

“The Golden Age of Software Architecture” Revisited

Paul Clements, *Carnegie Mellon University Software Engineering Institute*

Mary Shaw, *Carnegie Mellon University Institute for Software Research*

“**T**he Golden Age of Software Architecture” was about the maturation of a field of research and practice that has spanned over a quarter century, with roots going back still farther.¹ The article first appeared in spring 2006, only about two-and-a-half years before *IEEE Software* invited us to write this follow-up.

Our message was optimistic: software architecture is a healthy engineering discipline, following a classic technology maturation trajectory. Such a trajectory begins (using the Redwine-Riddle model of technology maturation²) with basic research; passes through intermediate stages of concept formulation, exploration, and codification; and ends when nobody would consider building a software system without the technology. “It will be considered an unexceptional, essential part of software system building,” we wrote, “taken for granted, employed without fanfare, and assumed as a natural base for further progress.”

Our analogy was aviation, whose golden age is considered to have been the 1930s, when innovation and concept exploration were at their peak. All the aviation progress since then, as magnifi-

cent as it is, is basically just 70 years of incremental improvement in propulsion, materials, controls, and production, with no seismic upheavals in technology or principles. Today’s ultramodern Airbus A-340 is in many ways nearly identical to the Douglas DC-8-60 that rolled out almost 50 years ago.

Our article viewed software architecture as having enjoyed a golden age of innovation and concept formulation, and beginning to enter the more mature stage of quiet discipline and unremarkable utilization. For sure, mature technologies aren’t as exciting as technologies in the midst of growth spurts, but the next time you step aboard a commercial jet, ask yourself how exciting you hope the flight will be. And the next time you craft the software architecture for a system on whose success your company’s future might depend, ask yourself how much adrenalin you really want to flow.

Recent Progress

Even in the relatively short time since the publication of “Golden Age,” we see evidence to strengthen our belief that we’re in transition from the exploration of concepts to routine application of those concepts.

In January 2009, I asked for follow-up pieces from several sets of authors whose insightful and influential *Software* classics made the magazine’s 25th-anniversary top-picks list (Jan./Feb. 2009, pp. 9–11). Here, Paul Clements and Mary Shaw provide fresh perspectives on their winning article, addressing how their thinking has evolved over the years, what has changed, and what has remained constant.

—Hakan Erdogmus, *Editor in Chief*

For example, many organizational initiatives in architecture competence have recently sprung up.³ These organizations are asking how they can help their architects do their jobs more effectively, more routinely, more predictably, and more professionally. Approaches include standardized training, certification, architect-specific career tracks, creation of forums for architects to share ideas and solutions, repositories of architectural artifacts such as style definitions and documentation templates, and mentoring of junior architects.⁴ No organization would undertake such an effort unless it believed that the practice of architecture was essential to its success and that there were mature and proven practices to which it could appeal.

At an internal architecture conference held by a major Indian IT service company—one of those organizations with initiatives to improve its architecture competence—a senior architect confided that his job was essentially complete as soon as he decided whether SAP or Oracle would provide the platform for his application. He wasn't wrong. Such is the happily unremarkable state of affairs in some organizations and for some domains.

How did we arrive here? Software engineering can be seen as a continuous journey to make the primitives of software design more sophisticated and capable. The primitives used to be the subroutine, then the module, then the object, then the component, and then the service. Today's primitives are breathtaking in their sophistication and include "relational database," "shopping cart," "transaction manager," "rules engine," "online auction," "global-positioning navigation," "search engine," and "user interface." Along the way, software architecture has been the essential unifying concept to make the primitives (of whatever scope) work together successfully.

In fact, you could argue that software architecture has been the conceptual foundation that gave us the intellectual control to successfully create and then piece together larger and larger chunks of software, and do it unremarkably. This has gotten us to the point where, in some domains, the bulk of a multimillion-line system comes wrapped in cellophane, and its architect views his or her job as choosing which cellophane-wrapped package to buy.

Architecture Will Be Architecture

For other domains, we aren't there yet. The new generation of multicore computers might demand whole new architecture styles, as will the increasing assembly of third-party functionality over the Web (going much beyond the current view of

software-as-a-service). We also need a systematic understanding of the architectures—the conceptual architectures, not the port-level protocols—for cyberspace and the emerging Web. Systems-of-systems⁵ and ultra-large-scale systems⁶ are harbingers of a future in which we don't have crisply bounded systems so much as (possibly ad hoc) coalitions. These will require new models of governance and organizational interaction (or lack thereof).

But even in this brave new world, these new creations' architecture will still be architecture. Just as the idea of architecture helped take us from sub-routines to subsystems, it will provide the strong, stable technical footing to let us work on the social and organizational issues brought about by these new paradigms. Yes, we might need new styles or solution approaches as we enter new domains; 'twill always be so. But we won't need to change the fundamental principles of architecture.

For example, service-oriented architecture (SOA) created a stir in the IT community the likes of which were unseen since Y2K, but all the commotion was on the business and economics side. Architecturally, SOA is just a style like any other, with its own set of implications for quality attributes.⁷ The concepts and foundations of architecture let architects take SOA in stride—you could almost feel architects around the world shrugging their shoulders and getting on with it.

Continuing Opportunities

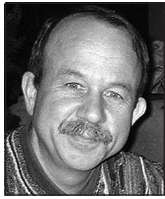
We closed "Golden Age" by listing a half dozen "significant opportunities ... for new contributions in software architecture." We close this follow-up by reiterating three that have strong potential to make real improvements.

