Foreword

Since the late 1980’s, architecture research has emerged as the principled study of the large-scale structures of systems. From its roots in qualitative descriptions of empirically observed useful system organizations, architecture has matured to encompass broad explorations of notations, tools, and analysis techniques. Whereas initially the research area interpreted practice, it now offers concrete guidance for complex design and development. It has made the transition from basic research to an essential element of design and construction.

So began a retrospective paper that Mary Shaw and I wrote for IEEE Software entitled “The Golden Age of Software Architecture” (Shaw & Clements, 2006). I took the liberty of removing six instances of the word “software” and one “system,” and I think the result makes a suitable beginning for this book.

There are countless examples like this. Notations, tools, analysis, methods, languages, papers, case studies, and more, each created within one architectural “genre” (software, system, enterprise) could be easily generalized to embrace the other two.

Are all of these kinds of architecture the same? That’s the wrong question to ask. The right question is “Does it pay to consider the similarities among these kinds of architectures, or does it pay to concentrate on their differences?” I think the answer is easy: “Examine the similarities!”

I have been thinking and writing about software architecture for, on some days, longer than I like to remember. It has been fun watching the field grow up. In the early days, we spent most of our time teaching and writing about what was meant by this new (for us) term “architecture,” and why anybody would want to devote time and resources in creating one, and what you could do with it after you had.

In those days, so much energy was spent on definitions, and on advocating my definition over your definition. This is symptomatic of a field in its early days. In the 1980s you could bring any object-oriented meeting to its knees by raising your hand, putting on your best innocent face, and asking, “What’s an object?” The resulting debate was guaranteed to de-rail the meeting’s agenda.

Architecture, like the O-O community, got past that. These days, neither they nor we have a single crisp universal definition of our central term, and yet we are not paralyzed because of that. We make progress. We communicate effectively. We build better systems. It’s as though all the competing definitions form a scatterplot in a concept space, and what we mean lives in the neighborhood of the centroid, and that’s good enough.

In fact, this “useful ambiguity” may help us as we try to merge the different kinds of architectures. We are already used to making allowances for the fact that what you mean by architecture is not precisely what I mean by architecture, but it is close enough that we can communicate and use the concept to our
mutual engineering advantage as we build systems (of whatever type). So it’s a small marginal effort, then, to admit that your kind of architecture is not the same as my kind, but we can still find common ground.

Maturing fields stop looking inward at some point and start looking outward, to see how they relate to the world they live in. I think a sad consequence of this early inward perspective for software is that software architecture still is not able to express its constructs in ways that seamlessly and unambiguously translate to the ways that implementers express theirs. This more than anything else leads to architectures that by dictate or by benign neglect become irrelevant after the first code release. But maybe we’ll learn from others. Do enterprise architects know how to bridge that gap? Do system architects?

And so we’ve started looking around, and this book is a wonderful and unmistakable sign of that.

The enterprise and system architects I talk to are looking around, too, and to my surprise and delight, many of them see the work we’ve done in software architecture as something they can put to good use. We have done a good job methodizing ways to evaluate our architectures, and to express the goals that our architectures need to satisfy. We have bodies of guidance (over and above notations) for capturing our architectures in way that stakeholders can see their needs met, and (oh, yes) an excellent grasp of who our stakeholders are and how to capture what they need. We have a rich body of knowledge comprising architectural solutions for a staggering collection of hard problems, and have thought a lot about effective ways to capture, catalog, and disseminate that knowledge.

The title of this book speaks of aligning these three kinds of architectures, but what I think it means is “aligning those aspects of these three kinds of architectures that aren’t already aligned, because a lot of them are.” I grant you that’s a lousy book title. But the reason that it’s an interesting topic is because there’s so much already shared in common, and the goal is to push that commonality harder, to nurture it and make it grow.

I played a small role in creating the Architectural Tradeoff Analysis Method (ATAM), for methodically analyzing and evaluating a software architecture for its suitability vis-à-vis a set of functional and quality attribute goals for it. The ATAM was created for use squarely in the genre of software, and yet we soon began to realize that its fundamental principles and many of its specific techniques are “genre-agnostic.” If you want to evaluate an architecture – any architecture – for suitability, it doesn’t take a genius to realize that you need to compare it against its stated goals. That means you have to know how to state them. For software architecture, we express those goals as quality attribute scenarios (very short structured stories of stakeholders interactions that depict how well – how fast, how securely, how reliably, etc. – the system reacts). Scenarios are very general, and work just fine for system and enterprise concerns as well (although the quality attributes of interest may differ). If you want to state the goals, you have to collect them, which means you have to know who owns them or can express them. That leads you to stakeholders. In the ATAM our results comprise risks and non-risks that we discover by running scenarios (prioritized by their importance to the salient stakeholders) against the architecture. There is nothing software-specific about that.

The kinds of architecture – what I call genres – don’t bring different governance principles to the table, just different concerns. One aspect of aligning the genres, then, is to try to discover as many of these common principles as we can.

