond!

Lately, we've been thinking a lot about free will. Do we have it? Do we want it? Is technology freeing? Or does it simply give you a menu to choose from?

In this issue's most highly upvoted story, Google's Design Ethicist Tristan Harris shows us the sneaky ways tech companies manipulate us and the true cost of social media.

Speaking of being social, we love reading mail from you, our readers, and more than a few of you have told us how busy your lives are, and what a challenge it is staying current on technology.

We totally get it (we have a toddler who decided to stop sleeping) and that's why we are simplifying the magazine and focusing only on the essential. ***Learn more***, ***read less*** and ***stay current***.

There are no ads, no redundant info and absolutely no BS. Our hope is that you'll learn something from every article in *Hacker Bits,* or at the very least, get a few chuckles out of it.

So enjoy another issue by our top-notch contributors and feel free to tell us what you think of the magazine!

Peace and plenty of sleep!

— Maureen and Ray

us@hackerbits.com

P.S. If y'all know of any ways to put a toddler to sleep, let us know too! :)

content bits

July 2016

contributor bits

Kent Beck
Kent is an software engineer, author, coach and creator of Extreme Programming. He was one of the 17 signatories of the Agile Manifesto and a leading proponent of TDD. He pioneered design patterns and his current academic project is a study of software design.

Tristan Harris
Tristan was Product Philosopher at Google until 2016 where he studied how technology affects a billion people's attention, wellbeing and behavior. Find more resources on Time Well Spent.

Sebastian Raschka
Sebastian is a Data Scientist developing novel techniques in the computational biology field at Michigan State University and is the author of the bestselling book *Python Machine Learning*.

Glyph Lefkowitz
Glyph works for Rackspace on open source research. He founded the Twisted project and has written Python for everything from online games to enterprise software, from embedded systems to mainframes.

Peter Welch
Peter is a high-functioning alcoholic out of backwater Brooklyn. He pokes at keyboards. Sometimes people give him money for it. He is also the author of And Then I Thought I Was a Fish and Observations of a Straight White Male with No Interesting Fetishes.

Piotr Solnica
Piotr is a software developer from Poland with over a decade of experience. An active Ruby developer since 2007 and Open Source contributor since 2009, he's currently working hard on rom-rb and dry-rb.

Salvatore Sanfilippo
Salvatore is an Italian self taught software developer. He is the main author of a few open source projects including Redis and Disque. He blogs at antirez.com and tweets @antirez.

Bill Sourour
Bill is the founder of DevMastery.com. A 20-year veteran programmer, architect, consultant, and teacher, he helps individual developers and billion-dollar organizations become more successful every day.

James Kettle
James is head of research at PortSwigger Web Security, where he designs and refines vulnerability detection techniques for Burp Suite's scanner.

Chris Beams
Chris is an open source software developer and part of the team behind the Gradle build system.

Ray Li
Curator

Ray is a software engineer and data enthusiast who has been blogging at rayli.net for over a decade. He loves to learn, teach and grow. You'll usually find him wrangling data, programming and lifehacking.

Maureen Ker
Editor

Maureen is an editor, writer, enthusiastic cook and prolific collector of useless kitchen gadgets. She is the author of 3 books and 100+ articles. Her work has appeared in the *New York Daily News*, and various adult and children's publications.

Mastering programming

By KENT BECK

From years of watching master programmers, I have observed certain common patterns in their workflows. From years of coaching skilled journeyman programmers, I have observed the absence of those patterns. I have seen what a difference introducing the patterns can make.

Here are ways effective programmers get the most out of their precious 3e9 seconds on the planet.

The theme here is scaling your brain. The journeyman learns to solve bigger problems by solving more problems at once. The master learns to solve even bigger problems than that by solving fewer problems at once. Part of the wisdom is subdividing so that integrating the separate solutions will be a smaller problem than just solving them together.

Time

- *Slicing.* Take a big project, cut it into thin slices, and rearrange the slices to suit your context. I can always

slice projects finer and I can always find new permutations of the slices that meet different needs.

- *One thing at a time.* We're so focused on efficiency that we reduce the number of feedback cycles in an attempt to reduce overhead. This leads to difficult debugging situations whose expected cost is greater than the cycle overhead we avoided.
- *Make it run, make it right, make it fast.* (Example of One Thing at a Time, Slicing, and Easy Changes)
- *Easy changes.* When faced with a hard change, first make it easy (warning: this may be hard), then make the easy change (e.g. slicing, one thing at a time, concentration, isolation). Example of slicing.
- *Concentration.* If you need to change several elements, first rearrange the code so the change only needs to happen in one element. *Isolation.* If you only need to change a part of an element, extract that part so the whole sub element changes.
- *Baseline measurement.* Start projects by measuring the current state of the world. This goes against our engineering instincts to start fixing things, but when you measure the baseline you will actually know whether you are fixing things.

Learning

- *Call your shot.* Before you run code, predict out loud exactly what will happen.
- *Concrete hypotheses.* When the program is misbehaving, articulate exactly what you think is wrong before making a change. If you have two or more hypotheses, find a differential diagnosis.
- *Remove extraneous detail.* When reporting a bug, find the shortest repro steps. When isolating a bug, find the shortest test case. When using a new API, start from the most basic example. "All that stuff can't possibly matter," is an expensive assumption when it's wrong.

  E.g. see a bug on mobile, reproduce it with curl
- *Multiple scales.* Move between scales freely. Maybe this is a design problem, not a testing problem. Maybe it is a people problem, not a technology problem [cheating, this is always true].

Transcend logic

- *Symmetry.* Things that are almost the same can be divided into parts that are identical and parts that are clearly different.
- *Aesthetics.* Beauty is a powerful gradient to climb. It is also a liberating gradient to flout (e.g. inlining a bunch of functions into one giant mess).
- *Rhythm.* Waiting until the right moment preserves energy and avoids clutter. Act with intensity when the time comes to act.
- *Tradeoffs.* All decisions are subject to tradeoffs. It's more important to know what the decision depends on than it is to know which answer to pick today (or which answer you picked yesterday).

Risk

- *Fun list.* When tangential ideas come, note them and get back to work quickly. Revisit this list when you've reached a stopping spot.
- *Feed ideas.* Ideas are like frightened little birds. If you scare them away they will stop coming around. When you have an idea, feed it a little. Invalidate it as quickly as you can, but from data, not from a lack of self-esteem.
- *80/15/5*. Spend 80% of your time on low-risk/reasonable-payoff work. Spend 15% of your time on related high-risk/high-payoff work. Spend 5% of your time on things that tickle you, regardless of payoff. Teach the next generation to do your 80% job. By the time someone is ready to take over, one of your 15% experiments (or, less frequently, one of your 5% experiments) will have paid off and will become your new 80%. Repeat.

Conclusion

The flow in this outline seems to be from reducing risks by managing time and increasing learning to mindfully taking risks by using your whole brain and quickly triaging ideas. ■

How technology hijacks people's minds — from a magician and Google's design ethicist

By TRISTAN HARRIS

I'm an expert on how technology hijacks our psychological vulnerabilities. That's why I spent the last three years as a Design Ethicist at Google caring about how to design things in a way that defends a billion people's minds from getting hijacked.

When using technology, we often focus *optimistically* on all the things it does for us. But I want to show you where it might do the opposite.

Where does technology exploit our minds' weaknesses?

I learned to think this way when I was a magician. Magicians start by looking for *blind spots, edges, vulnerabilities and limits* of people's perception, so they can influence what people do without them even realizing it. Once you know how to push people's buttons, you can play them like a piano.

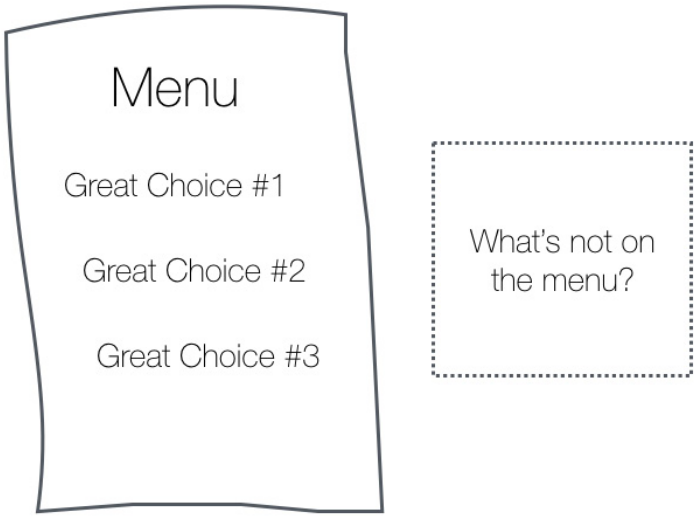
And this is exactly what product designers do to your mind. They play your psychological vulnerabilities (consciously and unconsciously) against you in the race to grab your attention.

I want to show you how they do it.

Hijack #1: If you control the menu, you control the choices

Western culture is built around ideals of individual choice and freedom. Millions of us fiercely defend our right to make "free" choices, while we ignore how those choices are manipulated upstream by menus we didn't choose in the first place.

This is exactly what magicians do. They give people the illusion of free choice while architecting the menu so that they win, no matter what you choose. I can't emphasize enough how deep this insight is.



When people are given a menu of choices, they rarely ask:

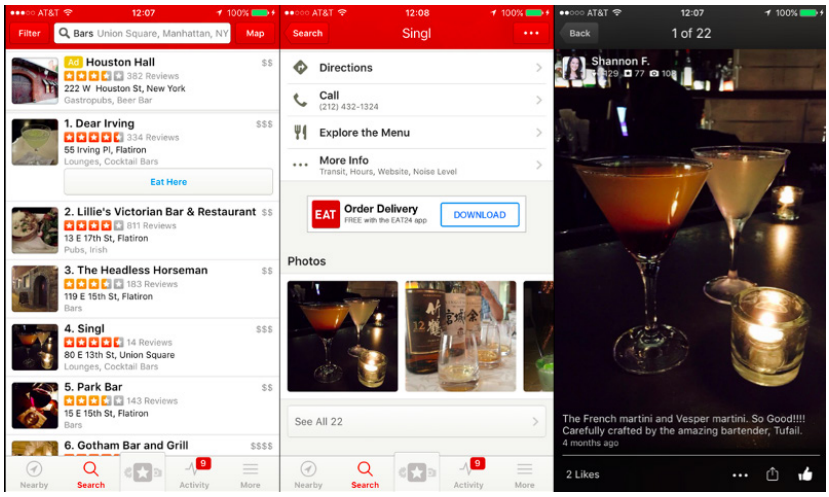
- "what's not on the menu?"
- "why am I being given these options and not others?"
- "do I know the menu provider's goals?"
- "is this menu empowering for my original need, or are the choices actually a distraction?" (e.g. an overwhelmingly array of toothpastes)

▲ How empowering is this menu of choices for the need, "I ran out of toothpaste?"

For example, imagine you're out with friends on a Tuesday night and want to keep the conversation going. You open Yelp to find nearby recommendations and see a list of bars. The group turns into a huddle of faces staring down at their phones *comparing bars*. They scrutinize the photos of each, comparing cocktail drinks. Is this menu still relevant to the original desire of the group?

It's not that bars aren't a good choice, it's that Yelp substituted the group's original question ("where can we go to keep talking?") with a different question ("what's a bar with good photos of cocktails?") all by shaping the menu.

Moreover, the group falls for the illusion that Yelp's menu represents a *complete set of choices* for where to go. While looking down at their phones, they don't see the park across the street with a band playing live music. They miss the pop-up gallery on the other side of the street serving crepes and coffee. Neither of those show up on Yelp's menu.



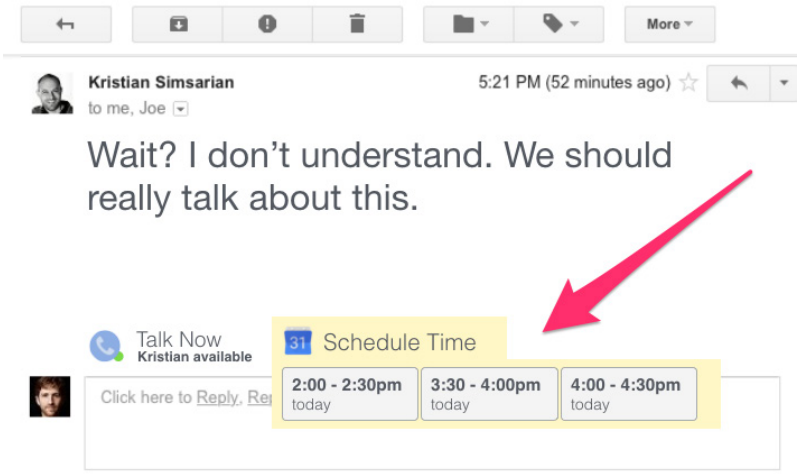
▲ Yelp subtly reframes the group's need of "where can we go to keep talking?" in terms of photos of cocktails served.

The more choices technology gives us in nearly every domain of our lives (information, events, places to go, friends, dating, jobs) — *the more we assume that our phone is always the most empowering and useful menu to pick from*. Is it?

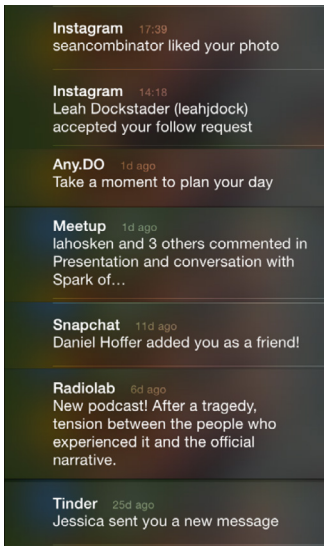
The "most empowering" menu is different than the menu that has the most choices. But when we blindly surrender to the menus we're given, it's easy to lose track of the difference:

- "Who's free tonight to hang out?" becomes a menu of the most recent people who texted us (who we could ping).
- "What's happening in the world?" becomes a menu of news feed stories.
- "Who's single to go on a date?" becomes a menu of faces to swipe on Tinder (instead of local events with friends, or urban adventures nearby).
- "I have to respond to this email." becomes a menu of keys to type a response (instead of empowering ways to communicate with a person).

When we wake up in the morning and turn our phone over to see a list of notifications — it frames the experience of "waking up in the morning" around a menu of "all the things I've missed since yesterday." (For more examples, see Joe Edelman's Empowering Design talk)



▲ All user interfaces are menus. What if your email client gave you empowering choices of ways to respond, instead of "what message do you want to type back?" (Design by Tristan Harris)



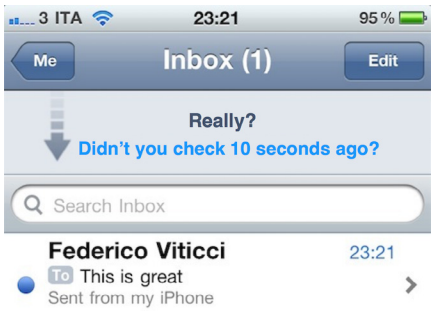
▲ A list of notifications when we wake up in the morning — how empowering is this menu of choices when we wake up? Does it reflect what we care about? (from Joe Edelman's Empowering Design Talk)

By shaping the menus we pick from, technology hijacks the way we perceive our choices and replaces them with new ones. But the closer we pay attention to the options we're given, the more we'll notice when they don't actually align with our true needs.

Hijack #2: Put a slot machine in a billion pockets

If you're an app, how do you keep people hooked? Turn yourself into a slot machine.

The average person checks their phone 150 times a day. Why do we do this? Are we making 150 conscious choices?



▲ How often do you check your email per day?

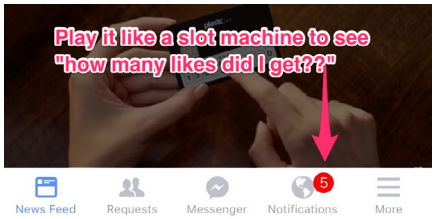
One major reason why is the #1 psychological ingredient in slot machines: intermittent variable rewards.

To maximize addictiveness, all tech designers need to do is link a user's action (like pulling a lever) with a variable reward. You pull a lever and immediately receive either an enticing reward (a match, a prize!) or nothing. Addictiveness is maximized when the rate of reward is most variable.

Does this effect really work on people? Yes. *Slot machines make more money in the United States than baseball, movies, and theme parks combined*. Relative to other kinds of gambling, people get "problematically involved" with slot machines 3–4x faster, according to NYU professor Natasha Dow Schull, author of *Addiction by Design*.

