In *Proceedings of the IPSJ Object-Oriented Symposium 2000*
August 2000, Tokyo, Japan.

Software Architecture and Object-Oriented Systems

David Garlan
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
(412) 268-5056
garlan@cs.cmu.edu

ABSTRACT
Over the past decade software architecture has emerged as an important subfield of software engineering. During that time there has been considerable progress in developing the technological and methodological base for treating architectural design as an engineering discipline, including specialized architectural description languages, tools, analytic techniques, handbooks, and methods. In this paper I survey the main features of the field, and then compare software architecture with object-oriented systems to illustrate some of the important similarities and differences between the two fields.

Keywords
Software architecture, software design, software engineering, object-oriented systems

1 INTRODUCTION

A critical issue in the design and construction of complex software systems is software architecture: that is, a system's organization as a collection of interacting components. A well-designed architecture can help ensure that a system will satisfy its key requirements, particularly with respect to system-wide properties, such as performance, reliability, portability, scalability, and interoperability. A badly-designed architecture can be disastrous.

Over the past decade software architecture has received increasing attention as an important subfield of software engineering. Practitioners have come to realize that getting an architecture right is a critical for successful system design and development. They have begun to recognize the importance of making explicit architectural choices, and leveraging past architectural designs in the development of new products. There are now numerous books on architectural design, regular conferences and workshops devoted specifically to software architecture, a growing number of commercial tools to aid in aspects of architectural design, courses in software architecture, major government and industrial research projects centered on software architecture, and an increasing number of formal architectural standards. Codification of architectural principles, methods, and practices has begun to lead to repeatable processes of architectural design, criteria for making principled tradeoffs among architectures, and standards for documenting, reviewing, and implementing architectures.

In parallel, object-oriented systems have come of age. Object-oriented programming languages, methods, tools, modeling notations, and handbooks are now common in industry. Component-based systems, adopting a distinct object-oriented flavor, are revolutionizing our ability to compose systems from parts. Object development methods are becoming *de rigueur* in many parts of the software industry.

An interesting question concerns the relationship between the two fields. Is software architecture merely a subfield of object-oriented design? Or vice versa? How should the two disciplines co-exist? And, most importantly, what kinds of synergies will best empower practitioners to build complex systems more effectively.

In this paper I begin by examining some of the important developments of software architecture in both research and practice. First I describe the roles of architecture in software systems development. Next I summarize the past and current state of research and practice. I then compare software architecture with object-oriented systems to illustrate some of the important similarities and differences between the two fields, and suggest possibilities for future interplay between these areas.

2 THE ROLES OF SOFTWARE ARCHITECTURE

There are numerous definitions of software architecture. However, at the core of all of them is the notion that the architecture of a system defines its gross structure. This structure illuminates the top level design decisions, includeing things such as how the system is composed of interacting parts, where are the main pathways of interaction, and

what are the key properties of the parts. Additionally, an architectural description often includes sufficient information to allow high-level analysis and critical appraisal of the system design.

Software architecture typically plays a key role as a bridge between requirements and implementation (see Figure 1).

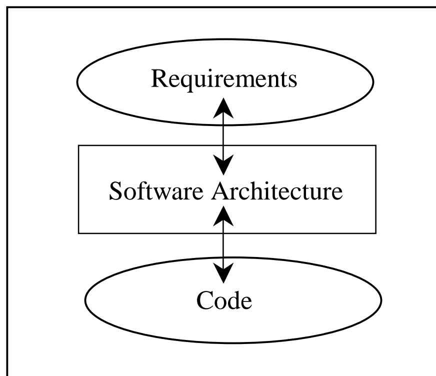

Figure 1: Software Architecture as a Bridge

By providing an abstract description of a system, the architecture exposes certain properties, while hiding others. Ideally this representation provides an intellectually tractable guide to the overall system, permits designers to reason about the ability of a system to satisfy certain requirements, and suggests a blueprint for system construction and composition. For example, an architecture for a signal processing application might be constructed as a dataflow network in which the nodes read input streams of data, transform that data, and write to output streams. Designers might use this decomposition, together with estimated values for input data flows, computation costs, and buffering capacities, to reason about possible bottlenecks, resource requirements, and schedulability of the computations.