Object-Oriented Programming vs. Architecture

Object-oriented (OO) programming is the leading software development paradigm of our time, and it's certainly an improvement over traditional procedural programming. The resulting focus on programming-level constructs and the avalanche of tools and frameworks to support them hamper architectural thinking, though. Whereas procedural programming is architecture-agnostic, the frameworks that support OO programming embed architectural decisions. Architects, eager to communicate with programmers and lacking a true lingua franca for architecture, must do so in the programmers' terms. Language constrains thought; here, the constraints undercut the designer's imperative to choose the best architecture for the job. Designers instead choose the (restricted

Software engineering can be seen as a continuous journey to make the primitives of software design more sophisticated and capable.

About the Authors



Paul Clements is a senior member of the technical staff at Carnegie Mellon University's Software Engineering Institute. His research interests involve software architectures and software product lines. Clements has a PhD in computer sciences from the University of Texas at Austin. Contact him at clements@sei.cmu.edu.

Mary Shaw is the Alan J. Perlis Professor of Computer Science at Carnegie Mellon University's School of Computer Science. Her research interests are value-based software engineering, everyday software, software engineering research paradigms, and software architecture. Shaw has a PhD in computer science from Carnegie Mellon University. She's a fellow of the ACM, IEEE, and American Association for the Advancement of Science. Contact her at mary.shaw@cs.cmu.edu.



and conceptually impoverished) architectures they can most easily represent and tool. It's symptomatic that the typical connector in these systems is still just the procedure call, although now often dynamically bound.⁸

Yes, designs of current OO frameworks are subtle and well reasoned and provide some of those breathtaking primitives we've mentioned. But in many cases, those breathtaking primitives are purchased at the cost of breathtaking complexity, much of which we suspect is, in Frederick Brooks' sense, accident rather than essence.⁹ Just as a crop circle is hard to see when you're standing in the middle of it, we believe that much of frameworks' accidental complexity comes from their bottom-up creation intended to give programmers, not architects, more powerful, expressive forms.

Given the trade between readily available tools with a misfit architecture and the right architecture with meager tools, which would you choose? We would choose the latter, but current standard practice tells us we're in the minority. Work remains to be done so that we no longer have to choose convenience over concept, and we look forward to the day when the architectures of OO systems transcend the workaday constructs of OO programming.


Design Decisions and Quality Attributes

The study of software architecture recognizes the tight coupling between an architecture and a software system's quality attributes such as performance, modifiability,

and security. Architectural patterns¹⁰ and tactics¹¹ are prepackaged design decisions created (and handily cataloged for use) to achieve quality-attribute goals and requirements. We're encouraged by recent signs that these concepts are taking root and being nurtured by practitioners whose primary purpose is to apply rather than create them.¹² Continuing to understand and tighten the link between quality-attribute requirements and architectural design decisions brings us closer to making those decisions more quickly and more assuredly.

Conformance Checking and Architecture Recovery

The best architecture is worthless if the code doesn't follow it. This is a risk during initial development; in many shops the risk becomes a near certainty in post-deployment maintenance. Tools to analyze code for architecture conformance are still woefully inadequate and rely on humans making suggestions (read: guesses) about architectural constructs that might be lurking in the code. The problem is hard. Many architectural patterns, fundamental to the system's design taken forward into code, are undetectable once programmed. Layers, for instance, usually compile right out of existence. To our knowledge, no one has even compiled a catalog detailing which often-used patterns and tactics can and can't be tracked down in code. For those that can't, it would help to find a way to tag the code with markers when they're used, to give code analyzer tools a fighting chance to report their existence.

The two-and-a-half years since "Golden Age" was published aren't long enough for the field to mature very much more, so we'll need more time to see whether our predictions hold true. Stay tuned. But there are strong indications that software architecture is enjoying a time of both external exploration and popularization. This tells us that it's on the cusp of passing from its golden age into its period of reliable use and value. It's on its way to becoming unremarkable. And that's wonderful. 

Acknowledgments

Mary Shaw's work was supported by the A.J. Perlis Chair of Computer Science at Carnegie Mellon University. We thank our colleagues at Carnegie Mellon for feedback on early drafts.

References

1. M. Shaw and P. Clements, "The Golden Age of Software Architecture," *IEEE Software*, vol. 23, no. 2, 2006, pp. 31–39.
2. S. Redwine and W. Riddle, "Software Technology Maturation," *Proc. 8th Int'l Conf. Software Eng.* (ICSE 85), IEEE CS Press, 1985, pp. 189–200.
3. L. Bass et al., "A Workshop on Architecture Competence," tech. note CMU/SEI-2008-TN-024, Software Eng. Inst., Carnegie Mellon Univ., Oct. 2008.
4. P. Clements et al., "The Duties, Skills, and Knowledge of Software Architects," *Proc. 6th Working IEEE/IFIP Conf. Software Architecture* (WICSA 07), IEEE CS Press, 2007, pp. 44–47.
5. M.W. Maier, "Architecting Principles for Systems-of-Systems," *Systems Eng.*, vol. 1, no. 4, 1998, pp. 267–284.
6. L. Northrop et al., *Ultra-Large-Scale Systems: The Software Challenge of the Future*, Software Eng. Inst., Carnegie Mellon Univ., 2006.
7. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
8. M. Shaw, "Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status," *Studies of Software Design*, LNCS 1078, Springer, 1996, pp. 17–32.
9. F.P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Apr. 1987, pp. 10–19.
10. F. Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.
11. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1993.
12. J.P. Scott, "Evaluating Distributed Systems Architectures for Fault-Tolerant Applications: Saturn 2008," Boeing, 2008; www.sei.cmu.edu/architecture/saturn/2008/presentations/SATURN08-jscott.pdf.