Complex
The Most Overloaded Word in Technology

LAURA NOLAN

Laura Nolan's background is in site reliability engineering, software engineering, distributed systems, and computer science. She wrote the "Managing Critical State" chapter in the O'Reilly *Site Reliability Engineering* book and was co-chair of SREcon18 Europe/Middle East/Africa. Laura Nolan is a production engineer at Slack. laura.nolan@gmail.com

"Site reliability engineers shall predict the behavior of complex systems."

This sentence, which I recently came across in a job description, is fascinating because it is one that, depending on your background, could seem either reasonable enough or an utter glaring contradiction in terms.

Most people use the word *complex* as a synonym of *complicated* or *intricate*—something with a lot of parts that's hard to fully grasp. Understanding something complicated may be hard, but make the effort and you can, at least potentially, do it.

However, both software engineers and systems engineers use the word *complex* as a specific term of art. Software engineers in fact use it in several different ways, distinct from the systems meaning. Software engineers and systems engineers (please read that term throughout this article to mean SREs, production engineers, systems administrators, DevOps practitioners, etc.) are overlapping groups of people who work together. We all need to understand which meaning is in use at any given time so we can communicate clearly.

First, software engineers talk about time and space complexity: in other words, Big-O. In this context, complexity refers to how the time or space requirements to execute an algorithm scale with the properties of the input. There are also code complexity metrics like McCabe's Cyclomatic Complexity—that metric counts the number of independent code paths in a piece of software. But neither of these are what most of us mean when we discuss complexity or its inverse, simplicity.

Software Complexity

Complexity has been the enemy of the software engineer for decades now. Fred Brooks' classic essay "No Silver Bullet" [1] divided software's complexity into two parts: essential complexity and accidental complexity. Essential complexity is that related solely to specifying the problem and how it should be solved. Accidental complexity is related to the details of implementation. Writing your business logic and unit testing it is (hopefully) mostly essential complexity, but HTTP and managing concurrency and garbage collection and deployment to production are largely accidental complexity. The overwhelming majority of the work of technology operation is about accidental complexity.

But this doesn't tell us what software engineers mean by complexity. Fundamentally, complexity is that which makes software difficult to fully understand and to correctly reason about. Moseley and Marks' paper "Out of the Tarpit" [2] discusses several sources of complexity. The biggest, and hardest to deal with, is state—state influences the flow of control of a program, but the number of potential states a piece of software can be in increases exponentially with the number of variables. Dealing with this is such a difficult problem that we basically handwave past it: we normally run all tests on modules in known states, and we routinely restart misbehaving programs in order to restore them to a known good internal state.

Other major sources of complexity are sheer code volume and the fact that programs, unlike complex physical structures, cannot be visually inspected. Mental models of the program must be constructed from the source code. This can of course be easier or harder depending on how the code is structured. John Osterhout's book *A Philosophy of Software Design* [3] is all about making the design of software systems less complex, and he advocates very strongly for relatively few *deep* modules, each of which implements powerful functionality behind a simple interface. This is much like the UNIX philosophy—write small programs that do one thing well and can be used together.

Systems Complexity

Systems engineers tend to have a completely different idea of complexity, stemming from systems theory. Systems theory is a distinct area of research, spanning all kinds of manmade or natural systems—everything from an anthill to a nuclear power plant—and complex systems theory is a subset of it. Complex systems have particular characteristics: multiple interacting parts, system state (i.e., a memory of some kind), and feedback loops. They display emergent phenomena, have nonlinear relationships (small changes in one part can lead to large deviations in overall system behavior) and tend to be prone to cascading failures or "vicious cycles." Complex system behavior cannot be predicted reliably.

An amusing example of a complex systems failure is the incident that led to two interacting book pricing bots driving the price of a book on the genetics of flies to over 23 million dollars [4]. One bot was designed to set its price to undercut its competition by 2%, and another bot was coded to price books it didn't have in stock at 27% above the price it found in the market (in order to make a profit reselling them). In the case of one rare book, each bot set its price based on the other bot's price on a daily basis, leading to a vicious cycle of compounding prices. This system has multiple interacting parts, state and feedback loops—it is a complex system, albeit a trivial example of one.