To elaborate, software architecture plays an important role in at least five aspects of software development. First, it helps *understanding: s*oftware architecture simplifies our ability to comprehend large systems by presenting them at a level of abstraction at which a system's high-level design can be easily understood [20, 35]. Second, it supports *reuse:* architectural design supports both reuse of large components, as well as *frameworks* into which components can be integrated [8, 31]. Third, it supports *construction:* An architectural description provides a partial blueprint for development by indicating the major components and dependencies between them. Fourth, it supports e*volution:* software architecture can expose the dimensions along which a system is expected to evolve. Fifth, it supports *analysis:* architectural descriptions provide new opportunities for checking system properties, such as system consistency [2, 25], conformance to constraints imposed by an architectural style [1], achievement of quality attributes [9], dependence analysis [42, 45, and domain-specific properties of architectures built in specific styles [10, 15, 26].

3 EVOLUTION OF THE FIELD

Until recently architectural design and documentation was largely an ad hoc affair.[1] Descriptions relied on informal box-and-line diagrams, which were rarely maintained once a system was constructed. Architectural choices were made in idiosyncratic fashion – typically by adapting some previous design, whether or not it was appropriate. Good architects – even if they were classified as such within their organizations – learned their craft by hard experience in particular domains, and were unable to teach others what they knew. It was usually impossible to analyze an architectural description for consistency or to infer non-trivial properties about it. There was no way to check that a system implementation faithfully represented its architectural design.

Despite their informality, architectural descriptions have long been central to successful system design. As people began to understand the critical role that architectural design plays in determining system success, they also began to recognize the need for a more disciplined approach. Early authors began to observe certain unifying principles in architectural design [36], to call out architecture as a field in need of attention [35], and to establish a working vocabulary for software architects [20]. Tool vendors began thinking about explicit support for architectural design. Language designers began to consider notations for architectural representation [30].

Within industry, two trends highlighted the importance of architecture. First was the recognition of a shared repertoire of methods, techniques, patterns and idioms for structuring complex software systems. For example, the box-and-line-diagrams and explanatory prose that typically accompany a high-level system description often refer to such organizations as a "pipeline," a "blackboard-oriented design," or a "client-server system." Although these terms were rarely assigned precise definitions, they permitted designers to describe complex systems using abstractions that make the overall system intelligible. Moreover, they provided significant semantic content about the kinds of properties of concern, the expected paths of evolution, the overall computational paradigm, and the relationship between this system and other similar systems.

The second trend was to exploit commonalities in specific domains to provide reusable frameworks for *product fami-*

[1] There were some notable exceptions: Parnas recognized the importance of system families [33], and architectural decomposition principles based on information hiding [34]. Others, such as Dijkstra, exposed various system structuring principles [12].

lies. Such exploitation is based on the idea that common aspects of a collection of related systems can be extracted so that each new system can be built at relatively low cost by "instantiating" the shared design. Familiar examples include the standard decomposition of a compiler (which permits undergraduates to construct a new compiler in a semester), standardized communication protocols (which allow vendors to interoperate by providing services at different layers of abstraction), fourth-generation languages (which exploit the common patterns of business information processing), and user interface toolkits and frameworks (which provide both a reusable framework for developing interfaces and sets of reusable components, such as menus and dialogue boxes).

Over the past decade much has changed. While there remains wide variation in the state of the practice, architecture is much more visible as an important and explicit design activity in software development. Job titles now routinely reflect the role of software architect; companies rely on architectural design reviews as critical staging points; and architects recognize the importance of making explicit tradeoffs within the architectural design space.

In addition, the technological basis for architectural design has improved dramatically. Three of the important advancements have been the development of architecture description languages and tools, the emergence of product line engineering and architectural standards, and the codification and dissemination of architectural design expertise.

