Editor: **George Fairbanks**
gf@georgefairbanks.com

Software Architecture is a Set of Abstractions

George Fairbanks

TWENTY YEARS AGO, in this magazine, Martin Fowler published the influential essay "Who Needs an Architect?"[1] Today, when I ask developers where they learned about software architecture, they point me to that essay, directly or indirectly. The essay provides three definitions of software architecture, quoting Ralph Johnson for the definitions and commentary. After weighing the options, Johnson is critical of all three and metaphorically throws up his hands:

> So, this makes it hard to tell people how to describe their architecture. "Tell us what is important." Architecture is about the important stuff. Whatever that is.

Twenty years is a long time in computer science. When those words were written, some of the engineers I work with today were in diapers. With the benefit of hindsight, I will make a case for the third definition, that software architecture is a set of abstractions.

Johnson rejects the definition that architecture is "the highest level concept of a system in its environment," retorting that "[t]here is no highest level concept of a system" because each stakeholder sees the system differently, and developers are just one stakeholder. I agree. Describing architecture as "high level" is a convenient crutch when introducing the idea, but it does not stand up to careful scrutiny.

I prefer a variant of this definition that is in the same vein but avoids the "high level" trap: "The set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both."[2] It focuses on reasoning, not levels. Architecture is what you need to reason about a system: software elements, relations, and properties. Those are abstractions that let you reason about software, both before and after you build it.

Missing Abstractions

But surely, we already have plenty of abstractions! Computer science is swimming in abstractions. So many, in fact, that it's easy to overlook what's missing. Let's review how we grew to understand architecture. Way back in 1975, Frank DeRemer and Hans Kron observed that developers have abstractions for writing data structures, methods, and modules, but not for assembling modules into systems.[3]

> [S]tructuring a large collection of modules to form a "system" is an essentially distinct and different intellectual activity from that of constructing the individual modules. That is, we distinguish programming-in-the-large from programming-in-the-small.

A few decades later, in 1993, David Garlan and Mary Shaw strengthened and generalized the argument, sketching out what we now call software architecture.[4]

> As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives. This is the software architecture level of design.

Digital Object Identifier 10.1109/MS.2023.3269675
Date of current version: 14 July 2023

Pause to consider their point. Could you reason about a system using only algorithms and data structures? Or do you find yourself going beyond these abstractions when you talk to other developers or think through the design of your system?

By 2010, after two more decades of innovation, the set of architecture abstractions had settled down. A group of authors collected the abstractions into a book titled "The Secret Abstractions of Software Architecture, Finally Revealed!" No, I'm pulling your leg. They actually titled it "Documenting Software Architectures" and released it when companies were abandoning heavyweight processes in favor of agile ones that discouraged documentation.[2] I think that's why it's not more famous.

A few years ago, when I read a book that told a story of innovations in mathematics, it stitched together ideas that had been independent islands in my mind. So, instead of reciting an inventory of the architecture abstractions, what I'll do here is sketch a story of innovation. As summarized in Figure 1, the story has several related plotlines: specifications, structure, views, and patterns. I'm surveying decades of work, so this is an overview, not a complete inventory.

Specification Abstractions

A specification is a precise description or clear identification of something. In the earliest days of computing, there were no specifications of software because the concepts of hardware and software were intertwined. People set out to build useful machines such as Babbage's analytical engine that, in the mid-1800s, was designed to compute tables of numbers. Ada Lovelace not only wrote the first algorithm for it, but she also recognized that with suitable instructions the hardware could operate on any kind of symbol—that software could be distinct from hardware.

By the late 1940s, software was in fact distinct from hardware. We had multiple algorithms for the same function, differing in their use of storage space or run time. People talked about speed–space tradeoffs. This is the germ of a critical idea in software architecture: specifying features versus qualities. Qualities, or more fully, quality attributes, go beyond just speed and space; they include latency, usability, modifiability, portability, and many more "-ities." As you reason about software, you want to know not only about its features (what it computes) but its qualities (how it computes them). You may have heard the term "non-functional requirements," but it carries baggage: "non-functional" means "broken" and "requirements" can imply a waterfall process. It is convenient to discuss qualities of designs—say that one has better latency—without stating a requirement or suggesting a development process.

By the 1960's, it was commonplace to compile and link programs in separate steps. That required a new abstraction: specifying a subroutine interface separately from its implementation. A program could depend on an interface—say, sorting—but wait until linking to choose a fast or space-efficient implementation. This abstraction helps you reason about a program from its interfaces, letting your mind skip past the implementations. As with all abstractions, you lose detail that way, but in return you gain the ability to reason about larger programs.

