Is it real or is it

apenwarr

Everything here is my opinion. I do not speak for your employer.

← *November 2020* *October 2021* →

2020-12-27 »

Systems design explains the world: volume 1

"Systems design" is a branch of study that tries to find universal architectural patterns that are valid across disciplines.

You might think that's not a possibility. Back in university, students used to tease the Systems Design Engineers, calling it "boxes and arrows" engineering. Not real engineering, you see, since it didn't touch anything tangible, like buildings, motors, hydrochloric acid, or, uh, electrons.

I don't think the Systems Design people took this criticism too seriously since everyone also knew that programme had the toughest admittance criteria in the whole university.

(A mechanical engineer told me they saw electrical/computer engineers the same way: waveforms on a screen instead of real physical things that you could touch, change, and fix.)

I don't think any of us really understood what boxes-and-arrows engineering really was back then, but luckily for you, now I'm old. Let me tell you some stories.

What is systems design?

I started thinking more clearly about systems design when I was at a big tech company and helped people refine their self-promotion employee review packets. Most of it was straightforward, helping them map their accomplishments to the next step up the engineering ladder:

- As a Novice going for Junior, you had to prove you could fix bugs without too much supervision;
- Going for Senior, you had to prove you could implement a whole design with little supervision;
- Going for Staff, you had to show you could produce designs based on business problems with basically no management;
- Going for Senior Staff, you had to solve bigger and bigger business problems; and so on.

After helping a few dozen people with their assessments, I noticed a trend. *Most* developers mapped well onto the ladder, but some didn't fit, even though they seemed like great engineers to me.

There were two groups of misfits:

1. People who maxed out as a senior engineer (building things) but didn't seem to want to, or be able to, make it to staff engineer (translating business problems).

2. People who were ranked at junior levels, but were better at translating business problems than at fixing bugs.

Group #1 was formally accounted for: the official word was most employees should never expect to get past Senior Engineer. That's why they called it Senior. It wasn't not much consolation to people who wanted to earn more money or to keep improving for the next 20-30 years of a career, but it was something we could talk about.

(The book Radical Candor by Kim Scott has some discussion about how to handle great engineers who just want to build things. She suggests a separate progression for "rock solid" engineers, who want to become world-class experts at things they're great at, and "steep trajectory" engineers, who might have less attention to detail but who want to manage ever-bigger goals and jump around a lot.)

People in group #2 weren't supposed to exist. They were doing some hard jobs - translating business problems into designs - with great expertise, but these accomplishments weren't interesting to the junior-level promotion committees, who had been trained to look for "exactly one level up" attributes like deep technical knowledge in one or two specific areas, a history of rapid and numerous bug fixes, small independent launches, and so on. Meanwhile, their peers who couldn't (yet) architect their way out of a paper bag rose more quickly through the early ranks, because they wrote reams of code fast.

Tanya Reilly has an excellent talk (and transcribed slides) called Being Glue that perfectly captures this effect. In her words: "Glue work is expected when you're senior... and risky when you're not."

What she calls glue work, I'm going to call systems design. They're two sides of the same issue. Humans are the most unruly systems of all, and yet, amazingly, they follow many of the same patterns as other systems.

People who are naturally excellent at glue work often stall out early in the prescribed engineering pipeline, even when they'd be great in later stages (staff engineers, directors, and executives) that traditional engineers struggle at. In fact, it's well documented that an executive in a tech company requires almost a totally different skill set than a programmer, and rising through the ranks doesn't prepare you for that job at all. Many big tech companies hire executives from outside the company, and sometimes even from outside their own industry, for that reason.

...but I guess I still haven't answered the question. What is systems design? It's the thing that will eventually kill your project if you do it wrong, but probably not right away. It's macroeconomics instead of microeconomics. It's fixing which promotion ladders your company even has, rather than trying to climb the ladders. It's knowing when a distributed system is or isn't appropriate, not just knowing how to build one. It's repairing the incentives in a political system, not just getting elected and passing your favourite laws.

