GREAT BUILDING ARCHITECTURE

often works without putting itself at center stage. It provides form to building structures and makes them functional, and it can inspire awe or bestow tranquility discreetly and unobtrusively. Software architectures also work like that. We value software systems mainly for what they offer us, and rarely pause to admire the architectural principles and practices that allowed such systems to be constructed without collapsing like a house of cards. In the words of R. Buckminster Fuller, “Ninety-nine percent of all that is going to affect our tomorrows is being developed . . . in ranges of reality that are nonhumanly sensible.”1

However, the increasing scale and complexity of the software we build puts ever more strain on its underlying architecture. A single architect can no longer comprehend and control systems comprising tens of millions of LOC evolving over decades. Consequently, large modern systems typically are built as federations of diverse components glued together through middleware and APIs.

This approach has worked admirably well, letting us develop huge systems that serve billions of users reliably and effectively. The price we pay when building such systems comes from the cacophonous mix of diverse architectural styles and the rigidity imposed by each element’s architecture. Microservices let us abstract system components and derive value from their distinct architectural styles. Still, in large systems, responses to radically changed requirements are difficult or ineffective, and evolution typically accumulates technical debt. The size of modern large systems can drown even the most accomplished software architect.

In computing, we have repeatedly solved problems associated with size and complexity by employing theory, systems, and automation. For example, writing low-level code is no longer the problem it used to be, because advances in programming-language theory, compiler design, and sophisticated register-allocation algorithms let us program effectively in high-level languages. Similarly, nowadays OSs abstract complex tasks such as process scheduling, memory allocation, and device virtualization, while relational database management systems organize our data and optimize the intricate queries we run on them. Our vision for software architectures can follow similarly ambitious lines.

Looking to Nature

Our aspirations on where self-evolving software architectures should lead us can come from nature. For example, consider the human body’s architecture. Over billions of years it has evolved, though not engineered, as a tremendously complex and highly effective system of systems.

At the lowest level, it comprises trillions of reusable, adaptable, and multipurpose elements—the cells. Each one of these has a complex structure delineated by its membrane, kept in shape by the cytoskeleton, and containing numerous specialized organelles, such as the nucleus, endoplasmic reticulum, and Golgi apparatus, as well as mitochondria, lysosomes, peroxisomes,
WELCOME NEW ADVISORY BOARD MEMBERS

Hans Aerts is the vice president of software engineering in TomTom’s automotive business unit, responsible for managing the engineering resources and internal development processes. His areas of expertise include project management, software measurement, software process improvement, and outsourcing. He previously worked at Philips Electronics and NXP. Hans has helped develop embedded software for telecommunications systems, test and measurement equipment, car navigation systems, and consumer electronics products. He helped start software-process-improvement activities throughout Philips Electronics and introduced CMMI and software process assessments. He was closely involved in setting up the Philips Software Center in Bangalore. Aerts received an MSc in mathematics and computer science from the Eindhoven University of Technology. Contact him at hans.aerts@tomtom.com.

Paulo Caroli is a principal consultant at ThoughtWorks Brazil and a cofounder of AgileBrazil. At ThoughtWorks, he has been an agile coach, a trainer, and a project manager. He has more than 20 years’ experience in software development in corporations in Brazil, India, the US, and Latin America. Ever since he met Extreme Programming, he has focused on agile and lean processes and practices. Caroli received a bachelor of computer science and an MS in software engineering, both from the Pontifical Catholic University of Rio de Janeiro. He’s the author of Lean Inception: How to Align People and Build the Right Product and coauthor (with Tainá Caetano) of Fun Retrospectives: Activities and Ideas for Making Agile Retrospectives More Engaging. Contact him at pcaroli@thoughtworks.com.

Filippo Lanubile is a professor in the University of Bari’s Computer Science Department, where he leads the collaborative-development research group. His research interests lie at the intersection of software engineering and social computing, and aim at advancing research and practice in collaborative software development. He has received Microsoft Research’s Software Engineering Innovation Foundation Award, IBM’s Faculty Award, the Eclipse Innovation Award, and the NASA Group Achievement Award. He’s on the editorial board of Empirical Software Engineering. Contact him at filippo.lanubile@uniba.it.

Andy Ruth is the managing director of Sustainable Evolution and focuses on enterprise transformation. He has worked for both large corporations and successful (and failed) startups, operating in most industry verticals. In the late 1990s, he shifted from a primary focus as a technical expert to an author and a public speaker. Since early 2000, he has gained experience creating training and certification programs, as well as modeling, creating, and managing IT roles, compensation plans, and career ladders. He has created or maintained Microsoft certification and training, created and maintained IASA training and certification programs, and worked with CompTIA on certifications. Recently, Ruth has been creating and updating IT frameworks to work with cloud environments in agile and DevOps environments, creating apprenticeship programs to onboard new IT talent, and updating how we bring new entrants into the workforce and educate them. Contact him at ar@sustainableevolution.com.
vacuoles, and ribosomes. At a much higher level, the human architecture features tens of highly specialized, extremely effective, long-living organs, such as the brain, eyes, liver, heart, and lungs. And at an even higher level, we have about a dozen organ systems, such as the circulatory, digestive, endocrine, immune, lymphatic, musculoskeletal, nervous, reproductive, respiratory, and urinary systems. The human body also features the means to organize and separate organs and systems, such as the body cavities and blood–brain barrier.

In the field of software architecture, our vision should be of processes and systems that allow code, the building block of software, to organize itself into architectures that evolve together with the software’s changing features and requirements. For example, when a system’s performance can’t satisfy increasing demand, we would like the architecture to automatically adjust the code so as to offer concurrency, speculation, and caching. Or, when code is exposed to potentially malicious agents, we would expect its architecture to grow protective barriers, including authentication, authorization, access control, and intrusion detection, as well as mechanisms for labeling data according to its origin.

Furthermore, as features accumulate, large modules should automatically split into smaller, more manageable ones, while modules that often work together should be grouped into larger agglomerations and develop appropriately effective APIs. The self-evolving architecture should recognize and exploit opportunities for reuse and gracefully deprecate and put to rest unused elements. Guidance on the architecture’s form can come from the properties Christopher Alexander identified on structures that we perceive to have life: levels of scale, strong centers, boundaries, local symmetries, deep interlock, graded variation, and others.²

The vision I’m proposing might appear outlandish, but some parts of it are already here. Refactoring and automated refactoring tools allow code to adjust according to evolving designs. Microservices provide a way to break unmanageable monoliths into more malleable units. Design patterns offer reusable blueprints for organizing code elements. A key challenge is upgrading these abstractions and practices into a higher-order form that’s amenable to machine-driven automation.

References

1. R.B. Fuller, R. Buckminster Fuller on Education, Univ. of Massachusetts Press, 1979, p. 130.
got flaws?

Find out more and get involved:
cybersecurity.ieee.org