Chapter 1. Origins of Software Architecture

We are most of us governed by epistemologies that we know to be wrong.

Gregory Bateson

The purpose of this book is to help you design systems well and to help you realize your designs in practice. This book is quite practical and intended to help you do your work better. We must begin theoretically and historically. This chapter is meant to introduce you to a new way of thinking about your role as a software architect that will inform both the rest of this text and the way in which you approach your projects moving forward.

Software's Conceptual Origins

We shape our buildings, and thereafter they shape us.

Winston Churchill

FADE IN:

INT. A CONFERENCE HALL IN GARMISCH GERMANY, OCTOBER 1968 — DAY

The scene: The NATO Software Engineering Conference.

Fifty international computer professors and craftspeople assembled to determine the state of the industry in software. The use of the phrase software *engineering* in the conference name was deliberately chosen to be “provocative” because at the time the makers of software were considered so far from performing a scientific effort that calling themselves “engineers” would be bound to upset the established apple cart.

MCILROY

We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters.
(pause)
The creation of software is backwards as an industry.

KOLENCE

Agreed. Programming management will continue to deserve its current poor reputation for cost and schedule effectiveness until such time as a more complete understanding of the program design process is achieved.

Though these words were spoken, and recorded in the conference minutes in 1968, they would scarce be thought out of place if stated today.

At this conference, the idea took hold was that we must make software in an *industrial* process.

That seemed natural enough, because one of their chief concerns was that software was having trouble defining itself as a field as it pulled away from hardware. At the time, the most *incendiary*, most *scary* topic at the conference was “the highly controversial question of whether software should be priced separately from hardware.” This topic comprised a full day of the four-day conference.

This is a way of saying that software didn’t even know it existed as its own field, separate from hardware, a mere 50 years ago. Very smart, accomplished professionals in the field were not sure whether software was even a "thing," something that had any independent value. Let that sink in for a moment.

Software was born from the mother of hardware. For decades, the two were (literally) fused together and could hardly be conceived of as separate matters. One reason is that software at the time was "treated as though it were of no financial value" because it was merely a necessity for the hardware, the true object of desire.

Yet today you can buy a desktop computer for $100 that's more powerful than any computer in the world was in 1968. (At the time of the NATO Conference, a 16-bit computer—that's two bytes—would cost you around $60,000 in today's dollars.)

And hardware is produced on a *factory line*, in a clear, repeatable process, determined to make dozens, thousands, millions of the same physical object.

Hardware is a commodity.

A commodity is something that is interchangeable with something of the same type. You can type a business email or make a word-processing document just as well on a laptop from any of 50 manufacturers.

And the business people want to form everything around the efficiencies of a commodity except one thing: their "secret sauce." Coca-Cola has nearly 1,000 plants around the world performing repeated manufacturing, putting Coke into bottles and cans and bags to be loaded and shipped, thousands of times each day, every day, in the same way. It’s a heavily scrutinized, sharply measured business: an internal commodity. Coke is bottled in factories in identical bottles in identical ways, millions of times every day. Yet only a handful of people know the secret *formula* for making the drink itself. Coke is copied millions of times a day, every day, and bottled in an identical process. But making the recipe a commodity would put Coke out of business.

In our infancy, we in software have failed to recognize the distinction between the commodities representing repeated, manufacturing-style processes, and the more mysterious, innovative, *one-time work* of making the recipe.

Coke is the recipe. Its production line is the factory. Software is the recipe. Its production line happens at runtime in browsers, not in the cubicles of your programmers.

Our conceptual origins are in hardware and factory lines, and borrowed from building architecture. These conceptual origins have confused us and dominated and circumscribed our thinking in ways that are not optimal, and not necessary. And this is a chief contributor to why our project track record is so dismal.

The term “architect” as used in software was not popularized until the early 1990s. Perhaps the first suggestion that there would be anything for software practitioners to learn from architects came in that NATO

Software Engineering conference in Germany in 1968, from Peter Naur:

Software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem. As one single example of such a source of ideas, I would like to mention: Christopher Alexander: Notes on the Synthesis of Form (Harvard Univ. Press, 1964) (emphasis mine).