Most of all, systems design is invisible to people who don't know how to look for it. At least with code, you can measure output by the line or the bug, and you can hire more programmers to get more code. With systems design, the key insight might be a one-sentence explanation given at the right time to the right person, that affects the next 5 years of work, or is the difference between hypergrowth and steady growth.

Sorry, I don't know how to explain it better than that. What I can do instead is talk about some systems design problems and archetypes that repeat, over and over, across multiple fields. If you can recognize these archetypes, and handle them before they kill your project, you're on your way to being a systems designer.

Systems of control: hierarchies and decentralization

Let's start with an obvious one: the problem of centralized vs distributed control structures. If I ask you what's a better org structure: a command-and-control hierarchy or a flat organization, most people have been indoctrinated to say the latter. Similarly if I ask whether you should have an old crusty centralized database or a fancy distributed database, everyone wants to build the latter. If you're an SRE and we start talking about pets and cattle, you always vote for cattle. You'd laugh at me if I suggested using anything but a distributed software version control system (ie. git). The future of money, I've heard, is distributed decentralized cryptocurrency. If you want to defeat censorship, you need a distributed social network. The trend is clear. What's to debate?

Well, real structures are more complicated than that. The best introductory article I know on this topic is Jo Freeman's The Tyranny of Structurelessness, which includes the famous quote: "This apparent lack of structure too often disguised an informal, unacknowledged and unaccountable leadership that was all the more pernicious because its very existence was denied."

"Informal, unacknowledged, and unaccountable" control is just as common in distributed computing systems as it is in human social systems.

The truth is, nearly every attempt to design a hierarchy-free, "flat" control system just moves the central control around until you can't see it anymore. Human structures all have leaders, whether implicit or explicit, and the explicit ones tend to be more diverse.

The web depends on centrally controlled DNS and centrally approved TLS certificate issuers; the global Internet depends on a small cabal who sorts out routing problems. Every blockchain depends on whoever decides if your preferred chain will fork this week, and whoever runs the popular exchanges, and whoever decides whether to arrest those people. Distributed radio networks depend on centralized government spectrum licenses. Democracy depends on someone enforcing your right to vote. Capitalism depends on someone enforcing the rules of a "free" marketplace.

At my first startup, we tried to run the development team as a flat organization, where everyone's opinions were listened to and everyone could debate the best way to do something. The overall consensus was that we mostly succeeded. But I was shocked when one of my co-workers said to me afterward: "Our team felt flat and egalitarian. But you can't ever forget that it was only that way because you forced it to be that way."

Truly distributed systems do exist. Earth's ecosystem is perhaps one (although it's becoming increasingly fragile and dependent on humans not to break it). Truly distributed databases using Raft consensus or similar algorithms certainly exist and work. Distributed version control (like git) really is distributed, although we ironically end up re-centralizing our usage of it through something like Github.

CAP theorem is perhaps the best-known statement of the tradeoffs in distributed systems, between consistency, availability, and "partition tolerance." Normally we think of the CAP theorem as applying to databases, but it applies to all distributed systems. Centralized databases do well at consistency and availability, but suck at partition tolerance; so do authoritarian government structures.

In systems design, there is rarely a single right answer that applies everywhere. But with centralized vs distributed systems, my rule of thumb is to do exactly what Jo Freeman suggested: at least make sure the control structure is explicit. When it's explicit, you can debug it.

Chicken-egg problems

Another archetypal systems design question is the "chicken-egg problem," which is short for: which came first, the chicken or the egg?

In case that's not a common question where you come from, the idea is eggs produce chickens, and chickens produce eggs. That's all fine once it's going, but what happened, back in ancient history? Was the very first step in the first iteration an egg, or a chicken?

The question sounds silly and faux-philosophical at first, but there's a real answer and that answer applies to real problems in the business world.

The answer to the riddle is "neither"; unless you're a Bible literalist, you can't trace back to the Original Chicken that laid the Original Egg. Instead there was probably a chicken-like bird that laid a mostly egg-ish egg, and before that, there were millions of years of evolution, going all the way back to single-celled organisms and whatever phenomenon first spawned those. What came "first"? All that other stuff.

