The Golden Age of Software Architecture

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

Paul Clements, *Software Engineering Institute, Carnegie Mellon University*

> In the near future, software architecture will attain the status of all truly successful technologies: It will be taken for granted.

Since the late 1980s, software architecture has emerged as the principled understanding of the large-scale structures of software systems. From its roots in qualitative descriptions of empirically observed useful system organizations, software architecture has matured to encompass a broad set of notations, tools, and analysis techniques. Whereas initially the research area interpreted software practice, it now offers concrete guidance for complex software design and development.

It has made the transition from basic research to an essential element of software system design and construction.

This retrospective examines software architecture's growth in the context of a technology maturation model, matching its significant accomplishments to the model's stages to gain perspective on where the field stands today. This trajectory has taken architecture to its golden age. In the near future, it will attain the status of all truly successful technologies: It will be considered an unexceptional, essential part of software system building—taken for granted, employed without fanfare, and assumed as a natural base for further progress.

How technologies mature

Samuel Redwine and William Riddle reviewed several software technologies to see how they develop and propagate.[1] They found that a technology typically takes 15 to 20 years to be ready for popularization. They identify six typical phases:

1. *Basic research.* Investigate basic ideas and concepts, put an initial structure on the problem, and frame critical research questions.
2. *Concept formulation.* Circulate ideas informally, develop a research community, converge on a compatible set of ideas, solve specific subproblems, and refine the problem's structure.
3. *Development and extension.* Explore preliminary applications of the technology, clarify underlying ideas, and generalize the approach.
4. *Internal enhancement and exploration.* Extend the approach to other domains, use the technology for real problems, stabilize the technology, develop training materials, and show value in the results.

5. *External enhancement and exploration.* This resembles phase 4 but involves a broader community of people who weren't developers, show substantial evidence of value and applicability, and flesh out the details to provide a complete system solution.

6. *Popularization.* Develop production-quality, supported versions of the technology, commercialize and market the technology, and expand the user community.

As technologies mature, their institutional mechanisms for disseminating results also change. These mechanisms begin with informal discussions among colleagues and progress to products in the marketplace. Along the way, preliminary results of the first two phases appear in position papers, workshops, and research conferences. As the ideas mature, results appear in conferences and then journals; larger conferences set up tracks featuring the technology, and eventually richer streams of results may justify topical conferences. Books that synthesize multiple results help to move the technology through the exploration phases. University courses, continuing education courses, and standards indicate the beginning of popularization.

Maturation of software architecture

Software architecture overlaps and interacts with the study of software families, domain-specific design, component-based reuse, software design, specific classes of components, and program analysis. Trying to separate these areas rigidly isn't productive.

One way to see the field's growth is to examine the rate at which earlier results serve as building blocks for subsequent results. We can estimate this roughly by counting citations of papers with "software architecture" in the title. In a more comprehensive survey,[2] the one on which this article was based, we analyzed the results of a search for such papers in the CiteSeer database (http://citeseer.ist.psu.edu). Virtually all the cited papers were published in 1990 or later. The number of such citations increased steadily from 1991 to 1996 and sharply increased for papers published in 1998. The two dozen most widely cited books and papers were published between 1991 and 2000. They include

■ five books (1995–2000),
■ four papers presenting surveys or models of the field (1992–1997),

■ six papers dealing with architecture for particular domains (1991–1998),
■ seven formalizations (1992–1996), and
■ one paper each on an architectural description language and an analysis technique.

The major changes in this pattern since a similar count in 2001[3] are an increase in citations on formalizations and a substantial turnover in the most-cited papers about architectures for specific domains. This indicator is based on the published literature, so it naturally reflects the first three phases of development. Imperfect though this estimate might be, it still indicates substantial growth over the past decade or so and a balance between exploration of specific problems and development of generalizations and formalizations. Of the two dozen papers that were most commonly cited in 2001, 14 remain among the most commonly cited papers in 2005—indicating that the seminal sources have been identified.

Here are some highlights of the field's development. The chronology isn't as linear as the Redwine-Riddle model might suggest: different aspects of the field evolve at different rates, transitions between phases don't happen instantly, and publication dates lag the actual work by different amounts (see figure 1). Nevertheless, overall progress corresponds fairly well to their model. The phases' time spans are suggested by the dates of the work mentioned, discounting foundational work from the 1960s and 1970s.