This, and other statements from the elder statesmen of our field at this conference in 1968, are the progenitors of how we thought we should think about software design. The problem with Naur's statement is obvious: it's simply false. It's also unsupported. To state that we're in a "similar position to architects" has no more bearing logically, or truthfully, to stating that we're in a similar position to, say, philosophy professors, or writers, or aviators, or bureaucrats, or rugby players, or bunnies, or ponies. An argument by analogy is always false. Here, no argument is even given. Yet here this idea took hold, the participants returning to their native lands around the world, writing and teaching and mentoring for decades, shaping our entire field. This now haunts and silently shapes—perhaps even circumscribes and mentally constrains, however artificially—how we conduct our work, how we think about it, what we "know" we do.

ORIGINS

To be clear, the participants at the NATO conference in 1968 were very smart, accomplished people, searching for a way to talk about a field that barely yet existed and was in the process of forming and announcing itself. This is a monumental task. I hold them in the highest esteem. They created programming languages such as ALGOL60, won Turing Awards, and created notations. They made our future possible, and for this I am grateful, and in awe. The work here is only to understand our origins, in hopes of improving our future. We are all standing on the shoulders of giants.

Some years later, in 1994, the Gang of Four created their *Design Patterns* book. They explicitly cite as inspiration the work of Christopher Alexander, a professor of architecture at University of California at Berkeley and author of *A Pattern Language*, which is concerned with proven aspects of architecting towns, public spaces, buildings, and homes. The *Design Patterns* book was pivotal work, one which advanced the area of software design and bolstered support for the nascent idea that *software designers are architects*, or are “like” them, and that we should draw our own concerns and methods and ideas from that prior field.

This same NATO conference was attended by now-famous Dutch systems scientist Edsger Dijkstra, one of the foremost thinkers in modern computing technology. Dijkstra participated in these conversations, and then some years later, during his chairmanship at the

Department of Computer Science at the University of Texas, Austin, he voiced his vehement opposition to the mechanization of software, refuting the use of the term “software engineering,” likening the term “computer science” to calling surgery “knife science.” He concluded, rather, that “the core challenge for computing science is hence a conceptual one; namely, *what (abstract) mechanisms we can conceive* without getting lost in the complexities of our own making” (emphasis mine).

This same conference saw the first suggestion that software needed a “computer engineer,” though this was an embarrassing notion to many involved, given that engineers did “real” work, had a discipline and known function, and software practitioners were by comparison ragtag. “Software belongs to the world of ideas, like music and mathematics, and should be treated accordingly.” Interesting. Let’s hang on to that for a moment.

* * *

Cut to:

INT. PRESIDENT’S OFFICE, WARSAW, POLAND — DAY

The scene: The president of the Republic of Poland updates the tax laws.

In Poland, software developers are classified as creative artists, and as such receive a government tax break of up to 50% of their expenses (see Deloitte report). These are the professions categorized as creative artists in Poland:

- Architectural design of buildings
- Interior and landscape
- Urban planning
- Computer software
- Fiction and poetry
- Painting and sculpture
- Music, conducting, singing, playing musical instruments, and choreography
- Violin making
- Folk art and journalism
- Acting, directing, costume design, stage design
- Dancing and circus acrobatics

Each of these are explicitly listed in the written law. In the eyes of the Polish government, software development is in the same professional category as poetry, conducting, choreography, and folk art.

And Poland is one of the leading producers of software in the world.

Cut to: HERE—PRESENT DAY.

Perhaps something has occurred in the history of the concept of structure that could be called an event, a rupture that precipitates ruptures.

This rupture would not have been represented in a single explosive moment, a comfortingly locatable and suitably dramatic moment. It would have emerged among the ocean tides of thought and expression, across universes, ebbing and flowing, with fury and with lazy ease, over time, until the slow trickling of traces and cross-pollination reveal, only later, something had transformed. Eventually, these traces harden into trenches, fixing thought, and thereby fixing expression and realization.

What this categorization illuminates is the tide of language, the patois of a practice that shapes our ideas, conversation, understanding, methods, means, ethics, patterns, and designs. We name things, and thereafter, they shape us. They circumscribe our thought patterns, and that shapes our work.