Chicken-egg problems appear all the time when building software or launching products. Which came first, HTML5 web browsers or HTML5 web content? Neither, of course. They evolved in loose synchronization, tracing back to the first HTML experiments and way before HTML itself, growing slowly and then quickly in popularity along the way.

I refer to chicken-egg problems a lot because designers are oblivious to them a lot. Here are some famous chicken-egg problems:

- Electrical distribution networks
- Phone and fax technologies
- The Internet
- IPv6
- Every social network (who will use it if nobody is using it?)
- CDs, DVDs, and Blu-Ray vs HD DVD
- HDTV (1080p etc), 4k TV, 8k TV, 3D TV
- Interstate highways
- Company towns (usually built around a single industry)
- Ivy league universities (could you start a new one?)
- Every new video game console
- Every desktop OS, phone OS, and app store

The defining characteristic of a chicken-egg technology or product is that it's not useful to you unless other people use it. Since adopting new technology isn't free (in dollars, or time, or both), people aren't likely to adopt it unless they can see some value, but until they do, the value isn't there, so they don't. A conundrum.

It's remarkable to me how many dreamers think they can simply outwait the problem ("it'll catch on eventually!") or outspend the problem ("my new mobile OS will be great, we'll just subsidize a few million phones"). And how many people think getting past a chicken-egg problem, or not, is just luck.

But no! Just like with real chickens and real eggs, there's a way to do it by bootstrapping from something smaller. The main techniques are to lower the cost of adoption, and to deliver more value even when there are fewer users.

Video game console makers (Nintendo, Sony, Microsoft) have become skilled at this; they're the only ones I know who do it on purpose every few years. Some tricks they use are:

- Subsidizing the cost of early console sales.
- Backward compatibility, so people who buy can use older games even before there's much native content.
- Games that are "mostly the same" but "look better" on the new console.
- Compatible gamepads between generations, so developers can port old games more easily.
- "Exclusive launch titles": co-marketing that ensures there's value up front for consumers (new games!) and for content producers (subsidies, free advertising, higher prices).

In contrast, the designs that baffle me the most are ones that absolutely ignore the chicken-egg problem. Firefox and Ubuntu phones, distributed open source social networks, alternative app stores, Linux on the desktop, Netflix competitors.

Followers of this diary have already seen me rant about IPv6: it provides nearly no value to anyone until it is 100% deployed (so we can finally shut down IPv4!), but costs immediately in added complexity and maintenance (building and running a whole parallel Internet). Could IPv6 have been rolled out faster, if the designers had prioritized unwinding the chicken-egg problem? Absolutely yes. But they didn't acknowledge it as the absolute core of their design problem, the way Android, Xbox, Blu-Ray, and Facebook did.

If your product or company has a chicken-egg problem, and you can't clearly spell out your concrete plan for solving it, then investors definitely should not invest in your company. Solving the chicken-egg problem should be the first thing on your list, not some afterthought.

By the way, while we're here, there are even more advanced versions of the chicken-egg problem. Facebook or faxes are the basic form: the more people who use Facebook or have a fax machine, the more value all those users get from each other.

The next level up is a two-sided market, such as Uber or Ebay. Nobody can get a ride from Uber unless there are drivers; but drivers don't want to work for Uber unless they can get work. Uber has to attract both kinds of users (and worse: in the same geographic region! at the same time of day!) before either kind gets anything from the deal. This is hard. They decided to spend their way to success, although even Uber was careful to do so only in a few markets at a time, especially at first.

The most difficult level I know is a three-sided market. For example, UberEats connects consumers, drivers, and restaurants. Getting a three-sided market rolling is insanely complicated, expensive, and failure-prone. I would never attempt it myself, so I'm impressed at the people who try. UberEats had a head start since Uber had consumers and drivers in their network already, and only needed to add "one more side" to their market. Most of their competitors had to attract all three sides just to start. Whoa.

If you're building a one-sided, two-sided, or three-sided market, you'd better understand systems design, chickens, and eggs.