But here's the unfortunate truth — several billion people have a slot machine their pocket:

- When we pull our phone out of our pocket, we're *playing a slot machine* to see what notifications we got.
- When we pull to refresh our email, we're *playing a slot machine* to see what new email we got.
- When we swipe down our finger to scroll the Instagram feed, we're *playing a slot machine* to see what photo comes next.
- When we swipe faces left/right on dating apps like Tinder, we're *playing a slot machine* to see if we got a match.
- When we tap the # of red notifications, we're *playing a slot machine* to what's underneath.



Apps and websites sprinkle intermittent variable rewards all over their products because it's good for business.

But in other cases, slot machines emerge by accident. For example, there is no malicious corporation behind *all of email* that consciously choose to make it a slot machine. No one profits when millions check their email and nothing's there. Neither did Apple and Google's designers want phones to work like slot machines. It emerged by accident.

But now companies like Apple and Google have a responsibility to reduce these effects by *converting intermittent variable rewards into less addictive, more predictable ones* with better design. For example, they could empower people to set predictable times during the day or week for when they want to check "slot machine" apps, and correspondingly adjust when new messages are delivered to align with those times.

Hijack #3: Fear of missing something important (FOMSI)

Another way apps and websites hijack people's minds is by inducing a "1% chance you could be missing something important."

If I convince you that I'm a channel for important information, messages, friendships, or potential sexual opportunities — it will be hard for you to turn me off, unsubscribe, or remove your account — because (aha, I win) you might miss something important:

- This keeps us subscribed to newsletters even after they haven't delivered recent benefits ("what if I miss a future announcement?")
- This keeps us "friended" to people with whom we hav-

en't spoke in ages ("what if I miss something important from them?")

- This keeps us swiping faces on dating apps, even when we haven't even met up with anyone in a while ("what if I miss that *one hot match* who likes me?")
- This keeps us using social media ("what if I miss that important news story or fall behind on what my friends are talking about?")

But if we zoom into that fear, we'll discover that it's unbounded: *we'll always miss something important* at any point when we stop using something.

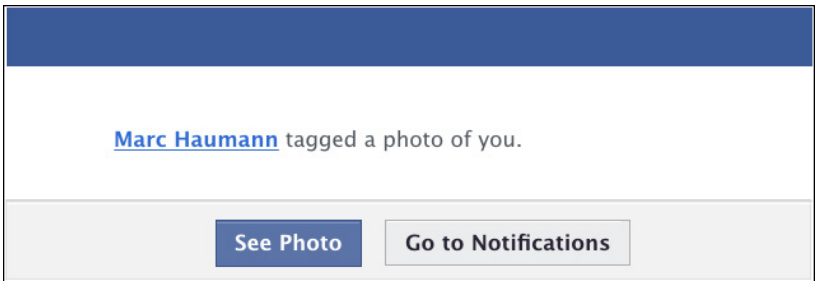
- There are magic moments on Facebook we'll miss by not using it for the 6th hour (e.g. an old friend who's visiting town *right now*).
- There are magic moments we'll miss on Tinder (e.g. our dream romantic partner) by not swiping our 700th match.
- There are emergency phone calls we'll miss if we're not connected 24/7.

But living moment to moment with the fear of missing something isn't how we're built to live.

And it's amazing how quickly, once we let go of that fear, we wake up from the illusion. When we unplug for more than a day, unsubscribe from those notifications, or go to Camp Grounded — the concerns we thought we'd have don't actually happen.

We don't miss what we don't see.

The thought, "what if I miss something important?" is generated *in advance of unplugging, unsubscribing, or turning off* — not after. Imagine if tech companies recognized that, and helped us proactively tune our relationships with friends and businesses in terms of what we define as "time well spent" for our lives, instead of in terms of what we might miss.



▲ Easily one of the most persuasive things a human being can receive.

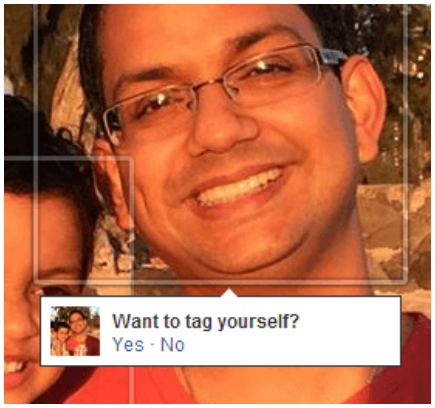
Hijack #4: Social approval

We're all vulnerable to *social approval*. The need to belong, to be approved of or appreciated by our peers is among the highest human motivations. But now our social approval is in the hands of tech companies.

When I get tagged by my friend Marc, I imagine him making a *conscious choice* to tag me. But I don't see how a company like Facebook orchestrated his doing that in the first place.

Facebook, Instagram or SnapChat can manipulate how often people get tagged in photos by automatically suggesting all the faces people should tag (e.g. by showing a box with a 1-click confirmation, "Tag Tristan in this photo?").

So when Marc tags me, *he's actually responding to Facebook's suggestion*, not making an independent choice. But through design choices like this, *Facebook controls the multiplier for how often millions of people experience their social approval on the line.*

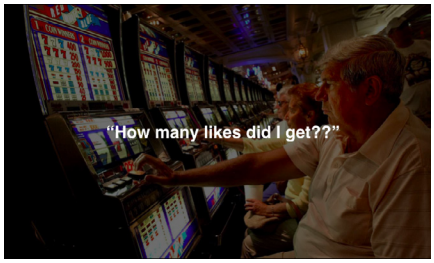


▲ Facebook uses automatic suggestions like this to get people to tag more people, creating more social externalities and interruptions.

The same happens when we change our main profile photo — Facebook knows that's a moment when we're *vulnerable to social approval: "what do my friends think of my new pic?"* Facebook can rank this higher in the news feed, so it sticks around for longer and more friends will like or comment on it. Each time they like or com-

ment on it, we'll get pulled right back.

Everyone innately responds to social approval, but some demographics (teenagers) are more vulnerable to it than others. That's why it's so important to recognize how powerful designers are when they exploit this vulnerability.



Hijack #5: Social reciprocity (tit-for-tat)

- You do me a favor — I owe you one next time.
- You say, "thank you"— I have to say "you're welcome."
- You send me an email — it's rude not to get back to you.
- You follow me — it's rude not to follow you back. (especially for teenagers)

We are *vulnerable to needing to reciprocate others' gestures*. But as with social approval, tech companies now manipulate how often we experience it.

In some cases, it's by accident. *Email, texting and messaging apps are social reciprocity factories*. But in other cases, companies exploit this vulnerability on purpose.

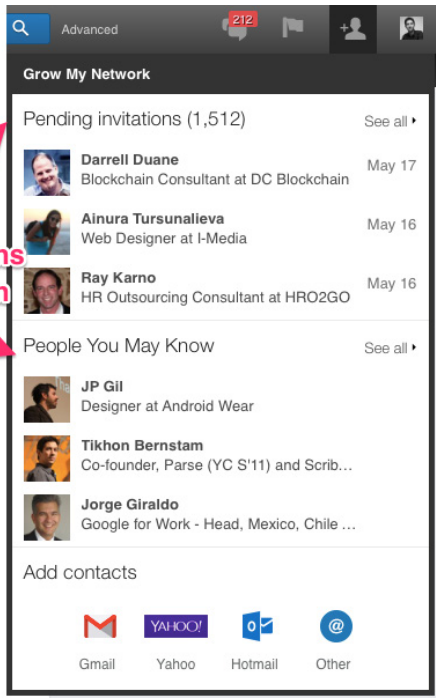
LinkedIn is the most obvious offender. LinkedIn wants as many people creating social obligations for each other as possible, because each time they reciprocate (by accepting a connection, responding to a message, or endorsing someone back for a skill) they have to come back to linkedin.com where they can get people to spend more time.

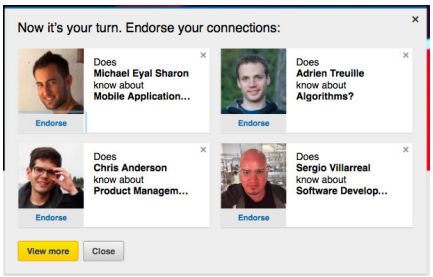
Like Facebook, LinkedIn exploits an asymmetry in perception. When you receive an invitation from someone to connect, you imagine that person making a *conscious choice* to invite you, when in reality, they likely unconsciously responded to LinkedIn's list of suggested contacts. In other words, LinkedIn turns your *unconscious impulses* (to "add" a person) into new social obligations that millions of people feel obligated to repay. All while they profit from the time people spend doing it.

Imagine millions of people getting interrupted like this throughout their day, running around like chickens with their heads cut off, reciprocating each other — all designed by companies who profit from it.

Welcome to social media.

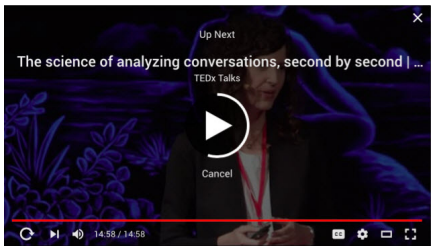
Imagine if technology companies had a responsibility to minimize social reciprocity. Or if there was an independent organization that represented the public's interests — an industry consortium or an FDA for tech — that monitored when technology companies abused these biases?



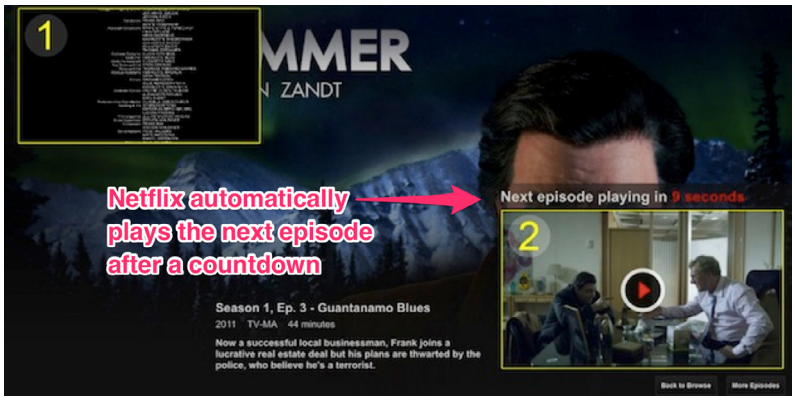


▲ After accepting an endorsement, LinkedIn takes advantage of your bias to reciprocate by offering *four* additional people for you to endorse in return.

Hijack #6: Bottomless bowls, infinite feeds, and autoplay



▲ YouTube autoplays the next video after a countdown



▲ Netflix also autoplays

Another way to hijack people is to keep them consuming things, even when they aren't hungry anymore.

How? Easy. *Take an experience that was bounded and finite, and turn it into a bottomless flow that keeps going.*

Cornell professor Brian Wansink demonstrated this in his study showing you can trick people into keep eating soup by giving them a bottomless bowl that automatically refills as they eat. With bottomless bowls, people eat 73% more calories than those with normal bowls and underestimate how many calories they ate by 140 calories.

Tech companies exploit the same principle. News feeds are purposely designed to auto-refill with reasons to keep you scrolling, and purposely eliminate any reason for you to pause, reconsider or leave.

It's also why video and social media sites like Netflix, YouTube or Facebook *autoplay* the next video after a countdown instead of waiting for you to make a conscious choice (in case you won't). A huge portion of traffic on these websites is driven by autoplaying the next thing.

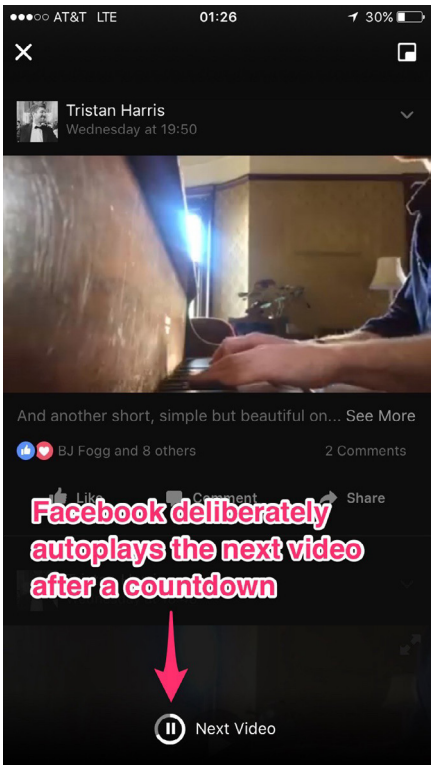
Tech companies often claim that "we're just making it easier for users to see the video *they want* to watch" when they are actually serving their business interests. And you can't blame them, because increasing "time spent" is the currency they compete for.

Instead, imagine if technology companies empowered you to *consciously bound your experience* to align with what would be "time well spent" for you. Not just bounding the *quantity* of time you spend, but the *qualities* of what would be "time well spent."

Hijack #7: Instant interruption vs. "respectful" delivery

Companies know that messages *that interrupt people immediately are more persuasive at getting people to respond* than messages delivered asynchronously (like email or any deferred inbox).

Given the choice, Facebook Messenger (or WhatsApp, WeChat or SnapChat for that matter) would *prefer to design their messaging system to interrupt recipients immediately*



(and show a chat box) instead of helping users respect each other's attention.

In other words, *interruption is good for business.*

It's also in their interest to heighten the feeling of urgency and social reciprocity. For example, Facebook automatically *tells the sender when you "saw" their message, instead of letting you*

avoid disclosing whether you read it ("now that you know I've seen the message, I feel even more obligated to respond.")

By contrast, Apple more respectfully lets users toggle "Read Receipts" on or off.

The problem is, maximizing interruptions in the name of business creates a tragedy of the commons, ruining global attention spans and causing billions of unnecessary interruptions each day. This is a huge problem we need to fix with shared design standards (potentially, as part of Time Well Spent).

Hijack #8: Bundling your reasons with their reasons

Another way apps hijack you is by taking *your reasons* for visiting the app (to perform a task) and *make them inseparable from the app's business reasons* (maximizing how much we consume once we're there).

For example, in the physical world of grocery stores, the #1 and #2 most popular reasons to visit are pharmacy refills and buying milk. But grocery stores want to maximize how much people buy, so they put the pharmacy and the milk at the back of the store.

In other words, they make the thing customers want (milk, pharmacy) inseparable from what the business wants. If stores were truly organized to support people, they would put the most popular items in the front.

Tech companies design their websites the same way. For example, when you you want to look up a Facebook event happening tonight (your reason), the Facebook app doesn't allow you to access it without first landing on the news feed (their reasons), and that's on purpose. *Facebook wants to convert every reason you have for using Facebook, into their reason which is to maximize the time you spend consuming things.*

Instead, imagine if …

- Facebook gave a *separate way* to look up or host Facebook Events, without being forced to use their news feed.
- Facebook gave you a *separate way* to use Facebook Connect as a passport for creating accounts on new apps and websites, without being forced to use Facebook's entire app, news feed and notifications.
- Email gave you a *separate way* to look up and reply to a specific message, without being forced to see all new unread messages.

In an ideal world, there is always a *direct way* to get what you want *separately* from what businesses want.

Imagine a digital "bill of rights" outlining design standards that forced the products used by billions of people to support empowering ways for them to navigate toward their goals.

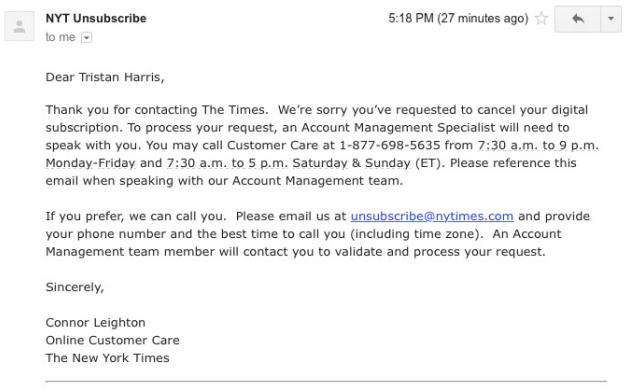
Hijack #9: Inconvenient choices

We're told that it's enough for businesses to "make choices available."

- "If you don't like it, you can always use a different product."
- "If you don't like it, you can always unsubscribe."
- "If you're addicted to our app, you can always uninstall it from your phone."

Businesses naturally *want to make the choices they want you to make easier, and the choices they don't want you to make harder.* Magicians do the same thing. You make it easier for a spectator to pick the thing you want them to pick, and harder to pick the thing you don't.

For example, NYTimes.com lets you "make a free choice" to cancel your digital subscription.