4.1 Architecture Description Languages and Tools

The informality of most box-and-line depictions of architectural designs leads to a number of problems: The intended meaning of the design may not be clear. Informal diagrams usually cannot be analyzed for consistency, completeness, or correctness. Architectural constraints assumed in the initial design are not enforced as a system evolves. There are few tools to help architectural designers with their tasks.

In response to these problems researchers in industry and academia have proposed a number of formal notations for representing and analyzing architectural designs. Commonly referred to as "Architecture Description Languages" (ADLs), these notations provide both a conceptual framework and a concrete syntax for characterizing software architectures [9, 30]. They also typically provide tools for parsing, displaying, compiling, analyzing, or simulating architectural descriptions.

Examples of ADLs include Adage [10], Aesop [15], C2 [28], Darwin [26], Rapide [25], SADL [32], UniCon [39], Meta-H [6], and Wright [3]. While all of these languages are concerned with architectural design, each provides certain distinctive capabilities: Adage supports the description of architectural frameworks for avionics navigation and guidance; Aesop supports the use of architectural styles; C2 supports the description of user interface systems using an event-based style; Darwin supports the analysis of distributed message-passing systems; Meta-H provides guidance for designers of real-time avionics control software; Rapide allows architectural designs to be simulated, and has tools for analyzing the results of those simulations; SADL provides a formal basis for architectural refinement; UniCon has a high-level compiler for architectural designs that supports a mixture of heterogeneous component and connector types; Wright supports the formal specification and analysis of interactions between architectural components.

Recently the proliferation of capabilities of ADLs has prompted an investigation of ways to integrate the notations and tools into larger ensembles. One of the results has been an architectural interchange language, called Acme, which provides a simple framework for describing architectural structure and a flexible annotation mechanism for adding semantics to that structure [18]. (Acme can be viewed as the XML of architectural description.) Acme also supports the definition of styles and enforcement of design constraints through its tools.

Although these languages (and their tools) differ in many respects, a number of key insights have emerged through their development. In particular, it is becoming clear that in its most general form, architectural design requires the ability to model the following as first class design entities:

- **Components:** Architecture is about compositions of systems from components. Components can be quite diverse and complex: databases, clients, servers, entire user interfaces, blackboards, etc.
- **Connectors:** In addition to components, architectural design requires the ability to define new forms of component *interactions.* Such interactions go beyond simple procedure call, permitting description of complex forms of communication as new component integration mechanisms.
- **Styles:** An architectural style defines a design vocabulary and specifies a set of constraints on how that vocabulary can be used. Styles allow the architect to specialize the design task to specific domains or products, and provide improved opportunities for system analysis.
- **Representations:** Architectural descriptions are typically hierarchical. It must be possible to associate more detailed representations (or models) with the individual parts of an architecture.
- **Visualizations:** Different architectural design domains require different visual conventions.
- **Extra-functional properties:** To support analysis of system properties it must be possible to model key properties of the parts of an architecture, and determine from those parts the overall properties

of the system. Typical properties include performance, reliability, and modifiability.

4.2 Product Lines and Standards

As noted earlier, one of the important trends in industry has been the desire to exploit commonality across multiple products. Two specific manifestations of that trend are improvements in our ability to create product lines within an organization and the emergence of cross-vendor integration standards.

With respect to product lines, a key challenge is that a product line approach requires different methods of development. In a single-product approach the architecture must be evaluated with respect to the requirements of that product alone. Moreover, single products can be built independently, each with a different architecture.

With a product line approach, in contrast, one must also consider requirements for the *family* of systems, and the relationship between those requirements and the ones associated with each particular instance. Figure 3 illustrates this relationship. In particular, there must be an up-front (and on-going) investment in developing a reusable architecture that can be instantiated for each product. Other reusable assets, such as components, test suites, tools, etc., typically accompany this.

