Essays on programming I think about a lot

JULY 2020

Every so often I read an essay that I end up thinking about, and citing in conversation, over and over again.

Here's my index of all the ones of those I can remember! I'll try to keep it up to date as I think of more.

⁂

Nelson Elhage, Computers can be understood. The attitude embodied in this essay is one of the things that has made the biggest difference to my effectiveness as an engineer:

> I approach software with a deep-seated belief that computers and software systems can be understood. …
>
> In some ways, this belief feels radical today. Modern software and hardware systems contain almost unimaginable complexity amongst many distinct layers, each building atop each other. …
>
> In the face of this complexity, it's easy to assume that there's just too much to learn, and to adopt the mental shorthand that the systems we work with are best treated as black boxes, not to be understood in any detail.
>
> I argue against that approach. You will never understand every detail of the implementation of every level on that stack; but you can understand all of them to some level of abstraction, and any specific layer to essentially any depth necessary for any purpose.

⁂

Dan McKinley, Choose Boring Technology. When people ask me how we make technical decisions at Wave, I send them this essay. It's probably saved me more heartbreak and regret than any other:

> Let's say every company gets about three innovation tokens. You can spend these however you want, but the supply is fixed for a long while. You might get a few more after you achieve a certain level of stability and maturity, but the general tendency is to overestimate the contents of your wallet. Clearly this model is approximate, but I think it helps.
>
> If you choose to write your website in NodeJS, you just spent one of your innovation tokens. If you choose to use MongoDB, you just spent one of your innovation tokens. If you choose to use service discovery tech that's existed for a year or less, you just spent one of your innovation tokens. If you choose to write your own database, oh god, you're in trouble.

⁂

Sandy Metz, The Wrong Abstraction. This essay convinced me that "don't repeat yourself" (DRY) isn't a good motto. It's okay advice, but as Metz points out, if you don't choose the right interface boundaries when DRYing up, the resulting abstraction can quickly become unmaintainable:

> 4. Time passes.

5. A new requirement appears for which the current abstraction is almost perfect.
6. Programmer B gets tasked to implement this requirement.

   Programmer B feels honor-bound to retain the existing abstraction, but since isn't exactly the same for every case, they alter the code to take a parameter….

7. … Loop until code becomes incomprehensible.
8. You appear in the story about here, and your life takes a dramatic turn for the worse.

⁂

Patrick McKenzie, Falsehoods Programmers Believe About Names. When programming, it's helpful to think in terms of "invariants," i.e., properties that we assume will always be true. I think of this essay as the ultimate reminder that *reality has no invariants*:

32. People's names are assigned at birth.
33. OK, maybe not at birth, but at least pretty close to birth.
34. Alright, alright, within a year or so of birth.
35. Five years?
36. You're kidding me, right?

⁂

Thomas Ptacek, The Hiring Post. This essay inspired me to put a lot of effort into Wave's work-sample interview, and the payoff was huge—we hired a much stronger team, much more quickly, than I expected to be able to. It's also a good reminder that most things that most people do make no sense:

> Nothing in Alex's background offered a hint that this would happen. He had Walter White's resume, but Heisenberg's aptitude. None of us saw it coming. My name is Thomas Ptacek and I endorse this terrible pun. Alex was the one who nonced.
>
> A few years ago, Matasano couldn't have hired Alex, because we relied on interviews and resumes to hire. Then we made some changes, and became a machine that spotted and recruited people like Alex: line of business .NET developers at insurance companies who pulled Rails core CVEs out of their first hour looking at the code. Sysadmins who hardware-reversed assembly firmware for phone chipsets. Epiphany: the talent is out there, but you can't find it on a resume.
>
> Our field selects engineers using a process that is worse than reading chicken entrails. Like interviews, poultry intestine has little to tell you about whether to hire someone. But they're a more pleasant eating experience than a lunch interview.

⁂

Gergely Orosz, The Product-Minded Engineer. I send this essay to coworkers all the time—it describes extremely well what traits will help you succeed as an engineer at a startup:

> Proactive with product ideas/opinions • Interest in the business, user behavior and data on this • Curiosity and a keen interest in "why?" • Strong communicators and great relationships with non-engineers • Offering product/engineering tradeoffs upfront • Pragmatic handling of edge cases • Quick product validation cycles • End-to-end product feature ownership • Strong product instincts through repeated cycles of learning

⁂

tef, Write code that is easy to delete, not easy to extend. *The Wrong Abstraction* argues that reusable code, unless carefully designed, becomes unmaintainable. tef takes the logical next step: design for disposability, not maintainability. This essay gave me lots of useful mental models for evaluating software designs.

> If we see 'lines of code' as 'lines spent', then when we delete lines of code, we are lowering the cost of maintenance. Instead of building re-usable software, we should try to build disposable software.

> Business logic is code characterised by a never ending series of edge cases and quick and dirty hacks. This is fine. I am ok with this. Other styles like 'game code', or 'founder code' are the same thing: cutting corners to save a considerable amount of time.
>
> The reason? Sometimes it's easier to delete one big mistake than try to delete 18 smaller interleaved mistakes. A lot of programming is exploratory, and it's quicker to get it wrong a few times and iterate than think to get it right first time.

tef also wrote a follow-up, Repeat yourself, do more than one thing, and rewrite everything, that he thinks makes the same points more clearly—though I prefer the original because "easy to delete" is a unifying principle that made the essay hang together really well.