Basic research: 1985–1993

For as long as complex software systems have been developed, designers have described their structures with box-and-line diagrams and informal explanations. Good designers recognized stylistic commonalities among these structures and exploited the styles in ad hoc ways. These structures were sometimes called architectures, but the knowledge of common styles—that is, generally useful structural forms—wasn't systematically organized or taught.

Significantly, by the mid-1980s several foundational concepts were firmly in place, having traveled their own 15- to 20-year Redwine-Riddle cycles. These included information hiding, abstract data types, and other ideas that considered software elements as black boxes. Object-oriented (OO) development was building on abstract data types and inheritance. These ideas all had their foundations on the ob-
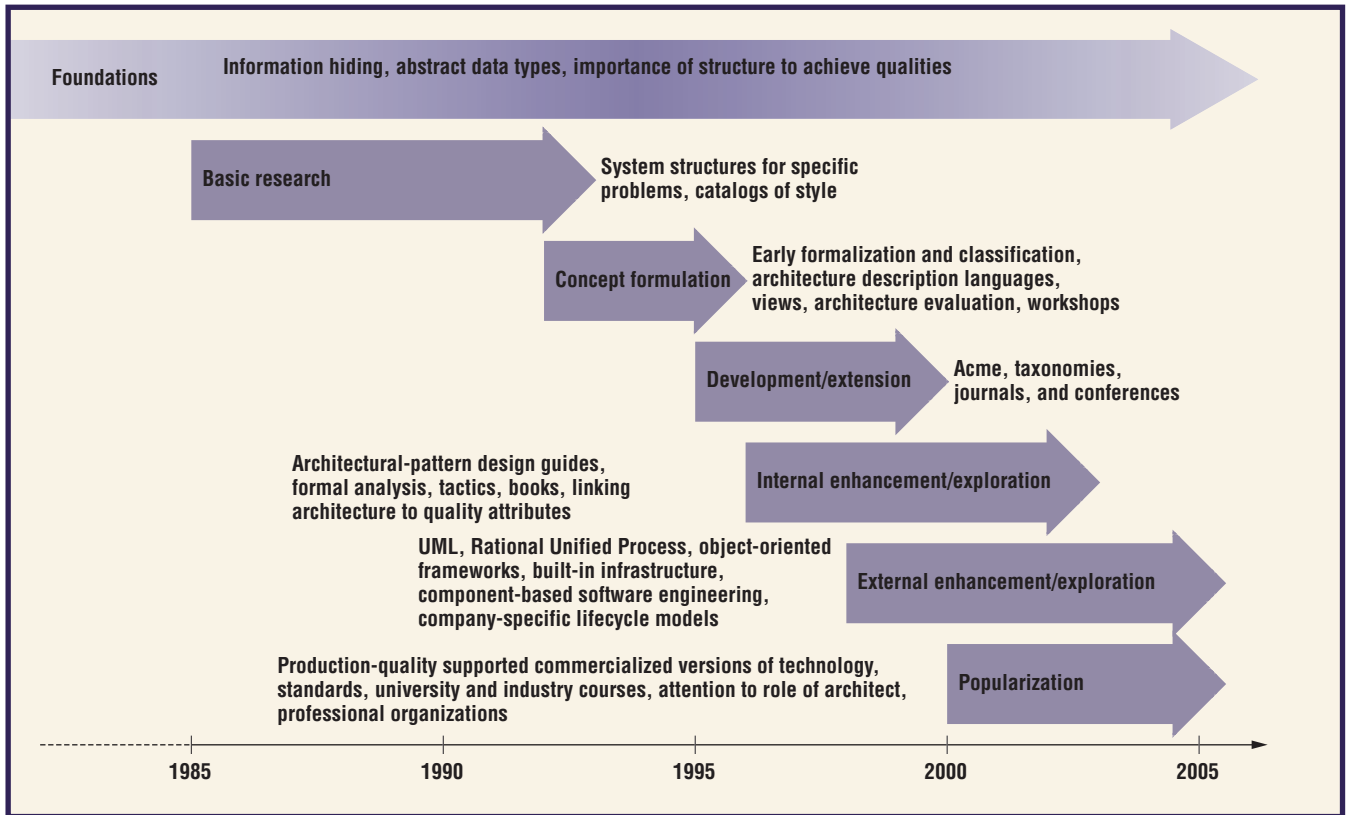
Figure I. Maturation of the software architecture field. Time spans for phases are suggested by the dates of the work mentioned in the corresponding sections, discounting foundational works from the I960s and I970s.