In 2008, the U.S. Army – a very large consumer of all of these kinds of architectures – engaged the Software Engineering Institute to arrange and hold a workshop on exactly this topic (Bergey, 2008). In addition to the three genres of architecture treated by this book, the Army workshop added system-of-systems (DUSD, 2008) architectures as well, for good measure. The workshop’s goals included clarifying the relationships among these different genres, and exploring and identifying areas of commonality...
and difference. Some two dozen people from the Army, their contractors, and government R&D labs participated.

Here are some statements that emerged from the workshop. Each of these statements was made about the architecture of a particular genre, but as I did in the opening paragraph, I’ve “sanitized” them by removing the genre name or replacing it with “#”. No matter which genre you consider your “native” one, see if you disagree with any of them:

- # architecture is the process of translating business vision and strategy into effective # change by creating, communicating and improving the key requirements, principles, and models that describe the #’s future state and enable its evolution.
- The scope of the # architecture includes the people, processes, information and technology of the #, and their relationships to one another and to the external environment.
- Architects compose holistic solutions that address the business challenges of the # and support the governance needed to implement them (Lapkin, 2008).
- # architecture is an ongoing process.
- # architecture must provide holistic solutions.
- # architects work for the client and with the builder; thus, architects generate requirements as much as receive requirements.
- # architects develop information in all of the views needed to make the client’s decision.
- # architects write an architecture description as a consequence of the information developed to support the decision.
- # architecture is an early life-cycle artifact and perfectly poised to serve as an early life-cycle risk mitigation vehicle.
- # architecture is an abstraction of a #.
- # architecture defines the properties of elements.
- #’s can and do have many structures; these constitute its architecture.
- Every # has an architecture.
- One of # architecture’s most important roles is to be the primary carrier of quality attributes.
- # architecture is the bridge between business and mission goals and a #.
- Quality attribute requirements drive the # architecture design.
- # architecture drives # development throughout the life cycle.
- # architecture must be central to # development activities.

Can you find one you didn’t like or rings false for the genre you work in? I can’t. This suggests that these genres have a great deal in common, which probably matches our intuition all along. And it also suggests that if we continue to look, we’ll continue to find more commonality – another reason to be happy that this book has come along.

A workshop being a place where people work, the participants were asked to form groups (one per genre) to answer, among other things, “What are the major activities involved in each genre?” The working groups came up with dozens of activities. Speaking as a long-time proponent and teacher of the practice of software architecture, I can tell you that I certainly wish that I’d thought of several of the things that came out of the other (that is, the non-software) working groups, such as “crafting an incremental integration strategy.” Yes, we do get around to teaching that in software architecture classes, but I was delighted to see it achieve first-class status in one of other genres.
Every one of the captured activities fell neatly into one of the following four buckets:

1. Understanding goals, context, and requirements.
2. Creating, evaluating, and documenting architecture.
3. Managing the architecture post-creation.
4. Assisting in post-architecture activities.

This commonality suggests that the alignment of these genres is already well under way. All is not solved, of course, and there are many hard problems remaining, as the chapters you are about to read attest. My own list of critical research areas includes these:

- Finding a common architectural ontology (concept set) among the genres that could lead to a documentation or specification language that applies very well, even if not perfectly well, to architectures in each genre.
- Finding a common ontology for the “implementation” phase of each genre, and finding a mapping to the elements of the first ontology. This could give us a path towards bridging the language gap between architects and downstream engineers.
- Finding a common concept set for evaluating architectures in any genre. Using a method like the ATAM as a starting point, this set could be seeded with concepts such as “stakeholder,” “scenarios,” “prioritized scenario,” “quality attributes,” “architectural risk,” “architectural non-risk,” “architectural tradeoff,” and more. All of these seem genre-generic.

This work lets us all become intellectual citizens of a larger world, and not just our own parochial regions. When we work in our own genre, I hope that we all can pause to reflect how our work could (if we just thought about it for a bit) apply and thereby contribute to the other genres as well. With this work, all of us have gained whole new communities that we can now call colleagues. Let’s help each other.

Paul Clements
BigLever Software, Inc, USA

REFERENCES


Paul Clements is the Vice President of Customer Success at BigLever Software, Inc., where he works to spread the adoption of systems and software product line engineering. Prior to this, he was a senior member of the technical staff at Carnegie Mellon University’s Software Engineering Institute, where he worked for 17 years leading or co-leading projects in software product line engineering and software architecture documentation and analysis. Clements is the co-author of three practitioner-oriented books about software architecture: “Software Architecture in Practice” (1998, second edition 2003), “Evaluating Software Architectures: Methods and Case Studies” (2001), and “Documenting Software Architectures: View and Beyond” (2002, second edition 2010). He also co-wrote “Software Product Lines: Practices and Patterns” (2001), and was co-author and editor of “Constructing Superior Software” (1999). In addition, Clements has also authored dozens of papers in software engineering reflecting his long-standing interest in the design and specification of challenging software systems. In 2005 and 2006 he spent a year as a visiting faculty member at the Indian Institute of Technology in Mumbai. He was a founding member of the IFIP WG2.10 Working Group on Software Architecture. He received a B.S. in Mathematical Sciences in 1977, and a M.S. in Computer Science in 1980, both from the University of North Carolina at Chapel Hill. He received a Ph.D. in Computer Sciences from the University of Texas at Austin in 1994.