⁂

Joel Spolsky, The Law of Leaky Abstractions. Old, but still extremely influential—"where and how does this abstraction leak" is one of the main lenses I use to evaluate designs:

> Back to TCP. Earlier for the sake of simplicity I told a little fib, and some of you have steam coming out of your ears by now because this fib is driving you crazy. I said that TCP guarantees that your message will arrive. It doesn't, actually. If your pet snake has chewed through the network cable leading to your computer, and no IP packets can get through, then TCP can't do anything about it and your message doesn't arrive. If you were curt with the system administrators in your company and they punished you by plugging you into an overloaded hub, only some of your IP packets will get through, and TCP will work, but everything will be really slow.
>
> This is what I call a leaky abstraction. TCP attempts to provide a complete abstraction of an underlying unreliable network, but sometimes, the network leaks through the abstraction and you feel the things that the abstraction can't quite protect you from. This is but one example of what I've dubbed the Law of Leaky Abstractions:
>
> **All non-trivial abstractions, to some degree, are leaky.**
>
> Abstractions fail. Sometimes a little, sometimes a lot. There's leakage. Things go wrong. It happens all over the place when you have abstractions. Here are some examples.

⁂

Reflections on software performance by Nelson Elhage, the only author of two different essays in this list! Nelson's ideas helped crystallize my philosophy of tool design, and contributed to my views on impatience.

> It's probably fairly intuitive that users prefer faster software, and will have a better experience performing a given task if the tools are faster rather than slower.
>
> What is perhaps less apparent is that having faster tools changes how users use a tool or perform a task. Users almost always have multiple strategies available to pursue a goal — including deciding to work on something else entirely — and they will choose to use faster tools more and more frequently. Fast tools don't just allow users to accomplish tasks faster; they allow users to accomplish entirely new types of tasks, in entirely new ways. I've seen this phenomenon clearly while working on both Sorbet and Livegrep…

⁂

Brandur Leach's series on using databases to ensure correct edge-case behavior: Building Robust Systems with ACID and Constraints, Using Atomic Transactions to Power an Idempotent API, Transactionally Staged Job Drains in Postgres, Implementing Stripe-like Idempotency Keys in Postgres.

Normally, article titles ending with "in [technology]" are a bad sign, but not so for Brandur's. Even if you've never used Postgres, the examples showing how to lean on relational databases to enforce correctness will be revelatory.

> I want to convince you that ACID databases are one of the most important tools in existence for ensuring maintainability and data correctness in big production systems. Lets start by digging into each of their namesake guarantees.

> There's a surprising symmetry between an HTTP request and a database's transaction. Just like the transaction, an HTTP request is a transactional unit of work – it's got a clear beginning, end, and result. The client generally expects a request to execute atomically and will behave as if it will (although that of course varies based on implementation). Here we'll look at an example service to see how HTTP requests and transactions apply nicely to one another.

> In APIs *idempotency* is a powerful concept. An idempotent endpoint is one that can be called any number of times while guaranteeing that the side effects will occur only once. In a messy world where clients and servers that may occasionally crash or have their connections drop partway through a request, it's a huge help in making systems more robust to failure. Clients that are uncertain whether a request succeeded or failed can simply keep retrying it until they get a definitive response.

⁂

Jeff Hodges, Notes on Distributed Systems for Young Bloods. An amazing set of guardrails for doing reasonable things with distributed systems (and note that, though you might be able to get away with ignoring it for a while, any app that uses the network is a distributed system). Many points would individually qualify for this list if they were their own article—I reread it periodically and always notice new advice that I should have paid more attention to.

> Distributed systems are different because they fail often • Implement backpressure throughout your system • Find ways to be partially available • Use percentiles, not averages • Learn to estimate your capacity • Feature flags are how infrastructure is rolled out • Choose id spaces wisely • Writing cached data back to persistent storage is bad • Extract services.

⁂

J.H. Saltzer, D.P. Reed and D.D. Clark, End-to-End Arguments in System Design. Another classic. The end-to-end principle has helped me make a lot of designs much simpler.

> This paper presents a design principle that helps guide placement of functions among the modules of a distributed computer system. The principle, called the end-to-end argument, suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level. Examples discussed in the paper include bit error recovery, security using encryption, duplicate message suppression, recovery from system crashes, and delivery acknowledgement. Low level mechanisms to support these functions are justified only as performance enhancements.

⁂

Bret Victor, Inventing on Principle:

> I've spent a lot of time over the years making creative tools, using creative tools, thinking about them a lot, and here's something I've come to believe: Creators need an immediate connection to what they're creating.

I can't really excerpt any of the actual demos, which are the good part. Instead I'll just endorse it: this talk dramatically, and productively, raised my bar for what I think programming tools (and tools in general) can be. Watch it and be amazed.

⁂

Post the essays you keep returning to in the comments!