NYT Unsubscribe 5:18 PM (27 minutes ago)
to me

Dear Tristan Harris,

Thank you for contacting The Times. We're sorry you've requested to cancel your digital subscription. To process your request, an Account Management Specialist will need to speak with you. You may call Customer Care at 1-877-698-5635 from 7:30 a.m. to 9 p.m. Monday-Friday and 7:30 a.m. to 5 p.m. Saturday & Sunday (ET). Please reference this email when speaking with our Account Management team.

If you prefer, we can call you. Please email us at unsubscribe@nytimes.com and provide your phone number and the best time to call you (including time zone). An Account Management team member will contact you to validate and process your request.

Sincerely,

Connor Leighton
Online Customer Care
The New York Times

▲ NYTimes claims it's giving a free choice to cancel your account

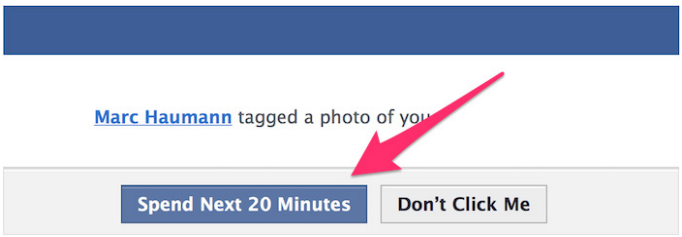
But instead of just doing it when you hit "Cancel Subscription," they *send you an email with information on how to cancel your account by calling a phone number that's only open at certain times*.

Instead of viewing the world in terms of *availability of choices*, we should view the world in terms of *friction required to enact choices*. Imagine a world where choices were labeled with how difficult they were to fulfill (like coefficients of friction) and there was an independent entity — an industry consortium or non-profit — that labeled these difficulties and set standards for how easy navigation should be.

Hijack #10: Forecasting errors, "foot in the door" strategies

Lastly, apps can exploit people's inability to forecast the consequences of a click.

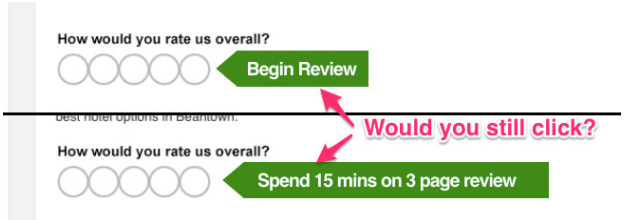
People don't intuitively forecast the *true cost of a click* when it's presented to them. Sales people use "foot in the door" techniques by asking for a small innocuous request to begin with ("just one click to see which tweet got retweeted") and escalate from there ("why don't you stay awhile?"). Virtually all engagement websites use this trick.



▲ Facebook promises an easy choice to "See Photo." Would we still click if it gave the true price tag?

Imagine if web browsers and smartphones, the gateways through which people make these choices, were truly watching out for people and helped them forecast the consequences of clicks (based on real data about what benefits and costs it actually had?).

That's why I add "estimated reading time" to the top of my posts. When you put the "true cost" of a choice in front of people, you're treating your users or audience with dignity and respect. In a Time Well Spent Internet, choices could be framed in terms of projected cost and benefit, so people were empowered to make informed choices by default, not by doing extra work.



▲ TripAdvisor uses a "foot in the door" technique by asking for a single click review ("How many stars?") while hiding the three page survey of questions behind the click.

Summary and how we can fix this

Are you upset that technology hijacks your agency? I am too. I've listed a few techniques but there are literally thousands. Imagine whole bookshelves, seminars, workshops and trainings that teach aspiring tech entrepreneurs techniques like these. Imagine hundreds of engineers whose job every day is to invent new ways to keep you hooked.

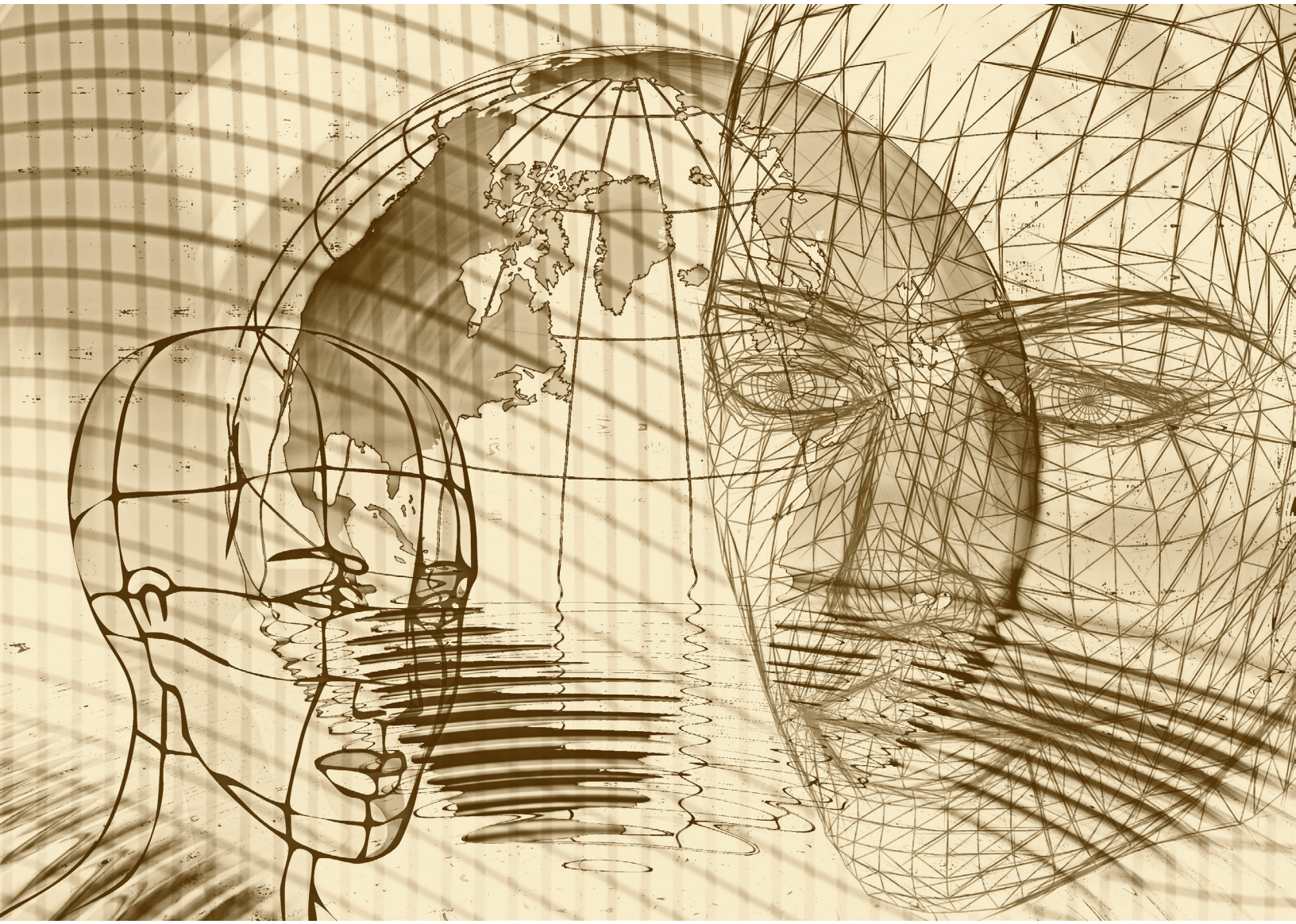
The ultimate freedom is a free mind, and we need technology that's on our team to help us live, feel, think and act freely.

We need our smartphones, notifications screens and web browsers to be exoskeletons for our minds and interpersonal relationships that put our values, not our impulses, first. People's time is valuable. And we should protect it with the same rigor as privacy and other digital rights. ■

UPDATE: The first version of this post lacked acknowledgements to those who inspired my thinking over many years including Joe Edelman, Aza Raskin, Raph D'Amico, Jonathan Harris and Damon Horowitz.

My thinking on menus and choicemaking are deeply rooted in Joe Edelman's work on Human Values and Choicemaking.


hacker bits
hackerbits.com/mobile



What is the difference between deep learning and usual machine learning?

By SEBASTIAN RASCHKA

That's an interesting question, and I'll try to answer this in a very general way. In essence, deep learning offers a set of techniques and algorithms that help us to parameterize deep neural network structures – artificial neural networks with many hidden layers and parameters.

One of the key ideas behind deep learning is to extract high level features from the given dataset. Thereby, deep learning aims to overcome the challenge of the often tedious feature engineering task and helps with parameterizing traditional neural networks with many layers.

Now, to introduce deep learning, let us take a look at a more concrete example involving multi-layer perceptrons (MLPs).

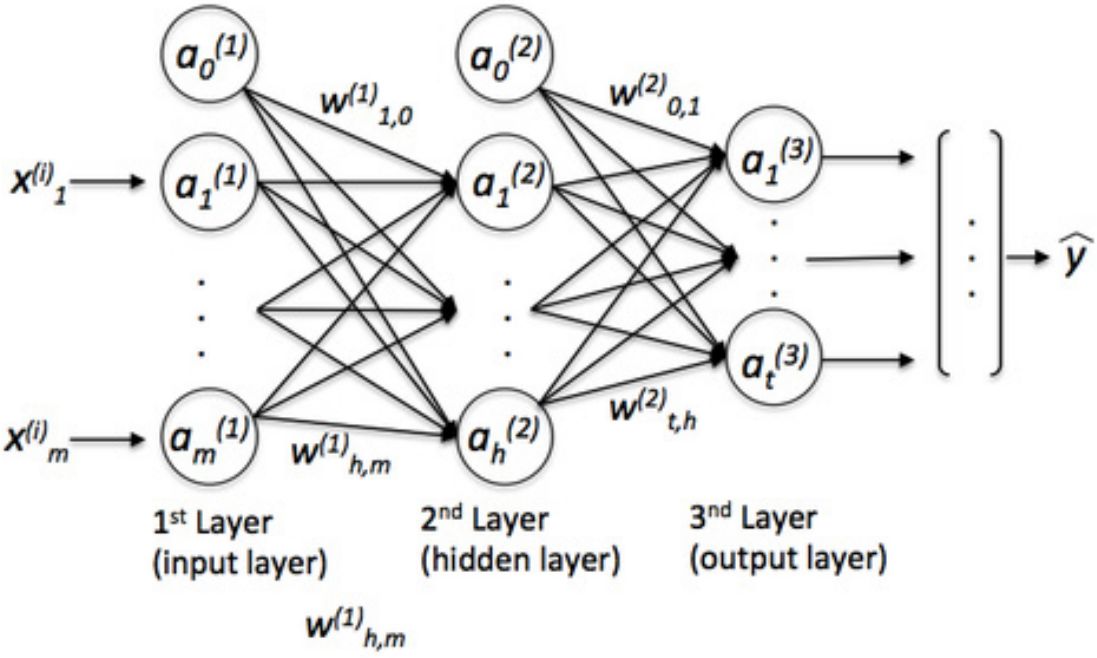
On a tangent: The term "perceptron" in MLPs may be a bit confusing since we don't really want only linear neurons in our network. Using MLPs, we want to learn complex functions to solve non-linear problems. Thus, our network is conventionally composed of one or multiple "hidden" layers that connect the input and output layer.

Those hidden layers normally have some sort of sigmoid activation function (log-sigmoid or the hyperbolic tangent etc.). For example, think of a log-sigmoid unit in our network as a logistic regression unit that returns continuous values outputs in the range 0-1. A simple MLP could look like the figure below.

In the figure below, y_hat is the final class label that we return as the prediction based on the inputs (x) if this are classification tasks. The "a"s are our activated neurons and the "w"s are the weight coefficients.

Now, if we add multiple hidden layers to this MLP, we'd also call the network "deep." The problem with such "deep" networks is that it becomes tougher and tougher to learn "good" weights for this network.

When we start training our network, we typically assign random values as initial weights, which can be terribly off from the "optimal" solution we want to find. During training, we then use the popular backpropaga-



▲ Simple multi-layer perceptron

tion algorithm (think of it as reverse-mode auto-differentiation) to propagate the "errors" from right to left and calculate the partial derivatives with respect to each weight to take a step into the opposite direction of the cost (or "error") gradient.

Now, the problem with deep neural networks is the so-called "vanishing gradient" — the more layers we add, the harder it becomes to "update" our weights because the signal becomes weaker and weaker. Since our network's weights can be terribly off in the beginning (random initialization), it can become almost impossible to parameterize a "deep" neural network with backpropagation.

Deep learning

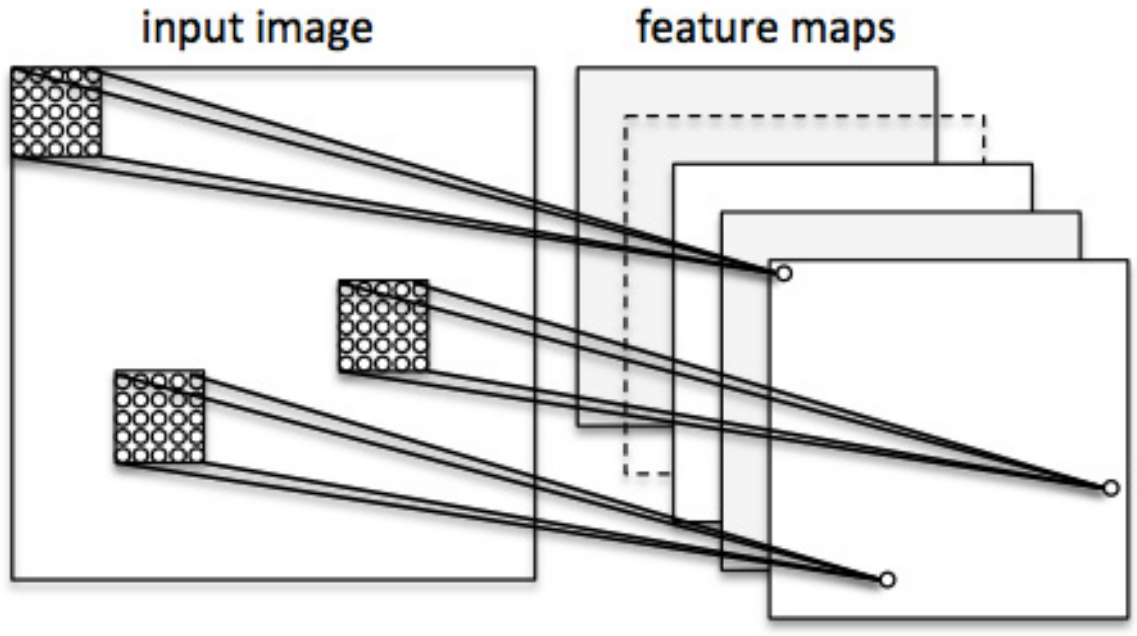
Now, this is where "deep learning" comes into play. Roughly speaking, we can think of deep learning as "clever" tricks or algorithms that can help us with the training of such "deep" neural network structures.

There are many, many different neural network architectures, but to continue with the example of the MLP, let me introduce the idea of convolutional neural networks (ConvNets). We can think of those as an "add-on" to our MLP that helps us detect features as "good" inputs for our MLP.

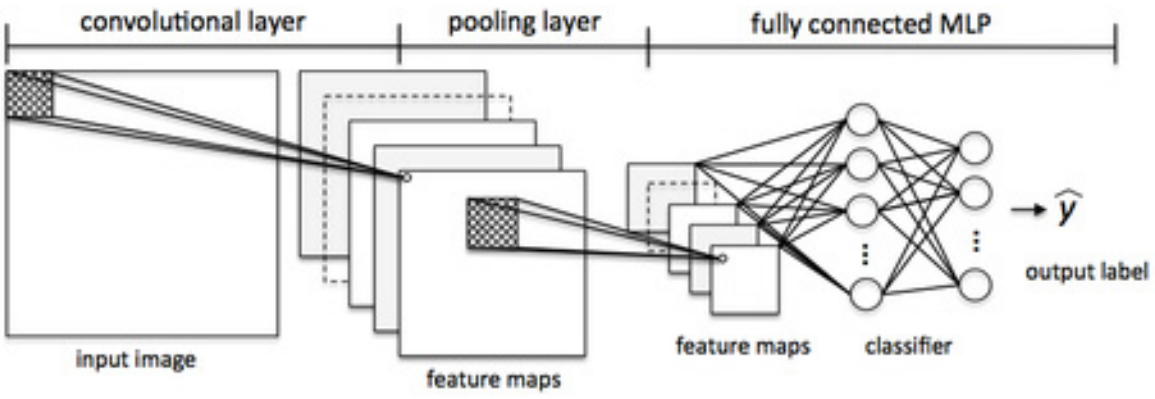
In applications of "usual" machine learning, there is typically a strong focus on the feature engineering part; the model learned by an algorithm can only be as good as its input data.

