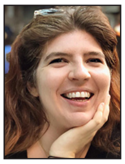
Revisiting Conway's Law

MARIANNE BELLOTTI



Marianne Bellotti has worked as a software engineer for over 15 years. She built data infrastructure for the United Nations to help humanitarian organizations share crisis data worldwide and spent three and a half years running incident response for the United States Digital Service. While in government she found success applying organizational change management techniques to the problem of modernizing legacy software systems. More recently, she was in charge of Platform Services at Auth0 and currently works as Principal Engineer for System Safety at Rebellion Defense. She has a book on running legacy modernization projects coming out this year from No Starch Press called *Kill It with Fire*. marianne.bellotti@gmail.com

After more than six years helping engineering organizations figure out how to modernize their technology, I've come to realize that Conway's Law is more about how organizational structure creates incentives than where boxes and lines are drawn on an org chart. Misaligned incentives for managers and individual contributors carve their impact into the system design, influencing tool selection and complicating future maintenance.

In 1968 Melvin Conway published a paper titled "How Do Committees Invent?" This paper, originally intended for *Harvard Business Review* but rejected for being too speculative, outlined the ways in which the structure and incentives of an organization influenced the software product it produced. It received little response but eventually made its way to the chair of the University of North Carolina at Chapel Hill's computer science department, Fred Brooks. At the time, Brooks had been pondering a question from his exit interview at IBM: why is it so much harder to manage software projects than hardware projects? Conway's insight linking the structure of software to the structure of the committees that invented them seemed significant enough for Brooks to repackage the thesis as "Conway's Law" when he published his guide on effectively managing software teams—*The Mythical Man-Month*—in 1975.

Yet this was not the only useful observation in Conway's paper. As it has subsequently been referenced by hundreds of computer science texts since Brooks's adoption of it as a universal truth, the more nuanced observations that supported Conway's argument have largely been omitted from the conversation. Conway's Law has become a voodoo curse, something that people believe only in retrospect. Few engineers attribute their architectural success to the structure of their organization, but when a product is malformed the explanation of Conway's Law is easily accepted.

Conway's original paper outlined not just how organizational structure influenced technology but also how human factors contributed to its evolution. Conway felt organizational structure influenced architecture because organizational structure influenced incentives. How individual contributors get ahead in a particular organization determined which technical choices were appealing to them.

Conway's observations are more important in maintaining existing systems than they are in building new systems. Organizations and products both change, but they do not always change at the same pace. Figuring out whether to change the organization or change the design of the technology is just another scaling challenge.

Individual Incentives

How do software engineers get ahead? What does an engineer on one level need to accomplish for the organization in order to be promoted to another level? Such questions are usually delegated to the world of engineering managers and not incorporated in technical decisions. And yet the answers absolutely have technical impacts.

Most of us have encountered this in the wild: a service, a library, or a piece of a system that is inexplicably different from the rest of the applications it connects to. Sometimes this is an older component of the system reimplemented using a different set of tools. Sometimes this is a new feature. It's always technology that was trendy at the time the code was introduced.

When the organization has no clear career pathway for engineers, software engineers grow their careers by building their reputation externally. This means getting drawn into the race of being one of the first to prove the production scale benefits of a new paradigm, language, or technical product. While it's good for the engineering team to experiment with different approaches as they iterate, introducing new tools and databases, and supporting new languages and infrastructures, increases the complexity of maintaining the system over time.

One organization I worked for had an entire stable of custom-built solutions for things like caching, routing, and message handling. Senior management hated this but felt their complaints—even their instructions that it stop—did little to course correct. Culturally, the engineering organization was flat, with teams formed on an ad hoc basis. Opportunities to work on interesting technical challenges were awarded based on personal relationships, so the organization's regular hack days became critical networking events. Engineering wanted to build difficult and complex solutions in order to advertise their skills to the lead engineers who were assembling teams.

Stern lectures about the importance of choosing the right technology for the job did not stop this behavior. It stopped when the organization hired engineering managers who developed a career ladder. By defining what the expectations were for every experience level of engineering and by hiring managers who would coach and advocate for their engineers, engineers could earn promotions and opportunities without the need to show off.

Organizations end up with patchwork solutions because the tech community rewards explorers. Being among the first with tales of documenting, experimenting with, or destroying a piece of technology builds an individual's prestige. Pushing the boundaries of performance by adopting something new and innovative contributes even more so to one's reputation.

Software engineers are incentivized to forego tried-and-true approaches in favor of new frontiers. Left to their own devices, software engineers will proliferate tools, ignoring feature overlaps for the sake of that one thing tool X does better than tool Y that is only relevant in that specific situation.