servation by Edsger Dijkstra, David Parnas, and others that it isn't enough for a computer program to produce the correct outcome. Other software qualities such as dependability and maintainability are also important and can be achieved by careful structuring.

In the late 1980s, people began to explore the advantages of deliberately designed, specialized software structures for specific problems. Some of this work addressed software system structures for particular product lines or application domains, such as avionics,[4] oscilloscopes,[5] and missile control.[6] Other work organized the informal knowledge about common software structures, or architectural styles, that can serve a variety of problem domains. This work cataloged existing systems to identify common architectural styles such as pipe-filter, repository, implicit invocation, and cooperating processes, both by identifying the architectures of specific classes of systems[7,8] and by finding general ways to describe such structures.[9–11] Complementary lines of research led to models for explaining architectural styles and to two widely cited papers in 1992 and 1993 that established the field's structure (and settled its name).[12,13]

Concept formulation: 1992–1996

Architecture description languages, early formalization, and classification helped elaborate and explore the models from the basic research phase. These early ideas centered on the system structures that commonly occur in software systems, and the results emphasized description of organizations found in practice.[12]

ADLs[14] served as a vehicle to flesh out specific details of various aspects of architecture, especially alternatives to the then-emerging object orientation.

Formalizations developed in parallel with language development. Sometimes this was integral to the language;[15–17] in other cases, it was more independent, as the formalization of style[18] or the formal analysis of a specific architectural model[19] or application area.[20,21] The recognition that architectural analysis must reconcile multiple views[22] helped to frame the requirements for formalism.

The early narrative catalogs of styles were expanded into taxonomies of styles[23] and of the elements that support those styles.[24] The common forms were cataloged and explained as patterns.[25] An early book on these ideas, *Software Architecture: Perspectives on*

> It isn't enough for a computer program to produce the correct outcome. Other software qualities are also important and can be achieved by careful structuring.

an Emerging Discipline, set the stage for further development.[26]

Understanding the relationship between architectural decisions and a system's quality attributes revealed software architecture validation as a useful risk-reduction strategy. Interconnectivity metrics and attribute-specific architecture analysis techniques[27] gave way to more general architecture evaluation methods such as the Software Architecture Analysis Method.[28]

Significant in this phase was the emergence of architectural views as a working concept. Parnas set the stage for this in 1974 by observing that software systems have many structures serving different engineering purposes and that selecting any one as distinguished makes little sense. After percolating for a Redwine-Riddle maturation period, the concept flowered in influential papers that firmly established views in architectural practice.[13,22,29]

Workshops on related topics (for example, the International Workshop on Software Specification and Design, http://portal.acm.org/toc.cfm?id=SERIES341) provided a temporary home for the software architecture community. A formative Dagstuhl seminar in 1995 (www.dagstuhl.de/9508/Report) gathered researchers to think about the field's layout and future directions. A series of International Software Architecture Workshops (associated with other conferences) from 1995 to 2000 provided a welcome, ongoing forum devoted solely to software architecture.

Development and extension: 1995–2000

During this phase, the focus shifted to unifying and refining initial results. The Acme architectural interchange language began with the goal of providing a framework to move information between ADLs[30]; it later grew to integrate other design, analysis, and development tools. Refinement of the taxonomies of architectural elements[31] and languages[14] also continued.

The field's institutions also matured. The IEEE's *Transactions on Software Engineering* had a special issue on software architecture in 1995. The special "Future of SE" track at the 22nd International Conference on Software Engineering (www.softwaresystems.org/future.html) in 2000 included software architecture among its topics, and now ICSE routinely has one or more sessions on architectural topics. A standalone conference, the Working IEEE/IFIP Conference on Software Architecture (www.wicsa.net), began in 1998 and continues to the present. One of its sponsors is a new International Federation for Information Processing working group on software architecture (www.softwarearchitectureportal.org).