Related

10x (engineer, context) pairs

Your actual output depends on a lot more than just how quickly you finish a given programming task. Everything besides the literal coding depends deeply on the way you interact with the organization around you.

What I've been doing instead of writing

I've been too busy with work to write much recently, but in lieu of that, here's a batch of links to other stuff I've been doing elsewhere. The thing I'm most excited about: Wave raises $200m from Sequoia, Stripe, Founders Fund and Ribbit at a $1.7b valuation. It'll fund faster expansion across Africa. I'm pumped for us to save tons of money + time for even more people! One of our investors for Founders Fund wrote an amazing deep dive into our vision and strategy.

My favorite essays of life advice

Life is short • There is no speed limit • How to Be Successful • You and your research • Becoming a Magician • 95th percentile isn't that good

Comments

Bryan

JULY 2020

Quite a few of these are on my list, here's some others that I keep returning to every so often:

- https://www.stilldrinking.org/programming-sucks
- https://medium.com/@nicolopigna/this-is-not-the-dry-you-are-looking-for-a316ed3f445f
- https://sysadvent.blogspot.com/2019/12/day-21-being-kind-to-3am-you.html
- https://jeffknupp.com/blog/2014/05/30/you-need-to-start-a-whizbang-project-immediately/

▷ REPLY

zindlerb

JULY 2020

Great list! Some essays I end up returning to are:

- https://www.destroyallsoftware.com/compendium/software-structure?share_key=6fb5f711cae5a4e6
- https://caseymuratori.com/blog_0015

▷ REPLY

Max
JULY 2020

These are conference talks on youtube, not blog posts, but here's a few of the ones I often end up sending to collaborators as addenda to discussions:

- Don Reinertsen - Second Generation Lean Product Development Flow
- Joshua Bloch
- The Language of the System - Rich Hickey

Some posts:

- https://speakerdeck.com/vjeux/react-css-in-js - diagnosis of problems with CSS (not because of React)
- https://zachholman.com/talk/firing-people

▷ REPLY

Karsten
JULY 2020

Especially for fault-tolerant systems, "why restart helps" really opened my eyes:

- https://ferd.ca/the-zen-of-erlang.html

▷ REPLY

Karsten
JULY 2020

Oh, I forgot: http://web.mit.edu/2.75/resources/random/How%20Complex%20Systems%20Fail.pdf

▷ REPLY

Doctor_eval
JULY 2020

Oldie but a goodie:

https://www.developerdotstar.com/mag/articles/reeves_design_main.html

LuckyBee
JULY 2020

+1 for that one

▷ REPLY

Andreas
JULY 2020

This is a great list. If i could make one addition it would have to be Rich Hickey's "simple made easy": https://www.youtube.com/watch?v=oytL881p-nQ

I was once working with a newly formed (4 person) team on a large and complex project under a tight deadline. For a while we weren't seeing eye to eye on many of the key decisions we made. Watching and reflecting on this talk gave us a shared aim and, perhaps even more importantly, a shared language for making choices that would reduce the complexity of our system. It is a gift that keeps on giving.

▷ REPLY

Dewald
JULY 2020

Another one that belongs on this list: https://www.kitchensoap.com/2012/10/25/on-being-a-senior-engineer/

▷ REPLY

Doodpants
JULY 2020

A couple of my favorites:

- https://nedbatchelder.com/text/deleting-code.html
- https://www.joelonsoftware.com/2002/01/23/rub-a-dub-dub/

▷ REPLY

Peter H
JULY 2020

Out of the Tar Pit. https://github.com/papers-we-love/papers-we-love/blob/master/design/out-of-the-tar-pit.pdf

▷ REPLY

Jod
JULY 2020

I'd like to nominate another of Nelson Elhage's posts:

- https://blog.nelhage.com/2016/03/design-for-testability

This has had more direct influence on my day-to-day code writing than anything else. (Also, his other writing on testing is great.)

As another commenter mentioned conference talks, Bryan Cantrill on debugging is important—it meshes well with Nelson's *Computer can be understood*. (https://www.slideshare.net/bcantrill/debugging-microservices-in-production)

▷ REPLY

Daniel Sickles
JULY 2020

A fave of mine: Clojure: Programming with Hand Tools https://www.youtube.com/watch?v=ShEez0JkOFw

▷ REPLY

visilii
MAY 2024

Some essays I like:

- Science and the compulsive programmer by Joseph Weizenbaum - written in 1976, but the described phenomena of a compulsive programmer still exists and may be relevant to many: https://www.sac.edu/academicprogs/business/computerscience/pages/hester_james/hacker.htm

- https://www.mit.edu/~xela/tao.html - Tao of Programming - not sure if you can classify as an essay, but it is classic!
- https://norvig.com/21-days.html - Teach Yourself Programming in Ten Years by Peter Novig - a great essay on how to master programming and why reading books like “Learn X in Y days” won’t be of much help. I recommend it to all beginners
- Reginald Braithwaite, Golf is a good program spoiled - http://weblog.raganwald.com/2007/12/golf-is-good-program-spoiled.html. Raganwald has more great essays on his weblog, I just like this one the most.

  ▷ REPLY