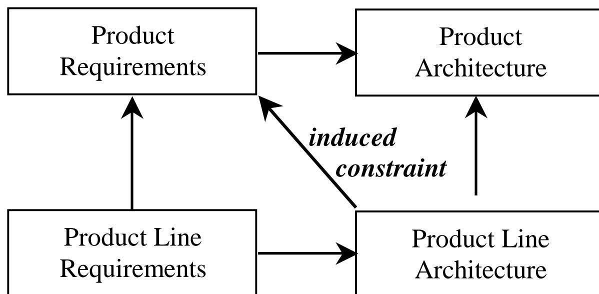

Figure 2: Product Line Architectures

Although product line engineering is not yet widespread, we are beginning to have a better understanding of the processes, economics, and artifacts required to achieve the benefits of a product line approach. A number of case studies of product line successes have been published. (For example, see [13].) Moreover, organizations such as the Software Engineering Institute are well on their way towards providing concrete guidelines and processes for the use of a product line approach [37].

Like product line approaches, cross-vendor integration standards require architectural frameworks that permit a system developer to configure a wide variety of specific systems by instantiating that framework. Integration standards typically provide the system glue (both conceptually and through run time infrastructure) that supports integration of parts provided by multiple vendors. Such standards may be formal international standards (such as those sponsored by IEEE or ISO), or ad hoc and de facto standards promoted by an industrial leader.

A good example of the former is the High Level Architecture (HLA) for Distributed Simulation [4]. This architecture permits the integration of simulations produced by many vendors. It prescribes interface standards defining services to coordinate the behavior of multiple semi-independent simulations. In addition, the standard prescribes requirements on the simulation components that indicate what capabilities they must have, and what constraints they must observe on the use of shared services.

An example of an ad hoc standard is Sun's Enterprise JavaBeans™ (EJB) architecture [27]. EJB is intended to support distributed, Java-based, enterprise-level applications, such as business information management systems. Among other things, it prescribes an architecture that defines a vendor-neutral interface to information services, including transactions, persistence, and security. It thereby supports component-based implementations of business processing software that can be easily retargeted to different implementations of those underlying services.

4.3 Codification and Dissemination

One early impediment was the lack of a shared body of knowledge about architectures and techniques for developing good ones. Today the situation has improved, due in part to the publication of books on architectural design [5, 8, 22, 36, 40] and courses [21].

A common theme in these books and courses is the use of standard architectural *styles*. As noted earlier, an architectural style typically specifies a design vocabulary, constraints on how that vocabulary is used, and semantic assumptions about that vocabulary. For example, a pipe-and-filter style might specify vocabulary in which the processing components are data transformers (filters), and the interactions are via order-preserving streams (pipes). Constraints might include the prohibition of cycles. Semantic assumptions might include the fact that pipes preserve order and that filters are invoked non-deterministically.

Other common styles include blackboard architectures, client-server architectures, event-based architectures, and object-based architectures. Each style is appropriate for certain purposes, but not for others. For example, a pipe-and-filter style would likely be appropriate for a signal processing application, but not for an application in which there is a significant requirement for concurrent access to shared data [38]. Moreover, each style is typically associated with a set of associated analyses. For example, it

makes sense to analyze a pipe-and-filter system for system latencies, whereas transaction rates would be a more appropriate analysis for a repository-oriented style.

The identification and documentation of such styles (as well as their more domain-specific variants) enables others to adopt previous architectural patterns as a starting point. In that respect, the architectural community has paralleled other communities in recognizing the value of established, well-documented patterns, such as those found in [14].

While recognizing the value of stylistic uniformity, realities of software construction often force one to compose systems from parts that were not architected in a uniform fashion. For example, one might combine a database from one vendor, with middleware from another, and a user interface from a third. In such cases the parts do not always work well together – in large measure because they make conflicting assumptions about the environments in which they were designed to work [16]. This has led to recognition of the need to identify architectural strategies for bridging mismatches. Although, we are far from having well understood ways of detecting such mismatch, and of repairing it when it is discovered, a number of techniques have been developed [11].