Second-system effect

Taking a detour from business, let's move to an issue that engineers experience more directly: second-system effect, a term that comes from the excellent book, The Mythical Man-Month, by Fred Brooks.

Second system effect arises through the following steps:

- An initial product starts small and is built incrementally, starting with a low budget and a few users.
- Over time, the product gains popularity and becomes profitable.

- The system evolves, getting more and more hacks on top, and early design tradeoffs start to be a bottleneck.
- The engineers figure out a new design that would fix all the mistakes we know about, plus more! (And they're probably right.)
- Since the product is already popular, it's easy to justify spending the time to "do it right this time" and "build a strong platform for the next 10 years." So a project is launched to rewrite everything from scratch. It's expected to take several months, maybe a couple of years, and a big engineering team.

Sound familiar? People were trying this back in 1975 when the book was written, and they're still trying it now. It rarely goes well; even when it does work, it's incredibly painful.

25 years after the book, Joel Spolsky wrote Things you should never do, part 1 about the company-destroying effect of Netscape/Mozilla trying this. "They did it by making the single worst strategic mistake that any software company can make: they decided to rewrite the code from scratch."

[Update 2020-12-28: I mention Joel's now-20-year-old article not because Mozilla was such a landmark example, but because it's such a great article.]

Some other examples of second system effect are IPv6, Python 3, Perl 6, the Plan9 OS, and the United States system of government.

The results are remarkably consistent:

- The project takes longer than expected to reach feature parity.
- The new design often does solve the architectural problems in the original; however, it unexpectedly creates new architectural problems that weren't in the original.
- Development time is split (or different developers are assigned) between maintaining the old system and launching the new system.
- As the project gets increasingly overdue, project managers are increasingly likely to shut down the old system to force users to switch to the new one, even though users still prefer the old one.

Second systems can be merely expensive, or they can bankrupt your company, or destroy your user community. The attention to Perl 6 severely weakened the progress of perl; the work on Python 3 fractured the python community for more than a decade (and still does); IPv6 is obstinately still trying to deprecate IPv4, 25 years later, even though the problems it was created to solve are largely obsolete.

As for solutions, there isn't much to say about the second system effect except you should do your utmost to prevent it; it's entirely self-inflicted. Refactor your code instead. Even if it seems like incrementalism will be more work... it's worth it. Maintaining two systems in parallel is a lot more expensive than you think.

In his book, Fred Brooks called it the "second" system on purpose, because it was his opinion that after experiencing it once, any designer will build their third and later systems more incrementally so they never have to go through that again. If you're lucky enough to learn from historical wisdom, perhaps even your second system won't suffer from this strategic error.