Of course, there must be sufficient discriminatory information in our dataset, however, the performance of machine learning algorithms can suffer substantially when the information is buried in meaningless features. The goal behind deep learning is to automatically learn the features from (somewhat) noisy data; it's about algorithms that do the feature engineering for us to provide deep neural network structures with meaningful information so that it can learn more effectively.

We can think of deep learning as algorithms for automatic "feature engineering," or we could simply call them "feature detectors," which help us to



▲ So-called "feature map"



▲ Convolutional add-on layer

overcome the vanishing gradient challenge and facilitate the learning in neural networks with many layers.

Let's consider a ConvNet in context of image classification. Here, we use so-called "receptive fields" (think of them as "windows") that slide over our image. We then connect those "receptive fields" (for example of the size of 5x5 pixel) with 1 unit in the next layer, this is the so-called "feature map."

After this mapping, we have constructed a so-called convolutional layer. Note that our feature detectors are basically replicates of one another – they share the same weights. The idea is that if a feature detector is useful in one part of the imagie, it is likely that it is useful somewhere else, but at the same time it allows each patch of the image to be represented in several ways.

Next, we have a "pooling" layer, where we reduce neighboring features from our feature map into single units (by taking the max feature or by averaging them, for example). We do this over many rounds and eventually arrive at an almost scale invariant representation of our image (the exact term is "equivariant"). This is very powerful since we can detect objects in an image no matter where they are located.

In essence, the "convolutional" add-on that acts as a feature extractor or filter to our MLP. Via the convolutional layers we aim to extract the useful features from the images, and via the pooling layers, we aim to make the features somewhat equivariant to scale and translation. ■

Email isn’t the thing you’re bad at

By GLYPH LEFKOWITZ

You and me, we’re bad at a lot of things. But email isn’t one of those things, no matter how much it seems like it.

I’ve been using the Internet for a good 25 years now, and I’ve been lucky enough to have some perspective dating back farther than that. The common refrain for my entire tenure here: We all get too much email.

A new, new, new, new hope

Luckily, something is always on the cusp of replacing email. AOL Instant Messenger will totally replace it. Then it was blogging. RSS. MySpace. Then it was FriendFeed. Then Twitter. Then Facebook.

Today, it’s in vogue to talk about how Slack is going to replace email. As someone who has seen this play out a dozen times now, let me give you a little spoiler: Slack is not going to replace email.

But Slack isn’t the problem here, either. It’s just another communication tool.

The problem of email overload is both ancient and persistent. If the problem were really with “email,” then presumably

one of the nine million email apps that dot the app stores, like mushrooms sprouting from a globe-spanning mycelium, would have just solved it by now, and we could all move on with our lives. Instead, it is permanently in vogue[1] to talk about how overloaded we all are.

If not email, then what?

If you have twenty-four thousand unread emails in your Inbox, like some kind of goddamn animal, what you're bad at is not email, it's transactional interactions.

Different communication media have different characteristics, but the defining characteristic of email is that it is the primary mode of communication that we use, both professionally and personally, when we are asking someone else to perform a task.

Of course you might use any form of communication to communicate tasks to another person. But other forms — especially the currently popular real-time methods — appear as bi-directional communication and are largely immutable.

Email's distinguishing characteristic is that it is discrete; each message is its own entity with its own ID. Emails may also be annotated, whether with flags, replied-to markers, labels, placement in folders, archiving, or deleting.

Contrast this with a group chat in IRC, iMessage, or Slack, where the log is mostly[2] unchangeable, and the only available annotation is "did your scrollbar ever move down past this point"; each individual message has only one bit of associated information. Unless you have catlike reflexes and an unbelievably obsessive-compulsive personality, it is highly unlikely that you will carefully set the "read" flag on each and every message in an extended conversation.

All this makes email much more suitable for communicating a task, because the recipient can file it according to their system for tracking tasks, come back to it later, and generally treat the message itself as an artifact. By contrast if I were to just walk up to you on the street and say "hey can you do this for me," you will almost certainly just forget.

The word "task" might seem heavyweight for some of the things that email is used for, but tasks come in all sizes. One task might be "click this link to confirm your sign-up on this website." Another might be "choose a time to get together for coffee." Or "please pass along my résumé to your hiring department." Yet another might be "send me the final draft of the Henderson report."

Email is also used for conveying information: here are the minutes from that meeting we were just in. Here is the transcription of the whiteboard from that design session. Here are some photos from our family vacation. But even in these cases, a task is implied: read these minutes and see if they're accurate; inspect this diagram and use it to inform your design; look at these photos and just enjoy them.

So here's the thing that you're bad at, which is why none of the fifty different email apps you've bought for your phone have fixed the problem: when you get these messages, you aren't making a conscious decision about:

1. how important the message is to you
2. whether you want to act on them at all
3. when you want to act on them
4. what exact action you want to take
5. what the consequences of taking or not taking that action will be

This means that when someone asks you to do a thing, you probably aren't going to do it. You're going to pretend to com-

What you're bad at is not email, it's transactional interactions.

If you're not going to do anything with it, just archive it and forget about it.

mit to it, and then you're going to flake out when push comes to shove. You're going to keep context-switching until all the deadlines have passed.

In other words: The thing you are bad at is saying 'no' to people.

Sometimes it's not obvious that what you're doing is saying 'no'. For many of us — and I certainly fall into this category — a lot of the messages we get are vaguely informational.

They're from random project mailing lists, perhaps they're discussions between other people, and it's unclear what we should do about them (or if we should do anything at all). We hang on to them (piling up in our Inboxes) because they might be relevant in the future. I am not advocating that you have to reply to every dumb mailing list email with a 5-part action plan and a Scrum meeting invite: that would be a disaster. You don't have time for that. You really shouldn't have time for that

The trick about getting to Inbox Zero[3] is not in somehow becoming an email-reading machine, but in realizing that most email is worthless, and that's OK. If you're not going to do anything with it, just archive it and forget about it.

If you're subscribed to a mailing list where only 1 out of 1,000 messages actually represents something you should do about it, archive all the rest after only answering the question "is this the one I should do something about?" You can answer that question after just glancing at the subject; there are times when checking my email I will be hitting "archive" with a 1-second frequency. If you are on a list where zero messages are ever interesting enough to read in their entirety or do anything about, then of course you should unsubscribe.

Once you've dug yourself into a hole with thousands of "I don't know what I should do with this" messages, it's time to declare email bankruptcy. If you have 24,000 messages in your Inbox, let me be real with you: you are never, ever going to answer all those messages. You do not need a smartwatch to tell you exactly how many messages you are never going to reply to.

We're in this together, me especially

A lot of guidance about what to do with your email deals with email overload as a personal problem. Over the years of developing my tips and tricks for dealing with it, I certainly saw it that way. But lately, I'm starting to see that it has pernicious social effects.

If you have 24,000 messages in your Inbox, that means you aren't keeping track or setting priorities on which tasks you want to complete. But just because you're not setting those priorities, that doesn't mean nobody is. It means you are letting availability heuristic — whatever is "latest and loudest" — govern access to your attention, and therefore your time.

By doing this, you are rewarding people (or #brands) who contact you repeatedly, over inappropriate channels, and generally try to flood your attention with their priorities instead of your own. This, in turn, creates a culture where it is considered reasonable and appropriate to assume that you need to do that in order to get someone's attention.

Since we live in the era of subtext and implication, I should explicitly say that I'm not describing any specific work environment or community. I used to have an email startup, and so I thought about this stuff very heavily for almost a decade. I have seen email habits at dozens of companies, and I help people in the open source community with their email on a regular basis. So I'm not throwing shade: almost everybody is terrible at this.

And that is the one way that email, in the sense of the tools and programs we use to process it, is at fault: technology has made it easier and easier to ask people to do more and more things, without giving us better tools or training to deal with the increasingly huge array of demands on our time. It's easier than ever to say "hey could you do this for me" and harder than

ever to just say “no, too busy.”

Mostly, though, I want you to know that this isn’t just about you any more. It’s about someone much more important than you: me.

I’m tired of sending reply after reply to people asking to “just circle back” or asking if I’ve seen their email. Yes, I’ve seen your email. I have a long backlog of tasks, and, like anyone, I have trouble managing them and getting them all done[4], and I frequently have to decide that certain things are just not important enough to do. Sometimes it takes me a couple of weeks to get to a message. Sometimes I never do. But, it’s impossible to be mad at somebody for “just checking in” for the fourth time when this is probably the only possible way they ever manage to get anyone else to do anything.

I don’t want to end on a downer here, though. And I don’t have a book to sell you which will solve all your productivity problems. I know that if I lay out some incredibly elaborate system all at once, it’ll seem overwhelming. I know that if I point you at some amazing gadget that helps you keep track of what you want to do, you’ll either balk at the price or get lost fiddling with all its knobs and buttons, and not getting a lot of benefit out of it. So if I’m describing a problem that you have here, here’s what I want you to do.

Step zero is setting aside some time. This will probably take you a few hours, but trust me, they will be well-spent.

Email bankruptcy

First, you need to declare email bankruptcy. Select every message in your Inbox older than 2 weeks. Archive them all, right now.

In the past, you might have to worry about deleting those messages, but modern email systems pretty much universally have more storage than you’ll ever need. So rest assured that if you actually need to do anything with these messages, they’ll all be in your archive. But anything in your Inbox right now that's older than a couple of weeks is just never going to get dealt with, and it’s time to accept that fact. Again, this part of the process is not about making a decision yet, it’s just about accepting reality.

Mailbox three

One extra tweak I would suggest here is to get rid of all of your email folders and filters. It seems like many folks with big email problems have tried to address this by ever-finer-grained classification of messages, ever more byzantine email rules. For me, it’s common, when looking over someone’s shoulder to see 24,000 messages, it’s common to also see 50 folders. Probably these aren’t helping you very much.

In older email systems, it was necessary to construct elaborate header-based filtering systems so that you can later identify those messages in certain specific ways, like “message X went to this mailing list”. However, this was an incomplete hack, a workaround for a missing feature. Almost all modern email clients (and if yours doesn’t do this, switch) allow you to locate messages like this via search.

Your mail system ought to have 3 folders:

1. Inbox, which you process to discover tasks
2. Drafts, which you use to save progress on replies
3. Archive, the folder which you access only by searching for information you need when performing a task

Getting rid of unnecessary folders and queries and filter rules will remove things that you can fiddle with.

Moving individual units of trash between different heaps of trash is not being productive. By removing all the different folders you can shuffle your messages into before actually acting upon them, you will make better use of your time spent looking at your email client.

There’s one exception to this rule, which is filters that do nothing but cause a message to skip your Inbox and go straight to the archive. The reason that this type of filter is different is that there are certain sources or patterns of messages which are not actionable, but rather, a useful source of reference material that is only available as a stream of emails. Messages like that should, indeed, not show up in your Inbox. But, there’s no reason to file them into a specific folder or set of folders; you can always find them with a search.

Make a place for tasks

Next, you need to get a task list. Your email is not a task list; tasks are things that you decided you’re going to do, not things that other people have asked you to do[5]. Critically, you are going to need to parse

Tasks are things that you decided you're going to do, not things that other people have asked you to do.

emails into tasks. To explain why, let's have a little arithmetic aside.

Let's say it only takes you 45 seconds to go from reading a message to deciding what it really means you should do; so, it only takes 20 seconds to go from looking at the message to remembering what you need to do about it.

This means that by the time you get to 180 unprocessed messages that you need to do something about in your Inbox, you'll be spending an hour a day doing nothing but remembering what those messages mean, before you do anything related to actually living your life, even including checking for new messages.

What should you use for the task list? On some level, this doesn't really matter. It only needs one really important property: you need to trust that if you put something onto it, you'll see it at the appropriate time. How exactly that works depends heavily on your own personal relationship with your computers and devices; it might just be a physical piece of paper. But for most of us living in a multi-device world, something that synchronizes to some kind of cloud service is important, so Wunderlist or Remember the Milk are good places to start, with free accounts.

Turn messages into tasks

The next step — and this is really the first day of the rest of your life — start at the oldest message in your Inbox, and work forward in time. Look at only one message at a time. Decide whether this message is a meaningful task that you should accomplish.

If you decide a message represents a task, then make a new task on your task list. Decide what the task actually is, and describe it in words; don't create tasks like "answer this message." Why do you need to answer it? Do you need to gather any information first?

If you need to access information from the message in order to accomplish the task, then be sure to note in your task how to get back to the email. Depending on what your mail client is, it may be easier or harder to do this[6], but in the worst case, following the guidelines above about eliminating unnecessary folders and filing in your email client, just put a hint in your task list about how to search for the message in question unambiguously.

Once you've done that, archive the message immediately.

The record that you need to do something about the message now lives in your task list, not your email client. You've processed it, and so it should no longer remain in your inbox.

If you decide a message doesn't represent a task, then archive the message immediately.

Do not move on to the next message until you have archived this message. Do not look ahead[7]. The presence of a message in your Inbox means you need to make a decision about it. Follow the touch-move rule with your email. If you skip over messages habitually and decide you'll "just get back to it in a minute," that minute will turn into 4 months and you'll be right back where you were before.

Circling back to the subject of this post; once again, this isn't really specific to email. You should follow roughly the same workflow when someone asks you to do a task in a meeting, or in Slack, or on your Discourse board, or wherever, if you think that the task is actually important enough to do. Note the Slack timestamp and a snippet of the message so you can search for it again, if there is a relevant attachment. The thing that makes email different is really just the presence of an email box.

Banish the blue dot

Almost all email clients have a way of tracking "unread" mes-

sages; they cheerfully display counters of them. Ignore this information; it is useless. Messages have two states: in your inbox (unprocessed) and in your archive (processed). "Read" vs. "Unread" can be, at best, of minimal utility when resuming an interrupted scanning session. But, you are always only ever looking at the oldest message first, right? So none of the messages below it should be unread anyway...

Be ruthless

As you try to start translating your flood of inbound communications into an actionable set of tasks you can actually accomplish, you are going to notice that your task list is going to grow and grow just as your Inbox was before.

This is the hardest step: Decide you are not going to do those tasks, and simply delete them. Sometimes, a task's entire life-cycle is to be created from an email, exist for ten minutes, and then have you come back to look at it and delete it.

This might feel pointless, but in going through that process, you are learning something extremely valuable: you are learning what sort of things are not actually important enough to do.

If every single message you get from some automated system provokes this kind of reaction, that will give you a clue that said system is wasting your time, and just making you feel anxious about work you're never really going to get to, which can then lead to you unsubscribing or filtering messages from that system.

Tasks before messages

To thine own self, not thy Inbox, be true.

Try to start your day by looking at the things you've consciously decided to do. Don't look at your email; don't look at Slack; look at your calendar, and look at your task list. One of those tasks, probably, is a daily reminder to "check your email," but that reminder is there more to remind you to only do it once than to prevent you from forgetting.

I say "try" because this part is always going to be a challenge; while I mentioned earlier that you don't want to unthinkingly give in to availability heuristic, you also have to acknowledge that the reason it's called a "cognitive bias" is because it's part of human cognition.

There will always be a constant anxious temptation to just check for new stuff; for those of us who have a predisposition towards excessive scanning behavior, we have it more than others.

Why email?

We all need to make commitments in our daily lives. We need to do things for other people. And when we make a commitment, we want to be telling the truth. I want you to try to do all these things so you can be better at that.

It's impossible to truthfully make a commitment to spend some time to perform some task in the future if, realistically, you know that all your time in the future will be consumed by whatever the top 3 highest-priority angry voicemails you have on that day are.

Email is a challenging social problem, but I am tired of email, especially the user interface of email applications, getting the blame for what is, at its heart, a problem of interpersonal relations.

It's like noticing that you get a lot of bills through the mail, and then blaming the state of your finances on the colors of the paint in your apartment building's mail room. Of course, the UI of an email app can encourage good or bad habits, but Gmail gave us a prominent "Archive" button a decade ago, and we still have all the same terrible

Start your day by looking at the things you've consciously decided to do.

Email...remains the best medium for communication that allows you to be in control of your own time.

habits that were plaguing Outlook users in the 90s.

Of course, there's a lot more to "productivity" than just making a list of the things you're going to do. Some tools can really help you manage that list a lot better. But all they can help you do is to stop working on the wrong things, and start working on the right ones.

Actually being more productive, in the sense of getting more units of work out of a day, is something you get from keeping yourself healthy, happy, and well-rested, and not from an email filing system.