Structure Abstractions

Software architecture depends on a second group of abstractions, ones related to the structure of the program. In the early 1950s, David Wheeler identified the need for reusable chunks of code, introducing subroutines and libraries. Subroutines were grouped into modules, and modules, like subroutines, were split between interface and implementation. By the 1970s, David Parnas saw that some ways of modularizing are better than others. Sometimes a change to a module required changes to its neighbors, other times not. Module interfaces could be designed to hide the details that might change, called *information hiding*. A program designed with information hiding would work the same (have the same features) but be easier to modify (have different quality attributes).

By the mid-1970s, programs were large enough that programmers complained that while they could understand any given piece, they had difficulty understanding the whole program. "[C]urrent languages discourage

Specification Abstractions	Structure Abstractions
• 1850s: Software and Hardware	• 1950s: Modules
• 1940s: Features and Quality Attributes	• 1970s: Information Hiding
• 1960s: Interfaces and Implementations	• 1990s: Components and Connectors
View Abstractions	**Pattern Abstractions**
• 1960s: Compile-Time and Run-Time Views	• 1990s: Named Architecture Patterns
• 1990s: Multiple Views	• 1990s: Styles Promote Qualities
• 2000s: Viewtypes	• 2010s: Patterns in Every Viewtype

FIGURE 1. Some abstractions in software architecture, shoehorned into a few categories.

the accurate recording of the overall solution structure; they force us to write programs in which we are so preoccupied with the trees that we lose sight of the forest, as do the readers of our programs!".[3] This is where the terms programming-in-the-small and programming-in-the-large originated.

In object-oriented programming, there is a clear distinction between a class and an object: A class representing a person can have several instances, one each for Ann, Bob, and Carl. A similar type-versus-instance distinction for modules was not made clear until the early 1990s. At that point, the terms module and component were no longer used interchangeably: Modules exist at compile-time and components at run-time.

With the advent of computer networking in the late 1960s, it was necessary to describe the interactions as protocols. It was not until the early 1990s, however, that interactions between components had a first-class abstraction: connectors. Connectors express protocols and much more. Examples of connectors include call–return, publish–subscribe, and pipes. The implementation of a connector often requires a lot of code, organized into many modules. A procedure call is a simple connector that developers use to implement more complex connectors, for example, remote procedure calls or event-based connectors.

View Abstractions

When designing physical structures like houses or bridges, it's common to create diagrams showing the structure from different perspectives, or views. Views play a critical role in software architecture. In the late 1960s, Edsger Dijkstra observed that mentally animating code is difficult and error-prone, so programs should be written in a structured way, so that it's easier to look at the code and envision how it behaves. Said another way, developers stare at one view (the code), imagine another view (its runtime behavior), and reason about how changes to one affect the other.

In the mid-1990s, Philippe Kruchten identified views as a useful abstraction for software architecture. Not just compile-time and run-time views, but also concurrency and deployment to hardware. Each view enables different kinds of reasoning, perhaps needed by different people on the team.

The halting problem says that we cannot always look at code and say if it will run forever. It's the extreme case of Dijkstra's point: It's hard to use one view (say, the source code) to reason about another (say, it's runtime behavior). It's similarly hard to look at code and answer: Is this code running in production, and if so, where? Yet developers confronted by a bug report must reason not about the code in their repository, but the version of the code that users are interacting with. This leads to a final view abstraction: viewtypes. Viewtypes are a grouping of views, the most common of which are compile-time, run-time, and deployment, and they cannot be easily reconciled with each other.

Pattern Abstractions

All kinds of engineers give names to recurring patterns, like truss bridges or hybrid cars, and software engineers have done the same. In the early 1990s, Mary Shaw created a catalog of architectural patterns that had been in use for decades, such as client-server, pipe-and-filter, and batch-sequential. As she and David Garlan formalized these patterns (also called styles), two ideas emerged.

First, architectural patterns are inherently linked to quality attribute tradeoffs. For example, if the system needs to be low latency, then client-server is more suitable than map-reduce, and if your system needs high throughput, then the choice is reversed. The linkage of architecture patterns to promoted/inhibited qualities lifts a fog covering architectural design. Qualities like latency or availability emerge from the design choices in a system. It's a relief to be able to influence them directly instead of just "rolling up your sleeves" and hoping that daily vigilance pays off.