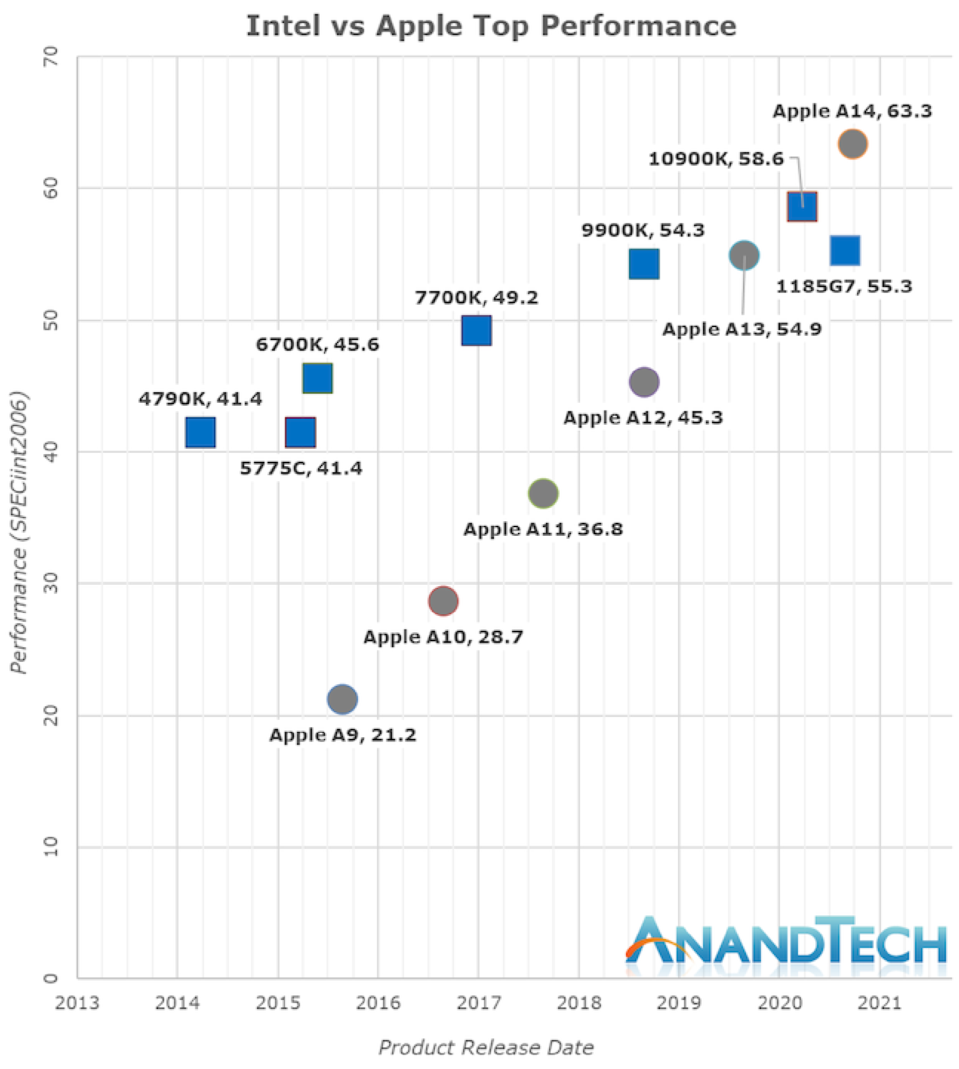
A more embarrassing related problem is when large companies try to build a replacement for their own first system, but the developers of the first system have left or have already learned their Second System Lesson and are not willing to play that game. Thus, a new team is assembled to build the replacement, without the experience of having built the first one, but with all the confidence of a group of users who are intimately experienced with its surface flaws. I don't even know what this phenomenon should be called; the vicarious second system effect? Anyway, my condolences if you find yourself building or using such a product. You can expect years of pain.

[Update 2020-12-28: someone reminded me that CADT ("cascade of attention-deficit teenagers") is probably related to this last phenomenon.]

Innovator's dilemmas

Let's finally talk about a systems design issue that's good news for your startup, albeit bad news for big companies. The Innovator's Dilemma is a great book by Clayton Christensen that discusses a fascinating phenomenon.

Innovator's dilemmas are so elegant and beautiful you can hardly believe they exist as such a repeatable abstraction. Here's the latest one I've heard about, via an Anandtech Article about Apple Silicon:



A summary of the Innovator's Dilemma is as follows:

- You (Intel in this case) make an awesome product in a highly profitable industry.
- Some crappy startup appears (ARM in this case) and makes a crappy competing product with crappy specs. The only thing they seem to have going for them is they can make some low-end garbage for cheap.
- As a big successful company, your whole business is optimized for improving profits and margins. Your hard-working employees realize that if they cede the ultra-low-end garbage portion of the market to this competitor, they'll have more time to spend on high-valued customers. As a bonus, your average margin goes *up!* Genius.
- The next year, your competitor's product gets just a little bit better, and you give up the new bottom of your market, and your margins and profits further improve. This cycle repeats, year after year. (We call this "retreating upmarket.")

- The crappy competitor has some kind of structural technical advantage that allows their performance (however you define performance; something relevant to your market) to improve, year over year, at a higher percentage rate than your product can. And/or their product can do something yours can't do at all (in ARM's case: power efficiency).
- Eventually, one year, the crappy competitor's product finally exceeds the performance metrics of your own product, and promptly blows your entire fucking company instantly to smithereens.