You can't violate causality to put more hours into the day, and as a frail and finite human being, there's only so much work you can reasonably squeeze in before you die.

The reason I care a lot about salvaging email specifically is that it remains the best medium for communication that allows you to be in control of your own time, and by extension, the best medium for allowing people to do creative work.

Asking someone to do something via SMS doesn't scale; if you have hundreds of unread texts there's no way to put them in order, no way to classify them as "finished" and "not finished", so you need to keep it to the number of things you can fit in short term memory.

Not to mention the fact that text messaging is almost by definition an interruption — by default, it causes a device in someone's pocket to buzz. Asking someone to do something in group chat, such as IRC or Slack, is similarly time-dependent; if they are around, it becomes an interruption, and if they're not around, you have to keep asking and asking over and over again, which makes it really inefficient for the asker (or the asker can use a @highlight, and assume that Slack will send the recipient, guess what, an email).

Social media often comes up as another possible replacement for email, but its sort order is even worse than "only the most recent and most frequently repeated." Messages are instead sorted by value to advertisers or likeliness to increase 'engagement'", i.e. most likely to keep you looking at this social media site rather than doing any real work.

For those of us who require long stretches of uninterrupted time to produce something good — "creatives," or whatever today's awkward buzzword for intersection of writers, programmers, graphic designers, illustrators, and so on, is — we need an inbound task queue that we can have some level of control over.

Something that we can check at a time of our choosing, something that we can apply filtering to in order to protect access to our attention, something that maintains the chain of request/reply for reference when we have to pick up a thread we've had to let go of for a while. Some way to be in touch with our customers, our users, and our fans, without being constantly interrupted.

Because if we don't give those who need to communicate with such a tool, they'll just blast `@everyone` messages into our Slack channels and `@mentions` onto Twitter and texting us `Hey, got a minute?` until we have to quit everything to try and get some work done.

Questions about this post?

Go ahead and send me an email. ■

Acknowledgements

As always, any errors or bad ideas are certainly my own.

First of all, to Merlin Mann, whose writing and podcasting were the inspiration, direct or indirect, for many of my thoughts on this subject; and who sets a good example because he won't answer your email.

Thanks also to David Reid for introducing me to Merlin's work, as well as Alex Gaynor, Tristan Seligmann, Donald Stufft, Cory Benfield, Piët Delport, Amber Brown, and Ashwini Oruganti for feedback on drafts.

Notes

1. Email is so culturally pervasive that it is literally in Vogue, although in fairness this is not a reference to the overflowing-Inbox problem that I'm discussing here.
2. I find the "edit" function in Slack maddening; although I appreciate why it was added, it's easy to retroactively completely change the meaning of an entire conversation in ways that make it very confusing for those reading later. You don't even have to do this intentionally; sometimes you make a legitimate mistake, like forgetting the word "not," and the next 5 or 6 messages are about resolving that confusion; then, you go back and edit, and it looks like your colleagues correcting you are a pedantic version of Mr. Magoo, unable to see that you were correct the first time.
3. There, I said it. Are you happy now?
4. Just to clarify: nothing in this post should be construed as me berating you for not getting more work done, or for ever failing to meet any commitment no matter how casual. Quite the opposite: what I'm saying you need to do is acknowledge that you're going to screw up and rather than hold a thousand emails in your inbox in the vain hope that you won't, just send a quick apology and move on.
5. Maybe you decided to do the thing because your boss asked you to do it and failing to do it would cost you your job, but nevertheless, that is a conscious decision that you are making; not everybody gets to have "boss" priority, and unless your job is a true Orwellian nightmare, not everything your boss says in email is an instant career-ending catastrophe.
6. In Gmail, you can usually just copy a link to the message itself. If you're using OS X's Mail.app, you can use this Python script to generate links that, when clicked, will open the Mail app (see figure below). You can then paste these links into just about any task tracker; if they don't become clickable, you can paste them into Safari's URL bar or pass them to the open command-line tool.
7. The one exception here is that you can look ahead in the same thread to see if someone has already replied.

```
from __future__ import (print_function, unicode_literals,
                        absolute_import, division)

from ScriptingBridge import SBApplication
import urllib

mail = SBApplication.applicationWithBundleIdentifier_("com.apple.mail")

for viewer in mail.messageViewers():
    for message in viewer.selectedMessages():
        for header in message.headers():
            name = header.name()
            if name.lower() == "message-id":
                content = header.content()
                print("message:" + urllib.quote(content))
```

▲ Python code to generate message link in OS X's Mail.app

How to worry less about being a bad programmer

By PETER WELCH

It's...easy to feel inferior when you waste an entire day struggling with a bug before remembering to search Stack Overflow...

I just came across another manifestation of imposter syndrome, in the form of *Am I really a developer or just a good googler?*

The answer I read missed the point, so I'm going to break this mess down, because too many people are afraid for no good reason.

The fact that information is easy to find doesn't make you stupid

This is one of those stories I hear so often I assume it's apocryphal, but fact or fiction, the point stands: When asked for his phone number, Einstein looked it up, saying why should he memorize something he can find in less than two minutes?

In the 80s, the mark of the nerd was owning an encyclopedia. You didn't even have to read most of it: The most impressive encyclopedia in my house was from 1937, and the entry about the Nazi party was two paragraphs implying it was no big deal.

Simply knowing about one of the most incredibly wrong bits of information ever written — which I learned from one of the very things I used to get information — put me in the smart club. Because back then, interesting information was hard to get, and the mere impulse to go find it made you a nerd.

Now that even the most ignorant plebeians can get whatever information they want, the nerd elite have retreated and proclaimed there's some essential brain function that allows them to navigate the information deluge better than everyone else.

As in all the most attractive fallacies, there's a transistor of truth in the notion: It's easy to feel superior to people who use the Internet to look for articles linking vaccines to lizard people.

But it's also easy to feel inferior when you waste an entire day struggling with a bug before remembering to search Stack Overflow, where you discover five people had figured it out three years ago and two of them think anyone who wasn't born knowing the answer is an idiot.

The new populist information retrieval engine may make you feel weak for using something that anyone can use, but that's a terrible elitist emotion you should stamp out along with all your secret homophobia.

Forget all this crap about loving your job

My favorite job of all time was washing dishes. I was good at it, and I could do it on autopilot, and it left my brain free to go braining. The best part? If I looked haggard at the end of the day after cleaning two thousand plates for a four hundred top restaurant, nobody sat me down and asked why I wasn't more enthusiastic about my scrubbing technique.

If loving your job was a nonnegotiable prerequisite for doing it, civilization would collapse. I'm sure somebody finds spiritual satisfaction benchmarking the speed differential between `i++` and `++i` in their for loops, and thank God for them because somebody has to program our nuclear guidance systems.

The rest of us are just praying that the number of unread warnings in the "debug" email folder doesn't start going up so fast that we actually have to deal with it.

The important part about jobs in the old days, before they took away the martini lunches and threw up motivational posters, wasn't that you loved it; the important part was that you didn't hate it and didn't make your coworkers hate it.

Now that they've successfully pedaled softcore happywork porn to a generation desperate for salaried positions, it's okay for our employers to tell us to keep our cell phones handy on Christmas Eve. It's okay for some codehumper to make everybody else hate their job and themselves, because, hey, that guy loves what he does, and if you don't, it's your own fault for not spending Saturday nights masturbating to tail recursion tutorials.

You can't win a perkiness contest against sales

Modern startups call out high-functioning apathy in the worst way. Because of the technology created by the people who truly did love hacking tape-based technology, we have a bunch of companies that are comprised of a sales department and a tech department, because every other job has been outsourced to a website run by another company composed of a sales department and a tech department.

If you're in sales, loving or pretending to love your job is an integral part of that job. That's what makes the sales bucks. If you're in tech, your job is to make something work, and you can be as bitter as you need to be to get that job done, because the only product you're selling is your ability to implement the Stripe api, and nobody has to be aggressively cheerful for that to happen.

All your company meetings consist of attractive and bright-eyed sales people contrasting the tired dev team that wishes the meeting would end because they're already wondering how long it's going to take them to figure out the race condition bug that they know isn't really a race condition because it's never lupus.

You can't worry about this. Maybe you're socially competent, attractive and bright-eyed, maybe you're not. It has nothing to do with your job.

Ignore the pedants

Of course somebody will say, "Every programmer should know X."

I don't know X. For any value. Bubble sort? I assume that has something to do with Guinness and Harp. B-tree? Sounds like an evergreen. Hash table? I learned programming in PHP, so it was two years before I knew a hash table was different from an array. I didn't know the difference between a hash table and an array when OkCupid hired me.

The gods themselves tremble before the judgmentalism of an OkCupid toilet paper dispenser, but they still gave me a job.

No matter what programming job you have, there will be a vast amount of programming you do not understand. If you manage to learn every programming language in the universe, some Russian twelve-year-old will mock you for not knowing how to overclock your CPU.

Simultaneously, a Korean kid will hack your PS4 account while an American sucks down a latte and asks you why you haven't closed a series B. The French ops person just spits at you when you ask her to stop smoking in the server room.

It's kind of cool to be STEM smart now, because a particular application of a particular kind

No matter what programming job you have, there will be a vast amount of programming you do not understand.

of logical problem solving puts the everywhiteman in a position to make an upper middle class income while making the business school graduates richer than anybody in the history of human civilization has ever been.

Since the first bubble generation of CTOs grew up without a lot of sex but with a whole lot of microchips on their shoulders, the culture they accidentally created is a culture obsessing over the ability to apply mathematics in whatever obscure way they thought mathematics should be applied, and everyone else can go kill themselves.

If they needed that, fine. When you're living pre-Google or pre-Vim, you need something to sustain you through the dark hours of discovering your Amiga doesn't remember your anniversary because you don't have an anniversary and probably never will.

Programming is new, and the original John McLanes who had to dig through machine code are still alive and accusing the rest of us of being lazy. But programming is now a job like any other, because everything you need to do to satisfy your BizDev team can be learned without reverse-engineering the prototype for Thag's Move Things Better Octagon.

Interviews are hell, get over it

You will walk into any given interview with what you think of as a cornucopia of arcane knowledge all but forcing its way out of your tear ducts to raise property values in a half mile radius.

Much of the time, you will walk out of that interview wanting to give up and raise guinea pigs for a living. Every human knows things other humans do not, and most of us will eventually be in a position where another human is determining our future employment based on us knowing things very few humans know.

All interview processes are flawed. They will be flawed for as long as we lack an algorithm to predict a candidate's ability to produce work and not be a jerk, based on a smattering of nearly random input.

An interview is a date with fifty thousand dollars on the line and no condom. No matter the profession, it is a waste of arrogance to claim the problem can be fixed if a couple of people think real hard about it.

Exactly no one knows what's going on anymore, but a lot of people are drawing paychecks and clicks by maintaining the illusion that they do. Some of them will interview you, and there's nothing you can do about it. When it starts to give you imposter syndrome, treasure it, because anybody who doesn't have imposter syndrome is a fool.

Make money

Did you get a paycheck last week? If so, good. You're ahead of the curve. Do you work in programming? Yes? Well, last week's paycheck just set you up above the heads of 80 percent of the world's wage earners, to say nothing of people who can't get a job.[1] If you get a paycheck next week, you're not a fraud.

Where you see a mass of rotting spaghetti kludge sending usable energy to a pointless death, your bosses see a black box labelled "Guy Who Speaks Computer Good."[2] They put money in, something happens, and lo! A product emerges that gets them more money.

You may compare yourself to Tesla's wet dreams and wish you were a tenth as prescient as Ada Lovelace, but you shouldn't and you're not. Might as well grow your first beard in eleventh century Norway and assume you're Thor. You're not Thor. You're the poet who stayed on the boat and got to breed because everyone else was dead.

If you get confusing sexual feelings about pointers and 3D graphics equations, power to you: you were born in a generation that respects you directly, and indirectly worships you in a really creepy way. If you just need a job and are able and willing to accept that computers are measurably dumber than lemmings, you have everything you need to keep the information age running. ■

[1] God knows we prefer to say nothing about them.

[2] "WARNING: Do not test for drugs. Dry clean only."

My time with Rails is up

By PIOTR SOLNICA

Last year I made a decision that I won't be using Rails anymore, nor will I support Rails in gems that I maintain. Furthermore, I will do my best to never have to work with Rails again at work.

Since I'm involved with many Ruby projects and people have been asking me many times why I don't like Rails, what kind of problems I have with it and so on, I decided to write this long post to summarize and explain everything.

This is a semi-technical, semi-personal and unfortunately semi-rant. I'm not writing this to bring attention, get visitors or whatever, I have no interest in that at all. I'm writing this because I want to end my discussions about Rails and have a place to refer people to whenever I hear the same kind of questions.

I would also like to tell you a couple of stories that "younger Rails devs" have probably never heard of, and highlight some issues that are important enough to at least think about them.

The good part

I'm not going to pretend that everything about Rails is bad, wrong, evil or damaging. That would not be fair, and is plain stupid. There's a lot of good stuff that you can say about Rails. I'm going to mention a couple of (obvious?) things for good balance.

So, Rails has made Ruby popular. It's a fact. I've become a Ruby developer, which in turn changed my career and gave me a lot of amazing opportunities, *thanks to Rails*. Many Rubyists these days have gone down the same path. We are all here thanks to Rails.

In many cases, Rails actually made tremendous impact on people's lives, making them *much better*. Literally. People got better jobs, better opportunities, and better money. This was a game-changer for many of us.

Over the years Rails & DHH have inspired many people, including people from other communities, to re-think what they're doing. For example, I'm pretty sure Rails has contributed to improvements in the PHP community (you can try to prove

As with everything Rails, it was "nice and easy in the beginning" and then it would turn into unmaintainable crap.

I'm wrong but I have a pretty vivid memory of Symfony framework taking heaps of inspiration from Rails). The same happened in Java (yes), and Play framework is an example.

Now, architecture design issues aside — this was a good thing. Being inspired to think outside of the box and do something new is valuable. Rails contributed to that process on a large scale.

There are other aspects of Rails that are fantastic. Because Rails has always focused on the ease of usage and the ability to quickly build web applications, it made it possible for initiatives like Rails Girls to succeed. Rails has proven to people that they are capable of creating something on their own, without any programming experience, in relatively short time. This *is amazing*, as it can easily become a gateway to the programming world for people who otherwise wouldn't even consider becoming a programmer.

My journey

First of all, let me tell you a little bit about my background and where I'm coming from.

I started working with Ruby in late 2006, as my bachelor thesis was about Rails (yep). I've been learning the language while I was writing my thesis. It was fun, it was exciting, and it was something new for me.

Back then, I was still working as a PHP developer. As a typical PHP developer back in ~ 2005 - 2006, I've done all the typical things — wrote raw sql queries in view templates, choked on procedural PHP to death, then built my own framework, my own ORM, got frustrated and burned out.

Despite knowing some C, C++, Java and Python, I decided to go with Ruby, because of Rails. I picked it up for my thesis and completely accidentally stumbled upon a job offer from a local Rails shop. I applied, and they hired me. It was in March of 2007.

And so since March 2007, I've been working with Ruby professionally, and since roughly 2009 - 2010, I started contributing to Ruby OSS projects. During that time, I worked for a consultancy for 3.5 years, mostly working on big and complicated projects. I then went freelance for a few years, worked with a bunch of clients, started my own company, took a full-time gig, then went back to freelance again, and now I'm a full-time employee again. I built greenfield rails apps and I helped with medium-xxl rails apps.

Let me tell you a story about what can happen in a convoluted Rails codebase. Once, I joined an existing project. It was a *huuuuge* app that was running an online shopping community website. Complicated sales model, complicated promotions, complicated product setups, coupons, user groups, messages — it had it all.

I joined them to help ship a few new features. One of my early tasks was to…add a link to something on some page. It took me a *few days to add this stupid link*. Why? The app was a big ball of complex domain logic scattered across multiple layers with view templates so complicated, it wasn't even simple to find the right template where the link was supposed to be added.

Since I needed some data in order to create that link, it wasn't obvious how I should get it. There was a lack of internal application APIs and relying on ActiveRecord exclusively made it extremely difficult. I am not kidding you.

My initial frustrations with Rails started quite early. I've become displeased with ActiveRecord after roughly the first 6 months of using it. I never liked how Rails approached handling JavaScript and AJAX. In case you don't remember or you were not around already, before Rails adopted UJS approach (which was a big subject in ~ 2007-2008, with blog posts, conference talks and so on), it used inline Javascript generated by a bunch of nasty helpers.