Internal enhancement and exploration: 1996–2003

Informally, software designers typically use architectural styles (which during this stage came to be called architectural patterns to acknowledge their kinship with design patterns) as design guides. Explicit attention to this aspect of design is increasing, and as a result we're gaining experience.

A few real system designs have been formally analyzed as well. For example, architectural specification of the High-Level Architecture for Distributed Simulation[32] was able to identify inconsistencies before implementation, thereby saving extensive redesign.

Architectural analysis and evaluation emerged as a fertile subtopic. The Software Architecture Analysis Method[28] gave way to the Architecture Tradeoff Analysis Method,[33] which supports analysis of the interaction among quality attributes as well as the attributes themselves. Books on the application of the research to practice[34,35] set the stage for external exploration. Books on specialized parts of the practice such as architecture evaluation[36] and documentation[37] also emerged, signaling a new kind of maturation of the overall field.

During this stage, the importance of quality attributes increased, along with architecture's role in achieving them.[38] The early 2000s saw work strongly connecting quality attributes and architectural design decisions, and for the first time an automated architectural design aid seemed within reach.[39]

External enhancement and exploration: 1998–present

Several technologies have matured enough to be useful outside their developer groups.

UML, under the leadership of (at the time) Rational, integrated a number of design notations and developed a method for applying them systematically. UML has, for better or (many would say) worse, become the industry standard ADL. Tied inextricably to UML is the Rational Unified Process, a tool-centered

industrialization of Philippe Kruchten's original elegant idea of 4+1 views.[22] For the most part, UML provides graphical notations. It still lacks, however, a robust suite of tools for analysis, consistency checking, or other means of automatically connecting the information expressed in UML with the system's code.

The rise of OO software frameworks provided a rich development setting for object-style architecture and considerable public enthusiasm for object-orientedness. The benefits of a built-in infrastructure and available, interoperable components provide substantial incentive to use the frameworks even when they're not ideal fits for the problems. So, work on general-purpose ADLs gave way to extensive support for specific architectures. At about the same time, architecture provided a solid enough foundation to implicitly support the component-based software engineering movement.[40]

Also indicative of external enhancement are company-specific end-to-end architecture-based development life-cycle models, such as the Raytheon Enterprise Architecture Process (http://wwwxt.raytheon.com/technology_today/v3_i2/feature_ent_arch.html).

Popularization: 2000–present

The popularization phase is characterized by production-quality, supported, commercialized, and marketed versions of the technology, along with an expanded user community.

Architectural patterns, fueled in part by the explosion of the World Wide Web and Web-based e-commerce, are leading the commercialization wave. N-tier client-server architectures, agent-based architectures, and service-oriented architectures—along with the interfaces, specification languages, tools, and development environments, and wholly implemented components, layers, or subsystems to go along with them—are examples of enormously successful architectural patterns that have entered everyone's vocabulary. Microsoft says its .NET platform "includes everything a business needs to develop and deploy a Web service-connected IT architecture: servers to host Web services, development tools to create them, applications to use them, and a worldwide network of more than 35,000 Microsoft Certified Partner organizations to provide any help you need" (www.microsoft.com/net/basics.mspx). Connected services, tools, applica-

tions, platforms, and an army of vendors, all built around an architecture, represent true popularization.

One hallmark of a production-ready technology is good standards. Standards for particular component families (such as the Component Object Model, or COM, and CORBA) and interfaces (such as XML) have existed for several years, but they reflect component reuse interests as much as architectural interests. An ANSI/IEEE standard, IEEE-Std-1471-2000, has attempted to codify the current best practices and insights of both the systems and software engineering communities in the area of documentation. Newer standards are emerging all the time, primarily to support the important patterns mentioned earlier. Recently, the Society of Automotive Engineers standardized its Architecture Analysis and Design Language (AADL)—a true ADL—as SAE Standard AS5506 (www.sae.org/servlets/product Detail?PROD_TYP=STD&PROD_CD=AS5506).