Hey now, we've started swearing, was that really called for? Yes, I think so. If I were an Intel executive looking at this chart and Apple's new laptops, I would be scared out of my mind right now. There is no more upmarket to retreat to. The competitor's product is better, and getting better faster than mine. The game is already over, and I didn't even realize I was playing.

What makes the Innovator's Dilemma so beautiful, from a systems design point of view, is the "dilemma" part. The dilemma comes from the fact that all large companies are heavily optimized to discard ideas that aren't as profitable as their existing core business. Any company that doesn't optimize like this fails; by definition their profitability would go down. So thousands of worker bees propose thousands of low-margin and high-margin projects, and the company discards the former and invests heavily in the latter (this is called "sustaining innovation" in the book), and they keep making more and more money, and all is well.

But this optimization creates a corporate political environment (aha, you see we're still talking about systems design?) where, for example, Intel could never create a product like ARM. A successful low-priced chip would take time, energy, and profitability away from the high-priced chips, and literally would have made Intel less successful for years of its history. Even once ARM appeared and their trendline of improvements was established, they still had lower margins, so competing with them would still cannibalize their own high-margin products, and worse, now ARM had a head start.

In case you're a big company reading this: the book has a few suggestions for what you can do to avoid this trap. But if you're Intel, you should have read the book a few years ago, not now.

Innovator's dilemma plots are the prettiest when discussing hardware and manufacturing, but the concept applies to software too, especially when software is held back by a hardware limitation. For example, distributed version control systems (where you download the entire repository history to every client) were amusing toys until suddenly disks were big enough and networks were fast enough, and then DVCSes wiped out everything else (except in projects with huge media files).

Fancy expensive databases were the only way to get high transaction throughput, until SSDs came along and made any dumb database fast enough for most jobs.

Complicated database indexes and schemas were great until AWS came along and let everyone just brute force mapreduce everything using short-term rental VMs.

JITs were mostly untenable until memory was so much slower than CPU that compiling was not the expensive part. Software-based network packet processing on a CPU was slower than custom silicon until generic CPUs got fast enough relative to RAM. And so on.

The Innovator's Dilemma is the book that first coined the term "disruptive innovation." Nowadays, startups talk about disrupting this and disrupting that. "Disruption" is an exciting word, everybody wants to do it! The word disruption has lost most of its meaning at this point; it's a joke as often as a serious claim.

But in the book, it had a meaning. There are two kinds of innovations: sustaining and disruptive. Sustaining is the kind that big companies are great at. If you want to make the fastest x86 processor, nobody does it better than Intel (with AMD occasionally nipping at their heels). Intel has every incentive to keep making their x86 processors better. They also charge the highest margins, which means the greatest profits, which means the most money available to pour into more sustaining innovation. There is no dilemma; they dump money and engineers and time into that, and they mostly deliver, and it pays off.

A "disruptive" innovation was meant to refer to specifically the kind you see in that plot up above: the kind where an entirely new thing sucks for a very long time, and then suddenly and instantly blows you away. This is the kind that creates the dilemma.

If you're a startup and you think you have a truly disruptive innovation, then that's great news for you. It's a perfect answer to that awkward investor question, "What if [big company] decides to do this too?" because the honest truth is "their own politics will tear that initiative apart from the inside."

The trick is to determine whether you *actually* have one of these exact "disruption" things. They're rare. And as an early startup, you don't yet have a historical plot like the one above that makes it clear; you have to convince yourself that you'll realistically be able to improve your thing faster than the incumbent can improve theirs, over a long period of time.

Or, if your innovation only depends on an existing trend - like in the software-based packet processing example above - then you can try to time it so that your software product is ready to mature at the same time as the hardware trend crosses over.

In conclusion: watch out for systems design. It's the sort of thing that can make you massively succeed or completely fail, independent of how well you write code or run your company, and that's scary. Sometimes you need some boxes and arrows.

Related
Books that explain (parts of) how the world really works (2018)
Factors in authentication (2019)
Unrelated
SimSWE 4: Wants, needs, and chasm-crossing (2021)

← *November 2020* *October 2021* →