4 OBJECT-ORIENTED SYSTEMS

Of course, software architecture is not the only sub-field of software engineering in which dramatic changes in languages, technology, and software practice have taken place. Object-oriented systems are area that has had even more impact on industrial practice over the past decade.

There are a number of obvious parallels between the evolution of object-oriented design techniques and the trends of software architecture, outlined above.

- **Description Languages and Tools:** Object-oriented systems have long had design languages and tools to support their use. Recently UML has emerged as a standard notation, unifying many of its predecessors. Increasingly vendors are developing tools that take advantage of this technological standardization.
- **Product Lines and Standards:** Object-oriented frameworks have long been an important point of leverage in system development. In particular, component-oriented integration mechanisms, such as Corba, COM, JavaBeans have played an important role in supporting integration of object-oriented parts. In other more domain-specific ways, frameworks like Enterprise JavaBeans™, VisualBasic™, and MFC™, have helped improve productivity in specific areas.
- **Codification and Dissemination:** There has been considerable work and interest in object-oriented patterns, which serve to codify common solutions to implementation problems.

Perhaps more importantly, object-oriented design techniques have always attempted to provide a clear path from requirements to implementation. To the extent that they support conceptual design of systems, they also address architectural concerns.

Given these similarities it is worth asking the question: what are the important differences between the two fields? There are some who might argue that architecture is simply a part of object-oriented development. Others might argue the opposite: that object-oriented design is simply a special form of architectural design. The answer to this question obviously has implications about the utility of object modeling notations for architectural design.

To shed light on the issue, I believe it is helps to view the relationship between architecture and object-oriented design from at least three distinct perspectives.

1. *OO as an architectural style:* This perspective treats the part of object-oriented development that is concerned with system structure as the special case of architectural design in which the components are objects and the connectors are procedure calls (method invocation). Some ADLs support this view, providing built-in primitives for inter-component procedure call.
2. *OO as an implementation base:* This perspective treats object-oriented development as a lower-level activity, more concerned with implementation. Viewed this way, object modeling becomes one viable implementation target for any architectural design.
3. *OO as an architectural modeling notation:* This perspective treats a notation such as UML as a suitable notation for all architectural descriptions. Proponents of this perspective have advocated various ways of using object modeling, including class diagrams, collaboration diagrams, and package diagrams [44]. From this perspective, architecture is viewed as a sub-activity of object-oriented design.

Which of these is most accurate? I believe that all are reasonable interpretations, although in each case there are mismatches. For the first perspective, there are many aspects of object-oriented design that are not captured well by ADLs. For the second perspective, it is clear that not all object-oriented modeling is related to implementation. For the third perspective, there are aspects of architectural design that are currently not handled well in notations such as UML. (See [44] for a discussion of some of these.)

Another way to understand the relationship is to ask what lessons each discipline can offer the other.

Lessons from Architecture:

1. *Object-oriented interfaces are not sufficient.* Most ADLs take the view that the interface descriptions of a

component should contain much more information than a list of provided procedures. In particular, one must also identify the aspects of the environment that the component depends on.
2. *Quality attributes are central.* Most ADLs support description and analysis of extra-functional properties such as performance and reliability. For architectural design such properties are as important (if not more so) than what is being computed.
3. *Connectors need not be procedure calls.* Most ADLs support a higher level notion of interaction than a set of procedure calls. As argued earlier, the ability to define new abstractions for system composition is key to providing higher-level views of system structure.

Lessons from Object-oriented Systems:
1. *Linkage to code is essential.* Many ADLs provide only high-level models, without any ways to relate those models to source code. The success of OO tools has shown that such linkages are important to preserve the integrity of the architectural design as the system evolves over time.
2. *Many views are needed.* Object modeling notations support multiple views. Such views are important because different aspects of a system (e.g., behavior versus structure) have different requirements for description.
3. *Methods must accompany notations.* Object-oriented systems design has long advocated specific development methods. Software architecture could learn much by adapting some of those methods to architecture-based design.
4. *Standards can foster community cohesion.* The object-oriented community has come together around UML as a standard notation. The architecture community is currently much less cohesive, leading to a proliferation of notations and approaches.

