RC25024 (W1007-072) July 21, 2010
Computer Science

IBM Research Report

On Architecture: Systems Architecture

Grady Booch
IBM Research Division
Boulder, CO
gbooch@us.ibm.com

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Systems Architecture

Grady Booch

The International Council on Systems Engineering (INCOSE) defines a system as “a construct or collection of different elements that together produce results not obtainable by the elements alone. The elements, or parts, can include people, hardware, software, facilities, policies, and documents, that is, all things required to produce system-level results. The value added by the system, as a whole, beyond that contributed independently by the parts, is primarily created by the relationship among the parts, that is, how they are interconnected.” Continuing, INCOSE notes “systems engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem.”

Building quality software is hard; building quality software-intensive systems is wickedly hard.

In his delightful and witty book *Systemantics*, John Gall makes a number of pithy observations. Everything is a system. Everything is part of a larger system. Systems display antics; the total behavior of large systems cannot be predicted. In complex systems, malfunction or even total non-function may not be detectable for long periods, if ever. Colossal systems foster colossal errors.

Now, I largely care about the creation, development, deployment, evolution, operation, and support of software-intensive systems, and by that I mean systems in the INCOSE sense but especially those systems for which software is the dominant and necessary element. In such systems, the most important artifact is the raw, running, naked code that executes on hardware and interacts with humans and the real world. All other artifacts are secondary, but nonetheless they are still critical, for they help the enterprise deliver the right system at the right time to the right stakeholders with the right balance of cost and value.

Software-intensive systems bring their own wickedness to the world because, as Fred Brooks notes, they have an essential complexity. Furthermore, there are fundamental challenges to discrete systems, since they exhibit non-continuous behavior, often embody a combinatorial explosion of state space, and may suffer corruption from unexpected external events. Furthermore, as a discipline we lack the mathematical tools and intellectual capacity to model the behavior of ultra large discrete systems. Indeed, as Flood notes in his book *Dealing With Complexity*, there are a number of triggers of complexity in such systems, due to the huge number of interactions, high numbers of parts, and degrees of freedom. Furthermore, discrete software-intensive systems often exhibit nonlinearity and

suffer from broken symmetry and, due to nonholonomic constraints, often exhibiting what Flood calls localized transient anarchy.

Koichi Tanaka, who won the Nobel Prize in Chemistry in 2002, has observed "most of the work performed by a development engineer results in failure." I might state it even more dismally: all complex systems fail. Some such failures are entirely invisible, some are benign, and some are just plain annoying, but other modes of failure may be positively painful if not fully fatal.

All systems will eventually fail when operated outside their design envelop: a 747 was not design for aerobatics; Twitter cannot accommodate every human on earth tweeting at the same moment; the New York Stock exchange can process only so many transactions per second. Some system failures flow from a direct chain of events triggered by the failure of a single component: the failure of the O rings that led to the Challenger disaster; a tree branch falling on a power line that contributed to the 2003 blackout in northeastern United States. Other failures follow from the unexpected interaction of subsystems that are otherwise functioning properly: the failure of the Mohave Generating Station in Nevada in 1971 comes to mind, where there were problems of electrical and mechanical resonance. Finally, some systems fail due to the unexpected consequences of its operation: introducing cane toads to attack pests in Australian cane fields led to the cane toads becoming their own sort of pest.

As Petroski states in *To Engineer is Human*, we advance by improving on those things that work and fixing those things that break. As such, there's much we can learn from how complex software-intensive systems fail and thus can constructively consider how systems succeed.

As Dave Parnas notes, "as a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications. John Gall, also in *Systemantics,* observes that "a complex system that works is invariably found to have evolved from a simple system that works" and as Herbert Simon states in *The Sciences of the Artificial*, it is best to have a continuous series of intermediate stable forms from the evolution of the simple to the complex. Simon goes on to say that "both vertically and horizontally, the most resilient systems tend to exhibit loose coupling and tight cohesion among components" or as Bosworth says even more bluntly that "software which is flexible, simple, sloppy, tolerant, and altogether forgiving turns out to be most resilient." Dick Gabriel says it even more tersely: worse is better. In his essay by the same name, Gabriel suggests that "simplicity is the most important consideration in a design; both implementation and interface must be simple, though it is more important for the implementation to be simple."

Eberhardt Rechtin, in *The Art of Systems Architecting*, offers a number of practical observations on delivering complex software-intensive systems. Do the hard part first. If it works, then it is useful. All design domains contain an

irreducible kernel of problems that are best addressed through creative and heuristic approaches that combine art and science. No complex system can be optimized to all parties concerned, nor all functions optimized. To state it another way, a system is shaped by a myriad of design decisions by different stakeholders that work to balance the forces swirling around the system.

In my earlier column on “Software Abundance in the Face of Economic Scarcity” I talked about attending to the pain one feels in a project by triage deciding if you need a tourniquet, an aspirin, or a vitamin. To elaborate, consider what one must do when a system fails or is on the verge of failure: first, do no harm; don’t panic; stop the bleeding. regain control.

That’s essentially the process of systems engineering, in the small as well as in the large. Fix what’s most broken, then the next thing, then the thing after that, and along the way reach for the next milestone, and the one after that, and the one after that. In so doing, you will stabilize your system and regain control, by growing the system’s architecture through the incremental and iterative release of testable executables.

Once you’ve done that, and your system is no longer lying on the floor bleeding out, then you can begin to make things better. Take stock of the project’s vitals: what significant decisions have been made; what are their consequences; what are their risks; what are the known unknowns. Then, attack those risks with a fierce and ruthless focus. Cast away what is meaningless. Refocus on the important. Intentionally manage the urgent. As the effect of this aspirin takes hold, then you can think about making your system even healthier with a regimen of daily hygiene and vitamins. Learn who you are most important stakeholders are. Learn where the skeletons are, who knows what, and when they knew it. Calibrate your team. Get control of your significant design decisions, and question their assumptions.

This, by the way, is where modeling comes in, as I discussed in “Architecture as a Shared Hallucination.” We model to abstract, to reason about, to document, to transform. Only once we have mastered these things can we meaningfully proceed to supercharging our system, by continuous systemic refactoring coupled with the intentional personal development of the team.

In the presence of essential complexity, establishing simplicity in one part of the system requires trading off complexity in another. This is why, as Philippe Kruchten has often said, “the life of a successful software architect is an ongoing series of suboptimal decisions often made in the dark and under pressure.” Systems architecting is not for the meek. As Douglas Adams suggests, “don’t panic” but then also delegate, and give yourself the space to recharge, and the space to innovate.

The life of the systems engineer is akin to the life of Hercules, who carried out his

twelve labors though his raw strength, power, and leadership. At times, a systems engineer may feel a bit like Sisyphus, forever pushing a boulder up a hill, only to have it roll back down (but remember that according to the legend, Sisyphus was given that punishment because of his clever and crafty ways). Finally, the life of a systems engineer is a bit like Job: patient and long-suffering.

Building quality software-intensive systems is wickedly hard, but in the end, it is essentially only a problem of engineering, albeit one that requires all our human and technical skill to carry out with grace and grit.