As with everything Rails, it was "nice and easy in the begin-

ning" and then it would turn into unmaintainable crap. Eventually, Rails adopted UJS in the big version 3.0 rewrite and it seems like the community agreed that it's a better approach. This was when Merb was ~~killed by~~ merged into Rails. Oh, you don't know what Merb was? Right, let's talk about that.

Why I was excited about Merb & DataMapper

Merb was a project created by Ezra Zygmuntowicz. It started as a hack to make file uploads faster and thread-safe. It went through an interesting path from that hack to a full-stack, modular, thread-safe, fast web framework. I remember people started talking about it a lot ~2008 and there was this amazing feeling that something new is happening and it's gonna be great.

You might be excited about Rails adding "API mode", right? Well, Merb had 3 modes: a full-stack mode, an API mode and a micro-mode where it was stripped down to bare minimum and I still remember it was the fastest thing ever in Ruby land. *It was over 7 years ago*. Ponder on that.

At the same time, another project brought community attention — DataMapper. It became a part of The Merb Stack, being its ORM of choice. I got really excited about it, as it addressed a lot of issues that ActiveRecord had. DataMapper back in ~ 2008 - 2009 already had attribute definitions in models, custom types, lazy queries, more powerful query DSL and so on.

In 2008, Yehuda Katz was one of the core contributors, and he was actively promoting the project and there was a lot of excitement about it. DataMapper was *ultimately a better ORM than ActiveRecord in 2008-2009*. It would be unfair not to mention that Sequel showed up already around the same time and till this day it's being used way less than ActiveRecord despite being a superior solution.

I was excited about Merb and DataMapper as they brought hope that we can do things better and create healthy competition for Rails. I was excited about it because both projects promoted more modular approach and thread-safety, amongst other things like simply better Ruby coding standards.

Both projects were ultimately killed by Rails as Merb was "merged" into Rails, in what turned out to be a major Rails refactor for its 3.0 version. DataMapper lost its community attention and without much support, it never evolved as it could if Merb was not "merged" into Rails.

With that decision, the Ruby ecosystem lost a bunch of important projects and only Rails benefited from this. Whether the decision to kill Merb was good or not is a matter of personal opinion, we can speculate what could've happened if the decision wasn't made.

However, there's a simple truth about competition — it's healthy. Lack of competition means monopoly, and there's a simple truth about monopoly — it's *not* healthy. Competition fosters progress and innovation, competition creates a healthier ecosystem, it allows people to collaborate more, to share what's common, and to build better foundations. This is not what's happening in the Ruby community.

After Merb & DataMapper were practically destroyed (in the long term), building anything new in the Ruby ecosystem turned out to be extremely difficult., Since peoples' attention is Rails-focused, new projects have been highly influenced by Rails.

Breaking through with new ideas is hard, to say the least, as every time you come up with something, people just want it to be Rails-like and work well with Rails. Making things work with Rails is hard, but I'll get to it later.

After all these years we've ended up with one framework dominating our entire ecosystem, influencing thousands of developers and creating standards that are…questionable. We've lost a diverse ecosystem that started to grow in 2008 and was taken over by Rails.

Hey, I know how this sounds almost like a conspiracy theory, but don't treat it like that. What I've said here are facts with a little bit of my personal feelings. I started contributing to DataMapper in late 2009 and seeing it crumble was very sad.

Complexity!

Complexity is our biggest enemy. People have become less enthusiastic about Rails, because it quickly turned out that dealing with growing complexity leaves us with lots of unanswered questions. What DHH & co. have offered has never been enough to address many issues that thousands of developers started to struggle with already back in ~2007-2008.

Some people hoped that maybe Merb/DataMapper will bring improvements but you

know what happened now, so we were all back using Rails again in 2010, when Rails 3.0 was released.

A couple of days ago somebody posted on /r/ruby a link to an article about organizing your code using "Service Objects". This is one of many articles like that. If you think it's some kind of a recent trend, go take a look at James Golick's article from March 2010 — Crazy, Heretical, and Awesome: The Way I Write Rails Apps.

We've been talking about ways of improving the architecture of our Rails applications for roughly 6 years. I've been trying to contribute to this discussion as much as I could, with articles, conference talks and by working on many OSS projects that strive to solve various hard problems.

The arguments and ideas people have had are always ridiculed by the Rails Core Team members, and especially by DHH. This has been off-putting and discouraging for me, and the reason why I never tried to contribute to Rails. I'm pretty damn sure that my proposals would end up being drastically downvoted.

Monkey-patches? C'mon, not a problem, we love our `10.years.ago`! New abstractions? Who needs that, Rails is SIMPLE! TDD? Not a problem, it's dead, don't bother! ActiveRecord is bloated — so what, it's so handy and convenient, let's add more features instead!

Rails ecosystem, especially around its core team, has never made a good impression on me and I don't have a problem admitting that I'm simply afraid of proposing any changes. This is especially so since the first issue I'd submit would be "Please remove ActiveSupport" (ha-ha… imagine that!).

OK let's get into some tech details.

Rails convenience-oriented design

As I mentioned, Rails has been built with the ease of use in mind. Do not confuse this with simplicity. Just yesterday I stumbled upon this tweet, and it says it all (see figure below).

This is how Rails works, classic example:

```
User.create(params[:user])
```

You see a simple line of code, and you can immediately say (assuming you know User is an AR model) what it's doing. The problem here is that people confuse simplicity with convenience. It's convenient (aka "easy") to write this in your controller and get the job done, right?

Now, this line of code is not simple, it's easy to write it, but the code is extremely complicated under the hood because:

- params must often go through db-specific coercions
- params must be validated
- params might be changed through callbacks, including external systems causing side-effects
- invalid state results in setting error messages, which depends on external system (i.e. I18n)
- valid params must be set as object's state, potentially setting up associated objects too
- a single object or an entire object graph must be stored in the database

This lacks basic separation of concerns, which is always damaging for any complex project. It increases coupling and makes it harder to change and extend code.

But in Rails world, this isn't a problem. In Rails world, basic design guidelines like SRP (and SOLID in general) are being ridiculed and presented as "bloated, unneeded abstractions causing complexity".

When you say you'd prefer to model your application use cases using your own objects and make complex parts explicit, Rails leaders will tell you YAGNI. When you say you'd prefer to

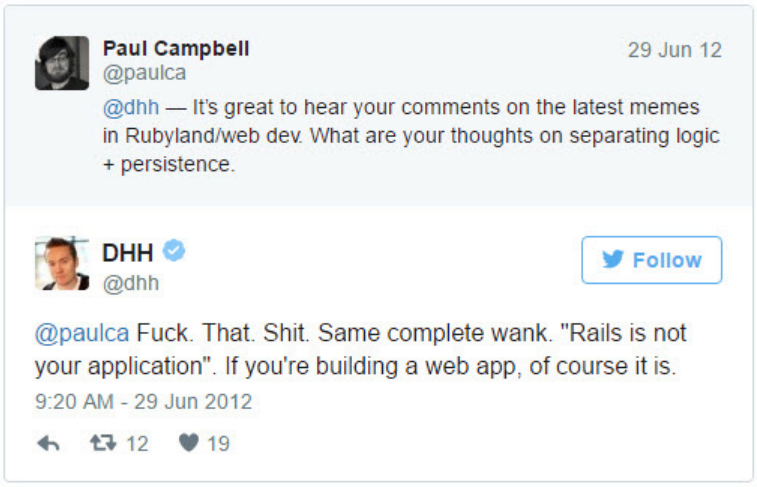
use composition, which makes your code more reliable and flexible, Rails leaders, except tenderlove, will tell you "use `ActiveSupport::Concerns`".

For a Rails developer, it's not a problem that *data coming from a web form are being sent to the depths of ActiveRecord* where God knows what will happen.

The really challenging part in this discussion is being able to explain that *it is a problem* in the first place. *People are attracted by Rails because it gives you a false sense of simplicity*, whereas what really happens is that complexity is being hidden by convenient interfaces. These interfaces are based on many assumptions about how you're gonna build and design your application. ActiveRecord is just one, representative example, but Rails is built with that philosophy in mind, and every piece of Rails works like that.

I should mention that I know there are huge efforts to make ActiveRecord better, like introducing Attributes API (done through some serious internal refactoring which improved the code base). Unfortunately, as long as ActiveRecord comes with over 200 public methods, and encourages the usage of callbacks and concerns, this will always be an ORM that will not be able to handle growing complexity, and it'll only contribute to this complexity and make things worse.

Will *that* change in Rails? I don't think so. We have zero indication that something can be improved as Rails leaders are simply against it. Simple proof is the recent controversial addition, `ActiveRecord.suppress` was proposed by DHH himself. Notice how yet again he makes fun of standalone Ruby classes saying "Yes, you could also accomplish this by having a separate factory for CreateCommentWithNotificationsFactory". Oh boy.



ActiveCoupling

Should Rails be your application? This was an important question asked by many after watching Uncle Bob's talk, where he basically suggests a stronger separation between the web part and your actual core application. Technical details aside, this is good advice, but Rails has not been designed with that in mind. If you're doing it with Rails, you're missing the whole point of this framework. In fact, take a look at what DHH said about this (see figure above).

It's pretty clear what his thoughts are, right? The important part is "of course it is". And you know what? I wholeheartedly agree!

Rails is your application, and it will always be, unless you go through the enormous effort of using it in a way that it wasn't meant to be used.

Think about this:

- *ActiveRecord is meant to become the central part of your domain logic*. That's why it comes with its gigantic interface and plenty of features. You only break things down and extract logic into separate components when it makes sense, but Rails philosophy is to *put stuff to ActiveRecord*, not bother about SRP, not bother about LoD, not bother about tight coupling between domain logic and persistence and so on. That's how you can use Rails effectively.
- The entire view "layer" in Rails is coupled to ActiveModel, thus making it coupled to an Active Record ORM (it could be Sequel, it doesn't matter).
- Controllers, aka your web API endpoints, are the integral part of Rails, tight-coupling takes place here too.
- Helpers, the way you deal with complex templates in Rails, are also an integral part of Rails, tight-coupling once again.

Rails has not been built with loose-coupling and component-oriented design in mind.

Don't fight it. Accept it.

- Everything in Rails, and in a plethora of 3rd party gems built for Rails, is happening through *inheritance (either mixins or class-based)*. Rails and 3rd party gems don't typically provide standalone, reusable objects, they provide abstractions that your objects inherit — this is another form of tight-coupling.

With that in mind, it would be crazy to think that Rails is not your application. If you try to avoid this type of coupling, you can probably imagine what kind of effort it would be and how much of the built-in functionality you'd lose — *and this is exactly why showing alternative approaches in Rails create an impression of bloated, unnecessary abstractions reminding people of their "scary" Java days*.

Rails has not been built with loose-coupling and component-oriented design in mind. Don't fight it. Accept it.

Not a good citizen

Having said all of that, *my biggest beef with Rails is actually ActiveSupport*. Since I ranted about it already, I don't feel like I need to repeat myself. I also recommend going through the comments in the linked blog post.

The only thing I'd like to add is that *because of ActiveSupport, I don't consider Rails to be a good citizen in the Ruby ecosystem*. This library is everything that is wrong with Rails for me. No actual solutions, no solid abstractions, just nasty workarounds to address a problem at hand, *workarounds that turn into official APIs*, and cargo-culted as a result. Gross.

Rails is a closed ecosystem, and it imposes its bad requirements on other projects. If you want to make something work with Rails, you gotta take care of things like making sure it actually works fine when ActiveSupport is required, or that it can work with the atrocious code reloading in development mode, or that objects are being provided as globals because you know, in Rails everything should be available everywhere, for your convenience.

The way Rails works demands a lot of additional effort from developers building their own gems. First of all, it is expected that your gems can work with Rails (because obviously everybody is going to use them with Rails), and that itself is a challenge.

You have a library that deals with databases and you want to make it work with Rails? Well, now you gotta make it work like ActiveRecord, more or less, because the integration interface is ActiveModel, originally based on ActiveRecord prior Rails 3.0.

There are *plenty of constraints* here that make it very hard to provide integration with Rails.

You have no idea how many issues you may face when trying to make things work with hot code reloading in development mode. Rails expects a global, mutable run-time environment. To make it even harder for everybody, they introduced Spring. This gem opened up a whole category of potential new bugs that your gems may face while you try to make them work with Rails.

I'm so done with this, my friends. *Not only is code reloading in Ruby unreliable, but it's also introducing a lot of completely unnecessary complexity to our gems and applications*. This affects everybody who's building gems that are supposed to work with Rails.

Nobody from the Rails Core team, despite the criticism throughout the years, thought that maybe it'd be a good idea to see how it could be done

better. If someone focused on making application code load faster, we could just rely on restarting the process. Besides, you should really use automated testing to see if a change you just made actually works, rather than hitting F5. Just saying.

I know it sounds like complaining, because it is! I've tried to support Rails and it was just too frustrating for me. I've given up, and I don't want to do it anymore.

Since my solution to the problems I've had would mean ditching ActiveSupport, removing Active Record as the pattern of choice, and adding an actual view layer that's decoupled from any ORM, I realized that it's unreasonable to think this will ever happen in Rails.

Leaving Rails

As a result of 9 freaking years of working with Rails and contributing like hell to many Ruby OSS projects, I've given up. I don't believe anything good can happen with Rails. This is my personal point of view, but many people share the same feelings.

At the same time, there's many more who are still happy with Rails. Good for them! Honestly! *Rails is here to stay, it's got its use cases, it still helps people and it's a mature, well maintained, stable web framework*. I'm not trying to convince anybody that Rails is ultimately bad! It's just *really bad for me*.

This decision has had its consequences though. This is why I got involved with dry-rb, hanami and trailblazer projects and why I've been working on rom-rb too. I want to help to build a new ecosystem that will hopefully bring back the same kind of enthusiasm that we all felt when Merb/DataMapper was a thing.

We need a diverse ecosystem, and we need more small, focused, simple and robust libraries. We need Rubyists who feel comfortable using frameworks as well as smaller libraries.

(Sort of) leaving Ruby

Truth is, leaving Rails is also the beginning of my next journey — leaving Ruby as my primary language. I've been inspired by functional programming for the last couple of years. You can see that in the way I write Ruby these days. I'm watching Elixir growing with great excitement. I'm also learning Clojure, which at the moment is on the top of my "languages to learn" list. The more I learn it, the more I love it.

My ultimate goal is to learn Haskell too, as I'm intrigued by static typing. Currently at work, I've been working with Scala. I could very quickly appreciate static typing there, even though it was a rough experience adjusting my development workflow to also include compilation/dealing with type errors steps. It is refreshing to see my editor tell me I'd made a mistake before I even get to running any tests.

The more I learn about functional programming, the more I see how Rails is behind when it comes to modern application design. Monkey-patching, relying on global mutable state, complex ORM, these things are considered major problems in functional languages.

I know many will say "but Ruby is an OO language, use that to your advantage instead of trying to make it what it cannot be" — this is not true. First of all, Ruby is an OO language with functional features (blocks, method objects, lambdas, anyone?).

Secondly, avoiding mutable state is in general, good advice which you can apply in your Ruby code. Ditching global state and isolating it when you can't avoid it is also really good general advice.

Anyhow, I'm leaving Ruby. I've already started the process. It's gonna take years, but that's my direction. I will continue working on and supporting rom-rb, dry-rb, helping with hanami and trailblazer, so don't worry, these projects are very important for me and it makes me very happy seeing the communities grow.

Common feedback/questions

This is a list of made-up feedback and questions, but it's based on actual, real feedback I've been receiving.

> *Shut up. Rails is great and works very well for me.*

This is the most common feedback I receive. First of all, it worries me that many people react like that. We're discussing a tool, not a person, no need to get emotional. Don't get me wrong, I understand that it's natural to "defend" something that helped you and that you simply like it, at the same time it's healthy to be able to think outside the box, and be open to hear criticism and *just think about it*. If you're happy with Rails, that's great, really.

You're just complaining, you're not helping, you haven't done anything to help Rails become better, you haven't suggested any solutions to the problems you're describing.

This type of feedback used to make me very angry and sad. In the moment of writing this, and according to GitHub, I made *2,435 contributions in the last year*. That was in my spare time. Yes, I haven't contributed to Rails directly, because of the reasons I explained in this article. There's too much I disagree with and it would've been a waste of time for both parties. I've been contributing through blog posts, conference talks and thousands of lines of OSS code that you can find on GitHub.

It's OSS, just fork it.

This misses the point completely. We need diversity in the ecosystem with a good selection of libraries, and a couple of frameworks with their unique feature-sets making them suitable for various use cases. A fork of Rails would make no sense. Nobody would fork Rails to go through a struggle like removing ActiveSupport and de-coupling the framework from concepts like Active Record pattern. It's easier and faster to build something new, which other people are already doing (see Hanami).