Problems that both disciplines share: Both software architecture and object-oriented systems design share a number of technical challenges. These define potential areas in which synergy may take place in the next few years.

- *Patterns:* Both need better support for description and use of patterns. It is not clear yet how best to describe the dimensions of variability in patterns, or how to automate checks for conformance to patterns.
- *Views:* Given a multi-view approach one would ideally like tools to check for consistency between the views of a system description.
- *Dynamism:* Modern systems evolve structurally during runtime. It is not clear how one should specify or analyze that dynamism using design models of a system.
- *Analysis:* Although considerable progress has been made in architectural analysis techniques, there remains much to be done to provide engineers with general, easy-to-use analysis techniques in areas like security, performance, and reliability.

5 CONCLUSION

The field of software architecture is one that has experienced considerable growth over the past decade, and it promises to continue that growth for the foreseeable future. Much of that grown has centered on areas of description and analysis, product-line development, and codification of architectural wisdom.

The parallel development of object notations and methods raises some interesting questions about the relationships between the two areas. While it may be tempting to try to reduce the problems of one area to those of another, in the end there remain numerous differences between the two approaches. These suggest useful lessons and possible areas of future synergy.

ACKNOWLEDGEMENTS

I would like to thank past and present members of the ABLE Research Group, Bob Balzer, Barry Boehm, Paul Clements, Dewayne Perry, John Salasin, Bran Selic, Mary Shaw, Dave Wile, and Alex Wolf for their help over the past years in clarifying the nature of software architecture and its relationship to object-oriented systems. Research contributing to this paper was sponsored by DARPA under contracts F30602-97-2-0031, F30602-96-1-0299, F30602-96-2-0224, and F33615-93-1-1330; by NSF under contracts CCR-9109469, CCR-9633532, and CCR-9357792; and by industrial support from HP, Siemens Corporation, and Kodak Corporation.

Mary Shaw provided some of the ideas and text for the section on pervasive computing architectures. The section describing the roles of software architecture was adapted from an introductory article on software architecture co-authored with Dewayne Perry [19].

REFERENCES

1. G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering.* ACM Press, December 1993.

2. R. Allen and D. Garlan. Formalizing architectural connection. In *Proceeding of the 16th International Conference on Software Engineering*, pages 71-80. Sorrento, Italy, May 1994.

3. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

4. S. Bachinsky, L. Mellon, G. Tarbox, and R. Fujimoto. RTI 2.0 architecture. In *Proceedings of the 1998 Spring Simulation Interoperability Workshop*, 1998.

5. L. Bass, P. Clements and R. Kazman. *Software Architecture in Practice.* Addison Wesley, 1099, ISBN 0-201-19930-0.

6. P. Binns and S. Vestal. Formal real-time architecture specification and analysis. *10th IEEE Workshop on Real-Time Operating Systems and Software*, May 1993.

7. B. Boehm, P. Bose, E. Horowitz and M. J. Lee. Software requirements negotiation and renegotiation aids: A theory-W based spiral approach. In *Proc of the 17th International Conference on Software Engineering*, 1994.

8. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

9. P. Clements, L. Bass, R. Kazman and G. Abowd. Predicting software quality by architecture-level evaluation. In *Proceedings of the Fifth International Conference on Software Quality*, Austin, Texas, Oct, 1995.

10. L. Coglianese and R. Szymanski, DSSA-ADAGE: An Environment for Architecture-based Avionics Development. In *Proceedings of AGARD'93*, May 1993.

11. R. Deline. *Resolving Packaging Mismatch*. PhD thesis, Carnegie Mellon University, December 1999.

12. E. W. Dijkstra. The structure of the "THE" – multiprogramming system. *Communications of the ACM*, 11(5):341-346, 1968.

13. P. Donohoe, editor. *Software Architecture: TC2 First Working IFIP Conference on Software Architecture (WICSA1)*. Kluwer Academic Publishers, 1999.