One sure sign of a thriving user community is the degree to which people take ownership of the terms and concepts. Bill Gates, who could have any title he chooses, is Microsoft's "chief software architect." The Object Management Group chose to call its development initiative separating business and application logic from platform technology "model-driven architecture." In response to the Software Engineering Institute's invitation to submit working definitions of "software architecture," practitioners in 24 countries had submitted more than 156 definitions by late 2005 (www.sei.cmu.edu/architecture/definitions.html). Another sign is the way the term gets co-opted and diluted by people pulling their own interests under the currently popular umbrella. Terms such as "program architecture" make us shudder.

An institutional indicator of popularization is the availability of courses in the subject. In universities, software architecture is moving from graduate to undergraduate curricula; more than one textbook for introductory software engineering courses now includes a chapter on "architectural design." In the ACM/IEEE undergraduate software engineering curriculum (http://sites.computer.org/ccse/SE2004Volume.pdf), 20 percent of the software design unit is devoted to software architecture. The Software Engineering Body of Knowledge (www.swebok.org) identifies software architecture as

N-tier client-server architectures, agent-based architectures, and service-oriented architectures are examples of enormously successful architectural patterns.

a major section in its software design chapter. Industrial courses and certificate programs are also widely available.

Finally, "software architect" is a common job title in companies that build software-intensive systems, and professional organizations such as the Worldwide Institute of Software Architects (www.wwisa.org) and the International Association of Software Architects (www.iasahome.org) enable communication, foster networking, encourage professional practice, and (we hope) help their members sort out the avalanche of books—over 50—now available on the topic.

Conferences continue to thrive, not only for the research community but also for user networks. In early 2006, the SEI listed two dozen upcoming conferences that explicitly mention "software architecture" in their calls for participation (www.sei.cmu.edu/architecture/events.html). These include user network meetings as well as research conferences.

Current status

The broad concept of software architecture has run the full course of the Redwine-Riddle model, pretty much right on schedule. The result is a breathtaking capability for reliably designing systems of unprecedented size and complexity verging on a true engineering discipline. Consider the resources readily available to a contemporary software architect:

- *Off-the-shelf industrial training and certification programs* reflect a converging sense of what software architecture is and why it's a critical discipline.
- *Standard architectures* exist for countless domains and applications. For example, nobody will ever again have to design from scratch a banking system, an avionics system, a satellite ground system, or a Web-based e-commerce system.
- Where total architectural solutions don't yet exist, partial ones certainly do in the form of *catalogs of architectural patterns and tactics* that help solve a myriad of problems and achieve various quality attributes.
- *End-to-end life-cycle models* (industry-wide or, more likely, company-specific) build on architectural principles.
- *Architecture evaluation and validation methods* support robust and repeatable design approaches.

- Practical approaches to *architecture documentation* build on standards for artifacts and standards for languages in which to render the artifacts.
- *Robust tool environments* exist for developing designs.
- *Commercial-quality architectural infrastructure layers* can handle intercomponent communication and coordination of distributed generic computing environments.
- *Commercial-quality application layers* (and tooling) can handle business logic, user interface, and support function layers.
- Software architects have access to *career tracks* and *professional societies*.
- An active *pipeline of journals and conferences* devoted to software architecture serves as a conduit between the research and practice communities.

These and other factors indicate that software architecture is integrated in the fabric of software engineering.

What's next?

Software engineering research is often motivated by problems arising in producing and using real-world software. Technical ideas often begin as qualitative descriptions of problems or practice and gradually become more precise—and more powerful—as practical and formal knowledge grow in tandem. Thus, as some aspect of software development comes to be better understood, more powerful specification mechanisms become viable, in turn enabling more powerful technology.

Software architecture has followed this paradigm, growing from its adolescence in research laboratories to the responsibilities of maturity. This brings with it additional responsibility for researchers to show not just that new ideas are promising (a sufficient grounds to continue research) but also that they're effective (a necessary grounds to move into practice). For example, at one time a new ADL seemed to emerge almost monthly. Now someone proposing a new language must be able to answer the question, "Does what you're proposing have any chance of unseating UML? What tooling will you provide with it?"