Typical systems have an inconsistent mish-mash of patterns. Consider a house that's been adapted over the years using the patterns and materials of the day. You'd like to reason about the house, for example "it has insulation, so I'll be warm in it." But can you? Perhaps one room has insulation, but another room does not. When patterns are applied inconsistently, you don't get much reasoning power.

This leads to the second idea. Consistency leads to stronger reasoning power: The more consistently a system applies a pattern, the easier it is to reason about. People tend to use the term *architectural style* when a pattern is applied consistently as opposed to piecemeal. When a system conforms to an architectural style, it plays by the style's rules. Consider the Portable Operating System Interface (POSIX) standard in which programs communicate with signals such as *SIGTERM* and *SIGKILL*. A well-behaved program listens and acts appropriately; a poorly behaved program learns that `kill -9` is the boss. Returning to the house analogy, if you consistently follow style rules about insulating a house, then you can reason about its warmth.

In 2010, a final architecture abstraction took me by surprise. I knew about the three main viewtypes (compile-time, run-time, and deployment), and I knew about architectural patterns. However, all of the patterns I knew about were run-time patterns, so I was shocked when I read about patterns for how source code is arranged, and patterns about how components are deployed to datacenters.[2] An example of a pattern in the deployment viewtype is redundant deployment to datacenters: by deploying the same components to many datacenters, the system can keep running if a datacenter goes offline. I'd like to say this was an easy extension of what I already knew but in reality, I had to struggle before I internalized it.

Chain of Intentionality

A few years ago, my software development team was at an offsite team building event where we learned how to cook. Because I like to cook, I already had some of the skills being taught, and I ended up coordinating several of the dishes that our team was preparing. We had some vegetarians, so we cooked two pans of Brussels sprouts, one with and one without bacon. Just before serving, however, someone picked up the pans and combined them. Luckily, there were other vegetarian dishes that evening.

The well-meaning person saw there were two pans, of course, but incorrectly inferred that the intent was to feed more people, not to accommodate different diets. This mistake happened because design intent was lost, and I'm to blame for that. (It's also an example of why it's dangerous to hoard the architectural knowledge of a system). There was a design to this meal, so to speak, and satisfying our audience led to arranging the cooking in a certain way.

Using two pans is a small detail, but it had a big impact. Was it architectural? As I said earlier, Ralph Johnson was right, "[t]here is no highest level concept of a system" because each stakeholder sees the system differently. Architecture isn't just the "high level" anything, even if that phrasing is a convenient shorthand. To our vegetarian stakeholders, however, the two-pan design detail was indeed architectural.

Does that mean all details are architectural? No, our systems are big and complex, so we must aggressively simplify if we hope to reason about them. Software architecture is a set of abstractions that lets you reason about your system, especially about quality attributes. Our field has had small abstractions like subroutines, algorithms, and data structures for a long time now. It has taken decades to accumulate larger abstractions like information hiding, components and connectors, multiple views, and architectural styles. When we design systems, we weave these abstractions together, preserving a chain of intentionality, so that the systems we design do what we want.[5]

Twenty years ago, Martin Fowler asked, "Who needs an architect?" and, with help from Ralph Johnson, helped shape the way a generation of software developers thought about architecture. It's not just twenty years later; our systems are twenty years bigger. Today, what's more important than asking about job roles is: Can you reason about the software you plan to build, or have already built? It's time for developers to take another look at software architecture and see it as a set of abstractions that helps them reason about software.

ABOUT THE AUTHOR

GEORGE FAIRBANKS is a software engineer at Google, New York, NY 10019 USA. Contact him at gf@georgefairbanks.com.

References

1. M. Fowler, "Design - Who needs an architect?" *IEEE Softw.*, vol. 20, no. 5, pp. 11–13, Sep./Oct. 2003, doi: 10.1109/MS.2003.1231144.
2. P. Clements et al., *Documenting Software Architectures*. Upper Saddle River, NJ, USA: Addison-Wesley, 2010.
3. F. DeRemer and H. Kron, "Programming-in-the large versus programming-in-the-small," in *Proc. Int. Conf. Reliable Softw.*, Apr. 1975, pp. 114–121, doi: 10.1145/800027.808431.
4. D. Garlan and M. Shaw, "An introduction to software architecture," in *Proc. Adv. Softw. Eng. Knowl. Eng., Ser. Softw. Eng. Knowl. Eng.*, V. Ambriola and G. Tortora, Eds. Singapore: World Scientific, 1993, vol. 2, pp. 1–39.
5. G. Fairbanks, *Just Enough Software Architecture*. Boulder, CO, USA: Marshall & Brainerd, 2010.