All computing systems are complex systems. Even if a system is running on a single physical machine you are still dealing with the interactions of multiple pieces of software, all of which are likely complex systems in their own right, running on complex hardware. Each running program may have multiple threads of control, state, interactions with the operating system and other programs—even if not explicitly then via shared resources.

The "Stella Report" [5] describes several real-world examples of the kinds of deviations and failures that are commonly experienced in complex computing systems. In one example from the report, the combination of centralized logging with the ELK stack plus installation of a keylogger for audit purposes resulted in system failure when the remote Logstash program experienced intermittent failure. The issue was compounded by the terminal becoming unresponsive (waiting for the logging system), hindering debugging. That outcome is hard to predict ahead of time by reasoning about system behavior. This is why chaos testing has become popular. It's easier, and far more reliable, to add latency to a component in a controlled fashion and see what is affected than to attempt to model all the possible interactions between system components.

This systems theory definition of complexity is the one often used by systems administrators, SREs, and DevOps practitioners—this is in no small part due to the impact of Richard Cook's paper "How Complex Systems Fail" [6] on the industry some years ago. Software engineers, on the other hand, mainly think in terms of code structure, interactions between modules, and interdependencies in their code bases. Software engineers' primary concern is the difficulty of making correct changes without introducing errors. Systems engineers' primary concern is stability of the deployed software in production.

This is why, when you ask a software engineer to promote simplicity as part of their job description, they look for opportunities to separate concerns and reduce coupling in their code base to refactor to well-known design patterns, create better-defined interactions between modules, and remove unused code.

When you ask systems engineers to do the same thing, they often look for ways to control extremes of the system's behavior (using load shedding and circuit breakers, for instance), or to make elements of the system more uniform. Dave Mangot's recent *;login:* article "Achieving Reliability with Boring Technology" [7] discusses the use of infrastructure-as-code techniques to make sure your production environments are standard and well-understood. That's a very good example of the kinds of ways that systems engineers can reduce complexity.

The two kinds of complexity that we discuss here are quite different, but they do also have one major thing in common: both software complexity and systems complexity make the task of understanding and predicting behavior impossible.

All of us—software engineers, systems administrators, site reliability engineers, production engineers, DevOps practitioners—we are all fighting the same two-faced demon named complexity. In both software and operations, complexity arises from state, from the sheer number of components or modules, from the number of interactions (both intended and unintended), as well as from the impossibility of direct inspection of the systems we work on.

Code and the running production system are two aspects of the same thing, and it's very unlikely we can run a stable, reliable, performant, maintainable system if either variety of complexity (code or systems) is not continually managed. Let's understand each other's language, and let's always have empathy for the challenges that our colleagues face.

References

[1] F. Brooks, "No Silver Bullet—Essence and Accident in Software Engineering," *Proceedings of the IFIP 10th World Computing Conference*, 1986.

[2] B. Moseley, P. Marks, "Out of the Tar Pit," BCS Software Practice Advancement (SPA 2006).

[3] J. Osterhout, *A Philosophy of Software Design* (Yaknyam Press, 2018).

[4] M. Masnick, "The Infinite Loop of Algorithmic Pricing on Amazon...Or How a Book on Flies Cost $23,698,655.93," Techdirt: http://bit.ly/2FagxMz (accessed March 18, 2019).

[5] D. Woods, "STELLA Report," SNAFUcatchers Workshop on Coping with Complexity, 2017.

[6] R. I. Cook, MD, "How Complex Systems Fail," Cognitive Technologies Lab, University of Chicago, 2002.

[7] D. Mangot, "Achieving Reliability with Boring Technology," *;login:*, vol. 44, no. 1 (Spring 2019): https://www.usenix.org/publications/login/spring2019/mangot.