Nevertheless, significant opportunities exist for new contributions in software architecture. Some of the more promising areas are the following:

- *Expanding formal relationships between architectural design decisions and quality attributes.* This could one day lead to a practical and sophisticated automated architecture design assistant. It could also enable earlier, more accurate predictions of the value a system would deliver to specific types of users.
- *Finding the right language in which to represent architectures.* UML 2.0 was a marginal improvement over its predecessor, but it still lacks basic architectural concepts such as *layer* or a faithful notion of *connector*. Also, it can't analyze interactions among views or make strong connections to code, and it too easily mixes design concepts with implementation directives.
- *Finding ways to assure conformance between architecture and code.* Lack of conformance dooms an architecture to irrelevance as the code sets out on its own independent trajectory. We might establish conformance by construction (via generation, refinement, and augmentation) or by extraction (analyzing an artifact statically or dynamically to determine its architecture). Early work exists in both approaches, but solutions are incomplete, especially in recovery and enforcement of runtime views and architectural rules that go beyond structure.
- *Rethinking our approach to software testing on the basis of software architecture.* An architecture can let us generate a wide variety of test plans, test cases, and other test artifacts. For code that originates in the architecture (such as implementations of connections and interaction mechanisms), automatic testing is possible. A strong model of architecture-based testing, backed by formal reasoning and easy-to-use tooling, could have a major economic impact on software system development.
- *Organizing architectural knowledge to create reference materials.* Mature engineering disciplines have handbooks and other reference materials that give engineers access to the field's systematic knowledge. Cataloging architectural patterns[26] is a first step in this direction, but we also need reference materials for domain-specific architectures and for analysis techniques. Grady Booch's handbook on software architecture (www.booch.com/

architecture/index.jsp) can provide important exemplars, but engineers also need reference material that organizes what we know about architecture into an accessible, operational body of knowledge.
- *Developing architectural support for systems that dynamically adapt to changes in resources and each user's expectations and preferences.* As computing becomes ubiquitous and integrated in everyday devices, both base resources such as bandwidth and information resources such as location-specific data change dynamically. Moreover, each user's needs change with time, and different users have different needs. Dynamically adapting to these changes would help to maximize the benefit to each user. Achieving this will require not only adaptive architectures but also component specifications that reflect variability in user needs as well as the component's intrinsic properties.

I t will be interesting to see how these ideas fare over the next 10 years or—more likely—what ideas now undreamed of will have emerged. But one thing seems clear. The last decade and a half has seen a phenomenal growth of software architecture as a discipline. It started in the late 1980s as an academic idea aimed at understanding and codifying system descriptions observed in industrial practice. From there it has grown to a relatively mature engineering discipline complete with standard and repeatable practices, a rich catalog of prepackaged design solutions, an enormous commercial market supplying tools and components, and a universal recognition that software architecture is an indispensable part of software system development.

A "golden age" is a period of prosperity and excellent achievement, often marked by numerous advances that rapidly move the technology from speculative to dependable. The last 15 years or so—roughly the middle four stages of the Redwine-Riddle model— truly have been software architecture's golden age. Like the golden age of air travel in the 1930s, it's been an exciting time of discovery, unfettered imagination, great progress, great setbacks, and a sense of the possible.

But all golden ages come to a close, and as software architecture moves from being novel

The last 15 years or so— roughly the middle four stages of the Redwine–Riddle model—truly have been software architecture's golden age.

About the Authors

Mary Shaw is the Alan J. Perlis Professor of Computer Science, codirector of the Sloan Software Industry Center, and member of the Institute for Software Research International and the Computer Science Department at Carnegie Mellon University. Her research interests include software engineering and programming systems, particularly value-driven software design, support for everyday users, software architecture, programming languages, specifications, and abstraction techniques. She received her PhD in computer science from Carnegie Mellon University. She received the Stevens Award for instrumental contributions in the development and recognition of software architecture as a discipline and the Warnier prize for contributions to software engineering. She is a fellow of the ACM, the IEEE, and the American Association for the Advancement of Science. She's also a member of the IFIP Working Group 2.10 (Software Architecture). Contact her at the Inst. for Software Research Int'l, Carnegie Mellon Univ., Pittsburgh, PA 15213; mary.shaw@cs.cmu.edu; www.cs.cmu.edu/~shaw.