The concept of structure within a field, such as we might call "architecture" within the field of technology, is thereby first an object of language.

Our language is constituted of an interplay of signs and of metaphors. A metaphor is a poetic device whereby we call something something that it isn't in order to reveal a deeper or hidden truth about that object by underscoring or highlighting or offsetting certain attributes. "All the world's a stage, and all the men and women merely players" is a well-known line from Shakespeare's *As You Like It*.

We use metaphors so freely and frequently that sometimes we even forget they are metaphors. When that happens, the metaphor “dies” (a metaphor itself!) and *becomes* the name itself, drained of its original juxtaposition that gave the phrase depth of meaning. We call these “dead metaphors.” Common dead metaphors include the “leg” of a chair, or when we “fall” in love, or when we say time is “running out,” as would sand from an hourglass. When we say these things in daily conversation, we do not fancy ourselves poets making metaphors. We don’t see the metaphor, or intend one. It’s now just The Thing.

In technology, “architecture” is a nonnecessary metaphor. That word, and all it’s encumbered by, directs our attention to certain facets of our work.

Architecture is a dead metaphor: we mistake the metaphor for The Case, the fact.

There has been considerable hot debate, for decades, over the use of the term architect as applied to the field of technology. There are hardware architectures, application architectures, information architectures, and so forth. So can we claim that architecture is a dead metaphor if we don’t quite understand what it is we’re even referring to? We use the term without quite understanding what we mean by it, what the architect’s process is, and what documents they produce toward what value. “Architect” means, from its trace in Greek language, “master builder.”

What difference does it make?

Copies and Creativity

No person who is not a great sculptor or painter can be an architect. If he is not a sculptor or painter, he can only be a builder.

John Ruskin, “True and Beautiful”

Dividing roles into distinct responsibilities within a process is one useful and very popular way to approach production in business. Such division makes the value of each moment in the process, each contribution to the whole, more direct and clear. This fashioning of the work, the “division of labor,” has the additional value of making each step observable and measurable.

This, in turn, affords us opportunities to state these in terms of SMART goals, and thereby reward and punish and promote and fire those who cannot meet the objective measurements. Credit here goes at least in some part to Henry Ford, who designed his car manufacturing facilities more than 100 years ago. His specific aim was to make his production of cars cheap enough that he could sell them to his own poorly compensated workers who made them, ensuring that what he could not keep in pure profit after the consumption of raw materials—his paid labor force—would return to him in the form of revenue.

This way of approaching production, however, is most (or only) useful when what is being produced is well defined and you will make many (dozens, thousands, or millions) of copies of identical items.

In *Lean Six Sigma*, processes are refined until the rate of failure is reduced to six standard deviations from the mean, such that your production process allows 3.4 quality failures per million opportunities. We seek to define our field, to find the proper names, in order to codify, and make repeatable processes, and improve our happiness as workers (the coveted “role clarity”), and improve the quality of our products.

But one must ask, how are our names serving us?

Processes exist to *create copies*. Do we ever create copies of the software itself? Of course, we create copies of software for *distribution* purposes: we used to burn copies of web browsers onto compact discs and send them in the mail, and today we distribute copies of software over the internet. That is a process facilitating distribution, however, and has little relation to the act of creating that single software application in the first place. In fact, *we never do that*.

Processes exist, too, in order to repeat the act of doing the same *kind* of thing, if not making the same exact thing. A software development methodology catalogs the work to be done, and software development departments have divisions and (typically vague) notions of the processes we undergo in the act of creating any software product or system. So, to produce software of some kind, we define roles that participate in some aspect of the process, which might or might not be formally represented, communicated, and executed accordingly.

This problem of determining our proper process, our best approach to our work, within the context of large organizations that expect measurable results according to a quarterly schedule, is exacerbated because competition and *innovation* are foregrounded in our field of technology. We must innovate, make something new and compelling, in order to compete and win in the market. As such, we squarely and specifically aim *not* to produce something again that has already been produced before. Yet our embedded language urges us toward processes and attendant roles that might not be optimally serving us.