14. E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1995.

15. D. Garlan, R. Allen and J. Ockerbloom. Exploiting style in architectural design environments. In *Proc of SIGSOFT'94: The second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 170-185. ACM Press, December 1994.

16. D. Garlan, R. Allen and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17-28, November 1995.

17. D. Garlan, A. J. Kompanek and P. Pinto. Reconciling the needs of architectural description with object-modeling notations. Technical report, Carnegie Mellon University, December 1999.

18. D. Garlan, R. T. Monroe and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169-183, Ontario, Canada, November 1997.

19. D. Garlan and D. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

20. D. Garlan and M. Shaw. An Introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1-39, Singapore, 1993. World Scientific Publishing Company.

21. D. Garlan, M. Shaw, C. Okasaki, C. Scott, and R. Swonger. Experience with a course on architectures for software systems. In *Proceedings of the Sixth SEI Conference on Software Engineering Education.* Springer Verlag, LNCS 376, October 1992.

22. C. Hofmeister, R. Nord and D. Soni. *Applied Software Architecture*. Addison Wesley, 2000.

23. C. Hofmeister, R. L. Nord and D. Soni. Describing software architecture with UML. *In Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, February 1999.

24. P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, pages 42-50, November 1995.

25. D. C. Luckham, L. M. Augustin, J. J. Kenny, J. Veera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering,* 21(4): 336-355, April 1995.

26. J. Magee, N. Dulay, S. Eisenbach and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference*, ESEC'95, September 1995.

27. V. Matena and M. Hapner. Enterprise JavaBeans™. Sun Microsystems Inc., Palo Alto, California, 1998.

28. N. Medvidovic, P. Oreizy, J. E. Robbins and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *SIGSOFT'96: Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering*. ACM Press. Oct 1996.

29. N. Medvidovic and D. S. Rosenblum. Assessing the suitability of a standard design method for modeling software architectures. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, February 1999.

30. N. Medvidovic and R. N. Taylor. Architecture description languages. In *Software Engineering ESEC/FSE'97*, Lecture Notes in Computer Science, Vol. 1301, Zurich, Switzerland, Sept 1997. Springer.

31. E. Mettala and M. H. Graham. The domain-specific software architecture program. Technical Report CMU/SEI-92-SR-9. Carnegie Mellon Univ., Jun 1992.

32. M. Moriconi, X. Qian and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):356-372, April 1995.

33. D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5:128-138, March 1979.

34. D. L. Parnas, P. C. Clements and D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*. SE-11(3):259-266, March 1985.

35. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40-52, October 1992.

36. E. Rechtin. *Systems architecting: Creating and Building Complex Systems*. Prentice Hall, 1991.

37. CMU Software Engineering Institute Product Line Program. http://www.sei.cmu.edu/activities/plp/, 1999.

38. M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of COMPSAC 1997*, August 1997.

39. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnick. Abstractions for software architecture and tools to support them. *IEEE Trans on Software Engineering*. 21(4):314-335. April 1995.

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

41. Mary Shaw. Architectural Requirements for Computing with Coalitions of Resources. 1st Working IFIP Conf. on Software Architecture, Feb 1999 http://www.cs.cmu.edu/~Vit/paper_abstracts/Shaw-Coalitions.html.

42. J. A. Stafford, D. J. Richardson, A. L. Wolf. Aladdin: A Tool for Architecture-Level Dependence Analysis of Software. University of Colorado at Boulder, Technical Report CU-CS-858-98, April, 1998.

43. C. Szyperski. Component Software: *Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

44. D. Garlan, A. Kompanek, P. Pinto: Reconciling the Needs of Architectural Description with Object-Modeling Notations. *Proceedings of UML 2000,* October 2000, York UK.

45. J. Zhao, Using Dependence Analysis to Support Software Architecture Understanding. In M. Li (Ed.), "New Technologies on Computer Software," pp.135-142, International Academic Publishers, September 1997.