Paul Clements is a senior member of the technical staff at Carnegie Mellon University's Software Engineering Institute, where he has led or co-led projects in software product line engineering and software architecture documentation and analysis. His most recent book is *Software Architecture in Practice* (2nd ed., 2003). He received his PhD in computer sciences from the University of Texas at Austin. He's a member of the IFIP Group 2.10 (Software Architecture). Contact him at the Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, PA 15213; clements@sei.cmu.edu.

to being indispensable, its golden age is receding. This is as it should be, because software architecture is entering a period where it can be taken for granted. We rely on it, we can't imagine our technological culture without it, and we're compelled to continually refine and improve it. This is the hoped-for fate of all successful technologies, and it's a time for deep satisfaction. It doesn't mean that the time for research, innovation, and improvement has passed. On the contrary, the excitement is now that of adulthood rather than of childhood: The strong foundations laid by the early phases of software architecture maturation, coupled with ongoing research to make new ideas practical, will enable even more breathtaking system-building capabilities in the future. For us, the intriguing question is this: What new software engineering technology and *its* golden age will the solidly established field of software architecture help to usher in? ⦿

References

1. S. Redwine and W. Riddle, "Software Technology Maturation," *Proc. 8th Int'l Conf. Software Eng.*, IEEE CS Press, 1985, pp. 189–200.
2. M. Shaw and P. Clements, *The Golden Age of Software Architecture: A Comprehensive Survey*, tech. report CMU-ISRI-06-101, Inst. for Software Research Int'l, Carnegie Mellon Univ., Feb. 2006.
3. M. Shaw, "The Coming-of-Age of Software Architecture Research," *Proc. 23rd Int'l Conf. Software Eng.*, IEEE CS Press, 2001, pp. 656–664a.
4. D.L. Parnas, P.C. Clements, and D.M. Weiss, "The Modular Structure of Complex Systems," *IEEE Trans. Software Eng.*, vol. SE-11, no. 3, 1985, pp. 259–266.
5. N. Delisle and D. Garlan, "Formally Specifying Electronic Instruments," *Proc. 5th Int'l Workshop Software Specification and Design*, ACM Press, 1989, pp. 242–248.
6. E. Mettala and M. Graham, eds., *The Domain-Specific Software Architecture Program*, tech. report CMU/SEI-92-SR-9, Software Eng. Inst., Carnegie Mellon Univ., 1992.
7. G. Andrews, "Paradigms for Process Interaction in Distributed Programs," *ACM Computing Surveys*, vol. 23, no. 1, 1991, pp. 49–90.
8. H.P. Nii, "Blackboard Systems," *AI Magazine*, vol. 7, no. 3, 1986, pp. 38–53, and vol. 7, no. 4, 1986, pp. 82–107.
9. M. Shaw, "Toward Higher-Level Abstractions for Software Systems" (invited), *Proc. Tercer Simposio Internacional del Conocimiento y su Ingenieria*, Rank Xerox, 1988, pp. 55–61. Reprinted in *Data and Knowledge Eng.*, vol. 5, no. 2, 1990, pp. 119–128; http://portal.acm.org/citation.cfm?id=87367.
10. M. Shaw, "Elements of a Design Language for Software Architecture," position paper for *IEEE Design Automation Workshop*, Jan. 1990.
11. M. Shaw, "Heterogeneous Design Idioms for Software Architecture," *Proc. 6th Int'l Workshop Software Specification and Design*, IEEE Press, 1991, pp. 158-165.
12. D. Garlan and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, vol. 1, World Scientific, 1993, pp. 1–39.
13. D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Eng. Notes*, Oct. 1992, pp. 40–52.
14. N. Medvidovic and R.N. Taylor, "A Framework for Classifying and Comparing Architecture Description Languages," *Proc. 6th European Software Eng. Conf.*, LNCS 1301, Springer, 1997, pp. 60–76.
15. J. Magee and J. Kramer, "Dynamic Structure in Software Architectures," *Proc. ACM SIGSOFT 96, 4th Symp. Foundations Software Eng.* (FSE4), ACM Press, 1996, pp. 3–14; http://portal.acm.org/citation.cfm?id=239104.
16. D.C. Luckham et al., "Specification and Analysis of System Architecture Using Rapide," *IEEE Trans. Software Eng.*, vol. 21, no. 4, 1995, pp. 336–355.
17. R. Allen and D. Garlan, "A Formal Approach to Software Architectures," *Proc. IFIP 92*, Elsevier, 1992, pp. 134–141.
18. G. Abowd, R. Allen, and D. Garlan, "Formalizing Style to Understand Descriptions of Software Architecture," *ACM Trans. Software Eng. and Methodology*, vol. 4, no. 4, 1995, pp. 319–364.
19. K. Sullivan, M. Marchukov, and D. Socha, "Analysis of a Conflict between Interface Negotiation and Aggregation in Microsoft's Component Object Model," *IEEE Trans Software Eng.*, vol. 25, no. 4, 1999, pp. 584–599.
20. C. Locke, "Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives," *J. Real-Time Systems*, vol. 4, no. 1, 1992, pp. 37–53.
21. M.R. Macedonia et al., "NPSNET: A Network Software