Well-integrated, high-functioning software that is easy to understand usually blends in. Simple solutions do not do much to enhance personal brand. They are rarely worth talking about. Therefore, when an organization provides no pathway to promotion for software engineers, the engineers are incentivized to make technical decisions that emphasize their individual contribution over smoothly integrating into an existing system.

Typically this manifests itself in one of three different patterns:

1. Creating frameworks, tooling, and other abstraction layers in order to make code that is unlikely to have more than one use case theoretically "reusable."
2. Breaking off functions into new services, particularly middleware.
3. Introducing new languages or tools in order to optimize performance for the sake of optimizing performance (in other words, without any need to improve an SLO or existing benchmark).

Essentially, engineers are motivated to create named things. If something can be named it can have a creator. If the named thing turns out to be popular, then the engineer's prestige is increased and her career will advance.

This is not to say that good software engineers should never create a new service, or introduce a new tool, or try out a new language on a production system. There just needs to be a compelling reason why these actions benefit the system versus benefit the prospects of the individual engineer.

Most of the systems I work on rescuing are not badly built. They are badly maintained. Technical decisions that highlight individuals' unique contributions are not always comprehensible to the rest of the team. For example, switching from language X to language Z may in fact boost memory performance significantly, but if no one else on the team understands the new language well enough to continue developing the code, those gains will be whittled away over time by technical debt that no one knows how to fix.

The folly of engineering culture is that we are often ashamed of signing our organization up for a future rewrite by picking the right architecture for right now, but we have no misgivings about producing systems that are difficult for others to understand and therefore impossible to maintain. This was a constant problem for software engineers answering the call to public service from organizations like United States Digital Service and 18F. When modernizing a critical government system, when should the team build it using common private sector tools and train the government owners on said tools, and when should the solution be built with the tools the government worker already knows? Wasn't the newest, greatest web application stack always the best option? Conway argued against aspiring for a universally correct architecture. He wrote in 1968, "It is an article of faith among experienced system designers that given any system design, someone someday will find a better one to do the same job. In other words, it is misleading and incorrect to speak of *the* design for a specific job, unless this is understood in the context of space, time, knowledge, and technology."

Manager Incentives

An engineering manager is a strange creature in the technical organization. How should we judge a good one from a bad one? Unfortunately, far too often managers advance in their careers by managing more people. And if the organization isn't properly controlling for that, then system design will be overcomplicated by the need to broadcast importance. Or as Conway put it: "The greatest single common factor behind many poorly designed systems now in existence has been the availability of a design organization in need of work."

Opportunities to go from an engineering manager and senior engineering manager come up from time to time as the organization grows and changes. It's the difference between handling one team and handling many. Managers leave, new teams form, existing teams grow past their ideal size. A good manager could easily earn those opportunities in the normal course of business. Going from senior manager to director, though, is more difficult. Going from director to vice president or higher even more so. It takes a long time for an organization to reach that level of growth organically.

Organizations that are unprepared to grow talent end up with managers who are incentivized to subdivide their teams into more specialized units before there is either enough people or enough work to maintain such a unit. The manager gets to check off career-building experiences of running multiple teams, hiring more engineers, and taking on more ambitious projects while the needs of the overall architecture are ignored.

Scaling an organization before it needs to be scaled has very similar consequences to scaling technology too early. It restricts your future technical choices. Deciding to skip the monolith phase of development and "build it right the first time" with microservices means the organization must successfully anticipate a number of future requirements and determine how code should be best abstracted to create shared services based on those predictions. Rarely if ever are all of those predictions right, but once a shared service is deployed, changing it is often difficult.

In the same way, a manager who subdivides a team before there is need to do so is making a prediction about future needs that may or may not come true. In my last role, our director of engineering decided the new platform we were building needed a dedicated team to manage data storage. Predictions about future scaling challenges supported her conclusions, but in order to get the head count for this new team, she had to cut it from teams that were working on the organization's existing scaling challenges. Suddenly, new abstractions around data storage that we didn't need yet were being developed while systems that affected our SLAs had maintenance and updates deferred.

Carrying existing initiatives to completion was not as attractive an accomplishment as breaking new ground. But the problem with designing team structure around the desired future state of the technology is that if it doesn't come true the team is thrown into the chaos of a reorganization. Aversions to reorganizations alone often incentivize people to build to their organizational structure.

Conclusion

Both individual contributors and managers make decisions with their future careers in mind. Those decisions create constraints on possible design choices that drive the organization to design systems that reflect the structure of the organization itself. Those wishing to benefit from the forces of Conway's Law would do well to consider how people within the engineering organization are incentivized before asking them to design a system.