Just don't use Rails

I did stop using Rails last year, but it's not happening "just like that". Being a Ruby developer means that in 99% of the cases your client work will be Rails. Chances of getting a gig without Rails are close to 0. "Selling" alternative solutions for clients is risky unless you are 100% sure you're gonna be the one maintaining a project for a longer period.

What is happening *right now* is that some businesses, in most of the cases, have two choices: go with Rails or not go with Ruby and pick a completely different technology. People won't be looking at other solutions in Ruby, because they don't feel confident about them and they are not interested in supporting them. I'm talking about common cases here, there are exceptions but they are extremely rare.

OK cool, but what are you suggesting exactly?

My suggestion is to *take a really good look at the current Ruby ecosystem and think about its future.* The moment somebody builds a web framework in a better language than Ruby, that provides similar, fast-pace prototyping capabilities, Rails might become irrelevant for businesses. When that happens, what is it that the Ruby ecosystem has to offer?

If we want Ruby to remain relevant, we need a stronger ecosystem with better libraries and alternative frameworks that can address certain needs better than Rails, so that businesses will continue to consider using Ruby (or keep using Ruby!). We've got over a decade of experience, we've learned so much, and we can use that to our advantage.

You obviously have some personal agenda. I don't trust your judgements.

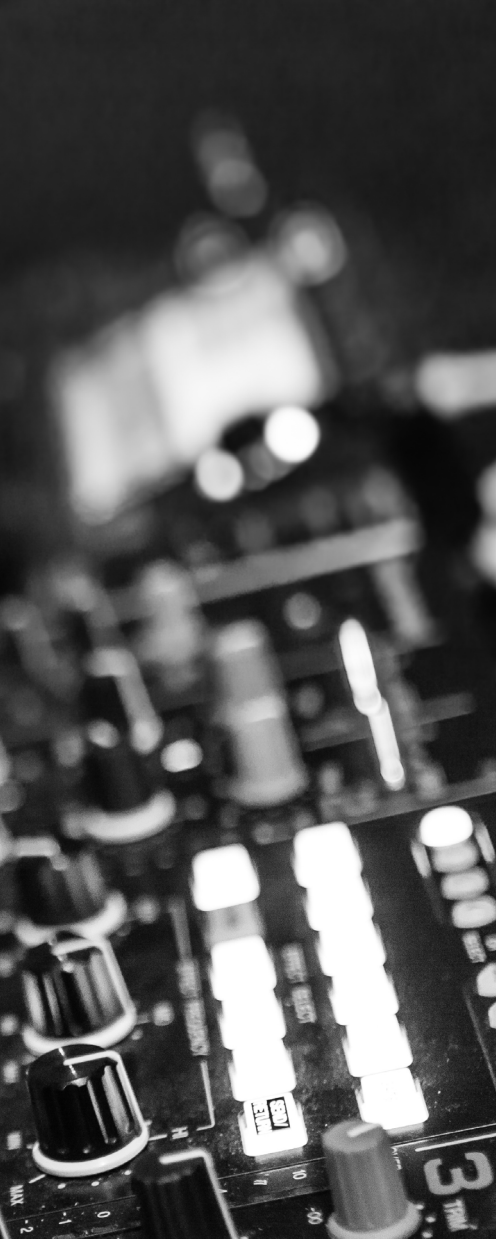
I don't! I've got my OSS projects, I'm working on a book, I have a rom-rb donation campaign and I understand that this creates an impression that I'm simply looking to gain something here.

That's not true, and this is not why I'm doing it. I've been working so hard first and foremost because I enjoy learning, experimenting, collaborating with people, and simply because I care about Ruby's future. The reason why I decided to write a book is because explaining all the new concepts we've introduced in various libraries is close to impossible without a book.

My donation campaign was started because I've invested countless hours into the project and I couldn't continue doing that because I was too busy with client work and, you know, something called life. ■

Programmers are not different, they need simple UIs

By SALVATORE SANFILIPPO



I'm spending days trying to get a couple of APIs right. New APIs about modules, and a new Redis data type.

I really mean it when I say *days*, just for the API. Writing drafts, starting the implementation shaping data structures and calls, and then restarting from scratch to iterate again in a better way, to improve the design and the user-facing part.

Why do I do that, delaying features for weeks? Is it really so important?

Programmers are engineers, maybe they should just adapt to whatever API is better to export for the system exporting it.

Should I really reply to my rhetorical questions? No, it is no longer needed today, and that's a big win.

I want to assume that this point is tacit, taken for granted, that programmers also have user interfaces, and that such user interfaces are so crucial to completely change the perception of a system. Database query languages, libraries calls, programming languages, and Unix command line tools, they all have a user interface part. If you use them daily, to you, they are more UIs than anything else.

So if this is all well known, then why am I here writing this blog post? Because I want to stress how important the concept of simplicity is, not just in graphical UIs, but also in UIs designed for programmers.

The act of iterating again and again to find a simple UI solution is not a form of perfectionism, it's not futile narcissism. It is more an exploration in the design space. It is sometimes huge, made of small variations that make a big difference, and made of big variations that completely change the point of view. There are no rules to follow but your sensibility. Yes there are good practices, but they are not a good compass when the sea to navigate is one of the design *space*.

So why should programmers have this privilege of having good, simple UIs? Sure, there is the joy of using something well made, that is great to handle, that feels *right*. But there is a more central question.

Learning to configure Sendmail via M4 macros, or struggling with an Apache virtual host setup is not real knowledge. If such a system one day is no longer in use, what remains in your hands, or better, in your neurons? Nothing. This is ad hoc knowledge. It is like junk food: empty calories without micronutrients.

For programmers, the micronutrients are the ideas that last for decades, not the ad hoc junk. I don't want to ship junk, so I'll continue to refine my designs before shipping. You should not accept junk, and your neurons are better spent to learn general concepts.

However, in part it is inevitable: every system will have something that is not general that we need to learn in order to use it. Well, if that's the deal, at least, let's make the ad hoc part a simple one, and if possible, something that is even fun to use. ■

How to win the coding interview

By BILL SOUROUR

I've designed and conducted dozens of coding interviews. Now, I'm going to show you how to beat me every single time.

Let's be honest, most developers don't love having to write code as part of an interview process. Some have even threatened to quit the business over it. But it's not going to change any time soon. So, if you're serious about getting a job, you need to understand how to succeed at these interviews. I'm here to help. We're going to go over what I look for in a coding interview and by the end, you should have a pretty good idea of how to succeed.

Before I start though, I have to say, if a company is going to hire a developer based *solely* and *entirely* on a piece of code the developer wrote in an interview, you probably don't want to work there.

Part 1 — Whiteboard coding

Who on Earth writes code on a whiteboard? Like, seriously. But I'm still going to ask you to do it. Don't worry, I haven't lost my mind. I get that Google is a thing and that whiteboards suck at autocomplete. I don't care. I'm not actually testing how well you write code on a whiteboard. I'm looking for something else.

When you get the job, you will *never* have to code on a whiteboard, but I guarantee you this, there will come a time when we are banging our heads against a problem and there is a deadline looming and we're tired and people are pissed at us and money and jobs and reputations are all on the line. When that time comes, we're going to have to get into a boardroom, hop on a whiteboard, and figure things out. Fast.

> *"I'm not actually testing how well you write code on a whiteboard."*

While I don't need a developer who can write code on a whiteboard, I do need a developer who can think on her feet, under pressure, in a room with others. The problem is, if you don't understand what I am actually testing, you're going to approach this task all wrong. You're going to try to prove that you are a whiteboard coding ninja. This is dumb. No one needs a whiteboard coding ninja. Here's how to win me over:

1. Verbalize your assumptions and seek to confirm them.

The best developers know that, fundamentally, every bug is the result of an invalid assumption. So, before you jump in and start coding, think about what assumptions you might be making, and ask me about them.

2. Think out loud.

I want to get some insights into your thought process. Knowing that you understand how to reason about a problem is far more valuable to me than knowing that you've memorized the name of some built-in function. So, think out loud. Say what's on your mind.

3. Don't be afraid to ask for help.

If you're stuck or don't know something, ask me. Do you have any idea how fantastically expensive it is to hire someone who refuses to ask for help when he is stuck? I have no time for a developer who fails to deliver because he pretended he had everything under control while being completely lost and floundering.

4. Represent your skills and experience honestly.

Having said all of the above, I also don't want to mislead you. There is a threshold for questions and commentary. If you are asking me about things that should be obvious to someone who presents with the skills and experience listed on your résumé, that's going to be a red flag. So, before we get to the whiteboard coding, make sure you've been honest in representing your skills and experience to me.

Part 2 — Coding on a computer

Unlike the whiteboard, if I give you a computer and ask you to write code, I *am* testing how well you can code. More specifically, I am testing how well you can code *to spec*.

The best way to understand what to do here is to look at a real world example. One of my favorite questions goes like this:

> *A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward or forward.*
>
> *Allowances may be made for adjustments to capital letters, punctuation, and word dividers. Examples in English include "A man, a plan, a canal, Panama!", "Amor, Roma", "race car", "stack cats", "step on no pets", "taco cat", "put it up", "Was it a car or a cat I*

saw?" and "No 'x' in Nixon".

Write the most efficient function you can that determines whether a given string is a palindrome.

Your function should accept a string as a parameter and return a boolean (true if the string is a palindrome, false if it is not).

Assume that this code will be put into a real production system and write accordingly.

When I offer a challenge like this in an interview, the first thing I'm looking for is whether or not you ask me any questions. As I said before, the best coders understand that assumptions are what kill you in this business. My advice to anyone who is handed instructions and asked to code something is to take a moment and consider what assumptions they will have to make in order to complete the task (there are always assumptions) and then find a way to confirm or clarify those assumptions.

I understand that in an interview situation, people go into "test mode" and feel like they're not allowed to speak. What I suggest is that you start by asking the interviewer "Am I allowed to ask you 1 or 2 questions just to clarify some assumptions?". If the interviewer says "no", then just do your best. If they say "yes" (I would always say "yes") then you have a HUGE advantage.

Good questions for this particular challenge would be:

- "Is this client-side or server-side JavaScript?"
- "For the purposes of this exercise, should an empty string be considered valid input?"
- "Do I need to handle unicode characters?"

The next thing I'm looking for is how well you can follow instructions. For example, I specified a string input parameter and a Boolean output parameter. Is that what you delivered?

After that, I want to see how you interpret the phrase "Assume that this code will be put into a real production system and write accordingly". If you have built real software before, you should take that phrase to mean a few things:

- Your code should be commented.
- You should have error handling or at least logging.
- Your code should avoid breaking at all costs.
- You should have a test harness.
- Your code should be easy-to-read and self-explanatory (clear variable names, good formatting, ideally "lint free" code).

If you have only ever seen code in tutorials and books, you won't know that any of the above is expected. My advice to you is to go look at the code for popular open source projects. Especially projects that have been around a long time and are stable. For JavaScript, the jQuery codebase on GitHub is a pretty good example.

Next, I want to see what you make of the word "efficient" when combined with "production system". If you're experienced, you should know that "efficient" in production code means three things:

1. Runs fast.
2. Doesn't take up more memory than it needs to.
3. Is stable and easy to maintain.

You should understand that #3 sometimes means small sacrifices to #1 and #2.

On this particular challenge, I am expecting many will use RegEx as a part of the solution. The regex needed for this is some of the most basic regex out there, and regex is universal to many languages, and it's fast and extremely handy (*edit: RegEx is not necessarily always fast, thanks AlexDenisov*). It's not unreasonable to expect that you know the basics of RegEx, but you could still write an answer without it.

For tests, I want to see that you included multiple tests, but that each test is testing a truly different scenario. Testing "mom", "dad", and "racecar" is redundant, they are all the same test. I also want to see that you included breaking tests; test for something that is not a palindrome. Consider edge cases, test for null or a number. Test for an empty string, or a bunch of special characters.

I use this test on all levels of developers, but my criteria is stricter the more senior I expect you to be.

For junior devs, if you can produce a working solution that's reasonably straightforward and the rest of the interview goes well, I expect that I'll be able to train you up.

For an intermediate dev, I want to see some comments in

there and good coding style. I want to see an efficient solution and hopefully a test case.

For senior devs, I want an optimal solution, clean, maintainable code, error handling, comments, a full suite of tests. And for bonus points I want you to flag any of your assumptions in the comments. For example, if you use console.log() in client-side JavaScript add a comment that tells me you know there should be server-side logging in production.

Here is an example of a good answer written in JavaScript.

Obviously, there are other ways to write a passing answer, but that should give you an idea.

If I give you a challenge to take home, my expectations are even higher. If you get a week to code something with full access to Google, etc…there's no excuse for giving me a solution that is anything less than top-notch.

Part 3 — Algorithms

Some interviewers will ask you to code an implementation of a particular algorithm. Personally, I think that's just a giant waste of time. It's far more important to me that you understand which algorithm to apply to which problem. You can always Google the implementation.

Nevertheless, because interviewers will ask, you should brush up on the biggies ahead of time. Khan Academy has a great free course.

Part 4 — Passing without solving the problem

If you are unable to solve the problem I give you, there are things you can do to stay in the running for the job.

1. Don't give up too easily

Make sure I see that you've put in a real effort. If you're the type who's going to give up as soon as the going gets tough, I have no time for you.

2. Pseudo-code it

If you're having trouble because you don't recall a certain function name or some other syntactic rule, use comments to explain what you were trying to do in pseudo-code. If I feel like you're just a quick Google search away from solving the problem, it will go a long way toward your cause. Especially if you have an excellent interview otherwise.

3. List your known unknowns

As an absolute "Hail Mary" if you are totally stumped, you can list for me all the things that you know you don't know and describe how, in a real world scenario, you would go about figuring those things out. Be as specific as possible. If you tell me you'd ask for help, tell me who you would ask (the role) and what you would ask them (the specific question, if possible). If you tell me you'd search online, tell me exactly what search strings you would use.

In this scenario, you really need to go out of your way to convince me that you could solve the problem if you were actually working for me.

Part 5 — Practice, practice, practice

Arguably, the most important thing in passing a coding interview is to be well prepared. The best way to do that is to practice common interview coding questions over and over and over again until you know them cold. If you practice enough, and really work at it, you'll even be able to handle a question you've never seen before. You'll have confidence and you'll be able to relate it to something else you've probably tried.

I've put together a massive list of online resources with sample questions and advice for coding in over 50 different languages and technologies; including C#, JavaScript, Mongo, Node, and so on…

You can get the list here.

Web Storage: the lesser evil for session tokens

By JAMES KETTLE

I was recently asked whether it was safe to store session tokens using Web Storage (sessionStorage/localStorage) instead of cookies. Upon googling this, I found the top results nearly all assert that web storage is highly insecure relative to cookies, and therefore not suitable for session tokens. For the sake of transparency, I've decided to publicly document the rationale that led me to the opposite conclusion.

The core argument used against Web Storage says because Web Storage doesn't support cookie-specific features like the Secure flag and the HttpOnly flag, it's easier for attackers to steal it. The path attribute is also cited. I'll take a look at each of these features and try to examine the history of why they were implemented, what purpose they serve and whether they really make cookies the best choice for session tokens.

The secure flag

The secure flag is quite important for cookies, and outright irrelevant for web storage. Web Storage adheres to the Same Origin Policy, which isolates data based on an origin consisting of a protocol and a domain name.

Cookies need the secure flag because they don't properly adhere to the Same Origin Policy — cookies set on *https*://example.com will be transmitted to and accessible via *http*://example.com by default. Conversely, a value stored in localStorage on *https*://example.com will be completely inaccessible to *http*://example.com because the protocols are different.

In other words, cookies are insecure by default, and the secure flag is simply a bodge to make them as resilient to MITM attacks as Web Storage. Web Storage used over HTTPS effectively has the secure flag by default. Further information on related nuances in the Same Origin Policy can be found in The Tangled Web by Michal Zalewski.

The path attribute

The path attribute is widely known to be pretty much useless for security. It's another example of where cookies are out of sync with the Same Origin Policy — paths are not considered part of an origin so there's no security boundary between them. The only way to isolate two applications from each other at the application layer is to place them on separate origins.

Web Storage offers an alternative that, if not secure by default, is less insecure by default.

The HttpOnly flag

The HttpOnly flag is an almost useless XSS mitigation. It was invented back in 2002 to prevent XSS being used to steal session tokens. At the time, stealing cookies may have been the most popular attack vector — it was four years later that CSRF was described as the sleeping giant.