Architecture for Large Scale Virtual Environments," *Presence, Teleoperators, and Virtual Environments*, vol. 3, no. 4, 1994, pp. 265–287.

22. P. Kruchten, "The 4+1 View Model of Software Architecture," *IEEE Software*, vol. 12, no. 6, 1995, pp. 42–50.

23. M. Shaw and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," *Proc. COMPSAC 97: Int'l Computer Software and Applications Conf.*, IEEE CS Press, 1997, pp. 6–13.

24. R. Kazman et al., "Classifying Architectural Elements as a Foundation for Mechanism Matching," *Proc. COMPSAC 97: Int'l Computer Software and Applications Conf.*, IEEE CS Press, 1997, pp. 14–17.

25. F. Buschmann et al., *Pattern-Oriented Software Architecture—A System of Patterns*, John Wiley & Sons, 1996.

26. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

27. C. Smith, "Performance Engineering," *Encyclopedia of Software Eng.*, vol. 2, John Wiley & Sons, 1994, pp. 794–810.

28. R. Kazman et al., "SAAM: A Method for Analyzing the Properties of Software Architectures," *Proc. 16th Int'l Conf. Software Eng.* (ICSE 94), IEEE CS Press, 1994, pp. 81–90.

29. D. Soni, R. Nord, and C. Hofmeister. "Software Architecture in Industrial Applications," *Proc. 17th Int'l Conf. Software Eng.*, ACM Press, 1995, pp. 196–207.

30. D. Garlan, R. Monroe, and D. Wile, "Acme: An Architecture Description Interchange Language," *Proc. CASCON 97*, ACM Press, 1997, pp. 169–183.

31. N.R. Mehta, N. Medvidovic, and S. Phadke, "Towards a Taxonomy of Software Connectors," *Proc. 22nd Int'l Conf. Software Eng.* (ICSE 00), IEEE CS Press, 2000, pp. 178–187.

32. R. Allen, D. Garlan, and J. Ivers, "Formal Modeling and Analysis of the HLA Component Integration Standard," *Proc. 6th Int'l Symp. Foundations of Software Eng.*, ACM Press, 1998.

33. R. Kazman et al., "Experience with Performing Architecture Tradeoff Analysis," *Proc. 21st Int'l Conf. Software Eng.* (ICSE 99), IEEE CS Press, 1999, pp. 54–63.

34. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998; 2nd ed., 2003.

35. C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*, Addison-Wesley, 1999.

36. P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2001.

37. P. Clements et al., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2002.

38. M.R. Barbacci, M.H. Klein, and C.B. Weinstock, *Principles for Evaluating the Quality Attributes of a Software Architecture*, tech. report CMU/SEI-96-TR-036, Software Eng. Inst., Carnegie Mellon Univ., 1996.

39. F. Bachmann, L. Bass, and M. Klein, *Preliminary Design of ArchE: A Software Architecture Design Assistant*, tech. report CMU/SEI-2003-TR-021, Software Eng. Inst., Carnegie Mellon Univ., 2003.

40. C. Szyperski, *Component Software—Beyond Object-Oriented Programming*, Addison-Wesley, 1997.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.