Today, I think any competent attacker will exploit XSS using a custom CSRF payload or drop a BeEF hook. Session token stealing attacks introduce a time-delay and environment shift that makes them impractical and error prone in comparison — see Why HttpOnly Won't Protect You for more background on why. This means that if an attacker is proficient, HttpOnly won't even slow them down. It's like a WAF that's so ineffective attackers don't even notice it exists.

The only instance I've seen where HttpOnly was a significant security boundary is on bugzilla.mozilla.org. Untrusted HTML attachments are served from a subdomain which has access to the parent domain's session cookies thanks to cookies' not-quite-same-origin-policy. Ultimately as with the secure flag, the HttpOnly flag is only really required to make cookies as secure as web storage.

Differences that matter

One major difference between the two options is that unlike cookies, web browsers don't automatically attach the contents of web storage to HTTP requests — you need to write JavaScript to attach the session token to a HTTP header.

This actually conveys a huge security benefit, because it means the session tokens don't act as an ambient authority. This makes entire classes of exploits irrelevant. Browsers' behaviour of automatically attaching cookies to cross-domain requests is what enables attacks like CSRF and cross-origin timing attacks.

There's a specification for yet another another cookie attribute to fix this very problem in development at the moment but for now to get this property, your best bet is Web Storage.

Meanwhile, the unhealthy state of the cookie protocol leads to crazy situations where the cookie header can contain a blend of trusted and untrusted data. This is something the ill-conceived double-submit CSRF defence fell foul of. The solution for this is yet another cookie attribute: Origin.

Unlike cookies, web storage doesn't support automatic expiry. The security impact of this is minimal as expiry of session tokens should be done server-side, but it is something to watch out for. Another distinction is that sessionStorage will expire when you close the tab rather than when you close the browser, which may be awesome or inconvenient depending on your use case. Also, Safari disables Web Storage in private browsing mode, which isn't very helpful.

This post is intended to argue that Web Storage is often a viable and secure alternative to cookies. Web Storage isn't ideal for session token storage in every situation — retrofitting it to a non single-page application may add a significant request overhead, Safari disables Web Storage in private browsing mode, and it's insecure in Internet Explorer 8. Likewise, if you do use cookies, please use both Secure and HttpOnly.

Conclusion

At first glance it looks like cookies have more security features, but they're ultimately patches over a poor core design. For an in depth assessment of cookies, check out HTTP cookies, or how not to design protocols. Web Storage offers an alternative that, if not secure by default, is less insecure by default. ■

Reprinted with permission of the original author. First appeared at blog.portswigger.net.

How to write a git commit message

By CHRIS BEAMS

Credit: xkcd.com

Introduction: Why good commit messages matter

If you browse the log of any random git repository, you will probably find its commit messages are more or less a mess. For example, take a look at these gems from my early days committing to Spring (see Figure 1).

Yikes. Compare that with these more recent commits from the same repository. (see Figure 2)

Which would you rather read?

The former varies wildly in length and form; the latter is concise and consistent. The former is what happens by default; the latter never happens by accident.

While many repositories' logs look like the former, there are exceptions. The Linux kernel and git itself are great examples. Look at Spring Boot, or any repository managed by Tim Pope.

The contributors to these repositories know that a well-crafted git commit message is the best way to communicate *context* about a change to fellow developers (and indeed to their future selves). A diff will tell you *what* changed, but only the commit message can properly tell you *why*. Peter Hutterer makes this point well:

> *Re-establishing the context of a piece of code is wasteful. We can't avoid it completely, so our efforts should go to reducing it [as much] as possible. Commit messages can do exactly that and as a result,* ***a commit message shows whether a developer is a good collaborator****.*

If you haven't given much thought to what makes a great git commit message, it may be the case that you haven't spent much time using `git log` and related tools.

There is a vicious cycle here: because the commit history is unstructured and inconsistent, one doesn't spend much time using or taking care of it. And because it doesn't get used or taken care of, it remains unstructured and inconsistent.

But a well-cared for log is a beautiful and useful thing. `git blame`, `revert`, `rebase`, `log`, `shortlog` and other subcommands come to life. Reviewing others' commits and pull requests becomes some-

thing worth doing, and suddenly can be done independently. Understanding why something happened months or years ago becomes not only possible but efficient.

A project's long-term success rests (among other things) on its maintainability, and a maintainer has few tools more powerful than his project's log. It's worth taking the time to learn how to care for one properly. What may be a hassle at first soon becomes habit, and eventually a source of pride and productivity for all involved.

In this post, I am addressing just the most basic element of keeping a healthy commit history: how to write an individual commit message. There are other important practices like commit squashing that I am not addressing here. Perhaps I'll do that in a subsequent post.

Most programming languages have well-established conventions as to what constitutes idiomatic style, i.e. naming, formatting and so on. There are variations on these conventions, of course, but most developers agree that picking one and sticking to it is far better than the chaos that ensues when everybody does their own thing.

A team's approach to its commit log should be no different. In order to create a useful revision history, teams should first agree on a commit message convention that defines at least the following three things:

Style – markup syntax, wrap margins, grammar, capitalization, punctuation. Spell these things out, remove the guesswork, and make it all as simple as possible. The end result will be a remarkably consistent log that's not only a pleasure to read but that actually *does get read* on a regular basis.

Content – what kind of information should the body of the commit message (if any) contain? What should it *not* contain?

Metadata – how should issue tracking IDs, pull request numbers, etc. be referenced?

Fortunately, there are well-established conventions as to what makes an idiomatic git commit message. Indeed, many of them are assumed in the way certain git commands function. There's nothing you need to re-invent. Just follow the seven rules below and you're on your way to committing like a pro.

```
$ git log --oneline -5 --author cbeams --before "Fri Mar 26 2009"

e5f4b49 Re-adding ConfigurationPostProcessorTests after its brief removal in r814. @Ig-
nore-ing the testCglibClassesAreLoadedJustInTimeForEnhancement() method as it turns out this
was one of the culprits in the recent build breakage. The classloader hacking causes sub-
tle downstream effects, breaking unrelated tests. The test method is still useful, but should
only be run on a manual basis to ensure CGLIB is not prematurely classloaded, and should not
be run as part of the automated build.

2db0f12 fixed two build-breaking issues: + reverted ClassMetadataReadingVisitor to revision
794 + eliminated ConfigurationPostProcessorTests until further investigation determines why
it causes downstream tests to fail (such as the seemingly unrelated ClassPathXmlApplication-
ContextTests)

147709f Tweaks to package-info.java files
22b25e0 Consolidated Util and MutableAnnotationUtils classes into existing AsmUtils
7f96f57 polishing
```

▲ Figure 1: Spring commit messages from early days

```
$ git log --oneline -5 --author pwebb --before "Sat Aug 30 2014"

5ba3db6 Fix failing CompositePropertySourceTests
84564a0 Rework @PropertySource early parsing logic
e142fd1 Add tests for ImportSelector meta-data
887815f Update docbook dependency and generate epub
ac8326d Polish mockito usage
```

▲ Figure 2: More recent Spring commit messages

The seven rules of a great git commit message

Keep in mind: This has all been said before.

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how

For example, see Figure 3.

1. Separate subject from body with a blank line

From the `git commit` manpage:

> *Though not required, it's a good idea to begin the commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description. The text up to the first blank line in a commit message is treated as the commit title, and that title is used throughout Git. For example, git-format-patch(1) turns a commit into email, and it uses the title on the Subject line and the rest of the commit in the body.*

Firstly, not every commit requires both a subject and a body. Sometimes a single line is fine, especially when the change is so simple that no further context is necessary. For example:

```
Fix typo in introduction to user guide
```

Nothing more need be said; if the reader wonders what the typo was, she can simply take a look at the change itself, i.e. use `git show` or `git diff` or `git log -p`.

If you're committing something like this at the command line, it's easy to use the `-m` switch to `git commit`:

```
$ git commit -m"Fix typo in introduction to
user guide"
```

However, when a commit merits a bit of explanation and context, you need to write a body. For example, see Figure 4.

This is not so easy to commit this with the -m switch. You really need a proper editor. If you do not already have an editor set up for use with git at the command line, read this section of Pro Git.

In any case, the separation of subject from body pays off when browsing the log. Here's the

```
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequenses of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded
   by a single space, with blank lines in between, but conventions
   vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: #123
See also: #456, #789
```

▲ Figure 3: Example of a great git commit message

full log entry in Figure 5.

And now `git log --oneline`, which prints out just the subject line:

```
$ git log --oneline
42e769 Derezz the master control program
```

Or, `git shortlog`, which groups commits by user, again showing just the subject line for concision:

```
$ git shortlog
Kevin Flynn (1):
      Derezz the master control program

Alan Bradley (1):
      Introduce security program "Tron"

Ed Dillinger (3):
      Rename chess program to "MCP"
      Modify chess program
      Upgrade chess program

Walter Gibbs (1):
      Introduce protoype chess program
```

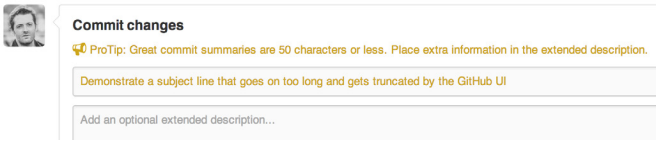
There are a number of other contexts in git where the distinction between subject line and body kicks in — but none of them work properly without the blank line in between.

2. Limit the subject line to 50 characters

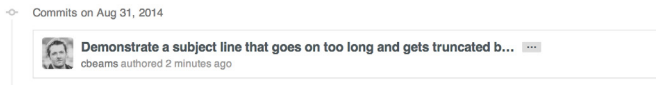
50 characters is not a hard limit, just a rule of thumb. Keeping subject lines at this length ensures that they are readable, and forces the author to think for a moment about the most concise way to explain what's going on.

Tip: If you're having a hard time summarizing, you might be committing too many changes at once. Strive for atomic commits (a topic for a separate post).

GitHub's UI is fully aware of these conventions. It will warn you if you go past the 50 character limit:



And will truncate any subject line longer than 69 characters with an ellipsis:



So shoot for 50 characters, but consider 69 the hard limit.

3. Capitalize the subject line

This is as simple as it sounds. Begin all subject lines with a capital letter.
For example:

```
Accelerate to 88 miles per hour
```

Instead of:

```
accelerate to 88 miles per hour
```

4. Do not end the subject line with a period

Trailing punctuation is unnecessary in subject

```
Derezz the master control program

MCP turned out to be evil and had become intent on world domination.
This commit throws Tron's disc into MCP (causing its deresolution)
and turns it back into a chess game.
```

▲ Figure 4: Some commit messages merit a body of explanation and context

```
$ git log
commit 42e769bdf4894310333942ffc5a15151222a87be
Author: Kevin Flynn <kevin@flynnsarcade.com>
Date:   Fri Jan 01 00:00:00 1982 -0200

 Derezz the master control program

 MCP turned out to be evil and had become intent on world domination.
 This commit throws Tron's disc into MCP (causing its deresolution)
 and turns it back into a chess game.
```

▲ Figure 5: Full log entry

lines. Besides, space is precious when you're trying to keep them to 50 chars or less.

Example:

```
Open the pod bay doors
```

Instead of:

```
Open the pod bay doors.
```

5. Use the imperative mood in the subject line

Imperative mood just means "spoken or written as if giving a command or instruction". A few examples:

- Clean your room
- Close the door
- Take out the trash

Each of the seven rules you're reading about right now are written in the imperative ("Wrap the body at 72 characters", etc.).

The imperative can sound a little rude; that's why we don't often use it. But it's perfect for git commit subject lines. One reason for this is that *git itself uses the imperative whenever it creates a commit on your behalf*.

For example, the default message created when using `git merge` reads:

```
Merge branch 'myfeature'
```

And when using `git revert`:

```
Revert "Add the thing with the stuff"

This reverts commit cc87791524aedd593cf-
f5a74532befe7ab69ce9d.
```

Or when clicking the "Merge" button on a GitHub pull request:

```
Merge pull request #123 from someuser/some-
branch
```

So when you write your commit messages in the imperative, you're following git's own built-in conventions. For example:

- Refactor subsystem X for readability
- Update getting started documentation
- Remove deprecated methods
- Release version 1.0.0

Writing this way can be a little awkward at first. We're more used to speaking in the indicative mood, which is all about reporting facts. That's why commit messages often end up reading like this:

- Fixed bug with Y
- Changing behavior of X

And sometimes commit messages get written as a description of their contents:

- More fixes for broken stuff
- Sweet new API methods

To remove any confusion, here's a simple rule to get it right every time.

A properly formed git commit subject line should always be able to complete the following sentence:

- If applied, this commit will *your subject line here*

For example:

- If applied, this commit will refactor subsystem X for readability
- If applied, this commit will update getting started documentation
- If applied, this commit will remove deprecated methods
- If applied, this commit will release version 1.0.0
- If applied, this commit will merge pull request #123 from user/branch

Notice how this doesn't work for the other non-imperative forms:

- If applied, this commit will fixed bug with Y
- If applied, this commit will changing behavior of X
- If applied, this commit will more fixes for broken stuff
- If applied, this commit will sweet new API methods

Remember: Use of the imperative is important only in the subject line. You can relax this restriction when you're writing the body.

6. Wrap the body at 72 characters

Git never wraps text automatically. When you write the body of a commit message, you must mind its right margin, and wrap text manually.

The recommendation is to do this at 72 characters, so that git has plenty of room to indent

text while still keeping everything under 80 characters overall.

A good text editor can help here. It's easy to configure Vim, for example, to wrap text at 72 characters when you're writing a git commit. Traditionally, however, IDEs have been terrible at providing smart support for text wrapping in commit messages (although in recent versions, IntelliJ IDEA has finally gotten better about this).

7. Use the body to explain what and why vs. how

This commit from Bitcoin Core is a great example of explaining what changed and why. (see Figure 6)

Take a look at the full diff and just think how much time the author is saving fellow and future committers by taking the time to provide this context here and now. If he didn't, it would probably be lost forever.

In most cases, you can leave out details about how a change has been made. Code is generally self-explanatory in this regard (and if the code is so complex that it needs to be explained in prose, that's what source comments are for). Just focus on making clear the reasons you made the change in the first place — the way things worked before the change (and what was wrong with that), the way they work now, and why you decided to solve it the way you did.

The future maintainer that thanks you may be yourself!

Tips

Learn to love the command line. Leave the IDE behind.

For as many reasons as there are git subcommands, it's wise to embrace the command line. Git is insanely powerful; IDEs are too, but each in different ways. I use an IDE every day (IntelliJ IDEA) and have used others extensively (Eclipse), but I have never seen IDE integration for git that could begin to match the ease and power of the command line (once you know it).

Certain git-related IDE functions are invaluable, like calling `git rm` when you delete a file, and doing the right stuff with `git` when you rename one. Where everything falls apart is when you start trying to commit, merge, rebase, or do sophisticated history analysis through the IDE.

When it comes to wielding the full power of git, it's command-line all the way.

Remember that whether you use Bash or Z shell, there are tab completion scripts that take much of the pain out of remembering the subcommands and switches.

Read Pro Git

The Pro Git book is available online for free, and it's fantastic. Take advantage! ■

```
commit eb0b56b19017ab5c16c745e6da39c53126924ed6
Author: Pieter Wuille <pieter.wuille@gmail.com>
Date:   Fri Aug 1 22:57:55 2014 +0200

    Simplify serialize.h's exception handling

    Remove the 'state' and 'exceptmask' from serialize.h's stream
    implementations, as well as related methods.

    As exceptmask always included 'failbit', and setstate was always
    called with bits = failbit, all it did was immediately raise an
    exception. Get rid of those variables, and replace the setstate
    with direct exception throwing (which also removes some dead
    code).

    As a result, good() is never reached after a failure (there are
    only 2 calls, one of which is in tests), and can just be replaced
    by !eof().

    fail(), clear(n) and exceptions() are just never called. Delete
    them.
```

▲ Figure 6: Commit from Bitcoin Core

food bit *

Baker's dozen

Ever wondered why a baker's dozen refers to 13 of something? Turns out for as long as there have been bakeries, there have been deceitful bakers who cheated their customers by selling lighter-than-usual loaves.

To combat this scourge of fraudulent baked goods, laws were passed to protect consumers and anyone caught passing off a lightweight loaf risked severe punishment — in ancient Babylon that meant losing an arm. Over the centuries, bakers (armed with uncanny marketing skills) began selling what they called a "baker's dozen" to highlight their trustworthiness. Their buy-12-get-1-free sales pitch worked and the rest, as they say, is history.

* FOOD BIT is where we, enthusiasts of all edibles, sneak in a fun fact about food.