Software Architecture

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

1. INTRODUCTION

As the size and complexity of software systems increase, the design, specification, and analysis of overall system structure becomes a critical issue. Structural issues include the organization of a system as a composition of components; global control structures, the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance, and dimensions of evolution. This is the *software architecture* level of design.

Over the past decade architectural design has emerged as an important subfield of software engineering. Practitioners have come to realize that having a good architectural design is a critical success factor for complex system development. A good architecture can help ensure that a system will satisfy key requirements in such areas as performance, reliability, portability, scalability, and interoperability. A bad architecture can be disastrous.

Practitioners have also begun to recognize the value of making explicit architectural choices, and leveraging past architectural designs in the development of new products. Today there are 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.

2. THE ROLES OF SOFTWARE ARCHITECTURE

What exactly is meant by the term "software architecture?" If we look at the common uses of the term "architecture" in software, we find that it is used in different ways, often making it difficult to understand what aspect is being addressed. Among the uses are: (a) the architecture of a particular system, as in "the architecture of system S contains components $C_1 \ldots C_n$," (b) an architectural style, as in "system S adopts a client-server architecture," and (c) the general study of architecture, as in "there are many books on software architecture."

Within software engineering, however, most uses of the term focus on the first of these interpretations. A typical definition is:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them [5].

While there are numerous similar definitions of software architecture, at the core of all of them is the notion that the architecture of a system describes its gross structure using one or more views. The structure in a view illuminates a set of top-level design decisions, including 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 ideally includes sufficient information to allow high-level analysis and critical appraisal.

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

To appear in *Encyclopedia of Software Engineering*,
John Wiley & Sons, Inc. 2001

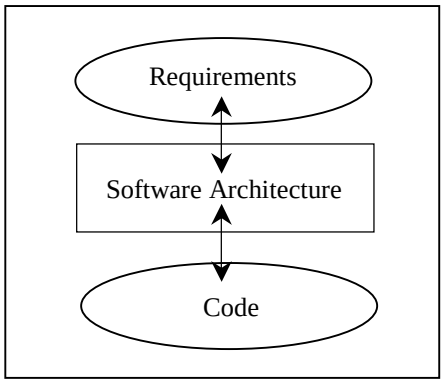

Figure 1: Software Architecture as a Bridge

By providing an abstract description (or model) 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 can play an important role in at least six aspects of software development.

1. *Understanding:* Software architecture simplifies our ability to comprehend large systems by presenting them at a level of abstraction at which a system's design can be easily understood [2, 20, 35]. Moreover, at its best, architectural description exposes the high-level constraints on system design, as well as the rationale for specific architectural choices.

2. *Reuse:* Architectural design can support reuse in several ways. Current work on reuse generally focuses on component libraries. Architectural design supports, in addition, both reuse of large components (or subsystems) and also frameworks into which components can be integrated. Such reusable frameworks may be domain-specific software architectural styles [4, 27], component integration standards [43], and architectural design patterns [8].

3. *Construction:* An architectural description provides a partial blueprint for development by indicating the major software components and dependencies between them. For example, a layered view of an architecture typically documents abstraction boundaries between parts of a system's implementation, clearly identifying the major internal system interfaces, and constraining what parts of a system may rely on services provided by other parts [2].

4. *Evolution:* Software architecture can expose the dimensions along which a system is expected to evolve. By making explicit the "load-bearing walls" of a system, system maintainers can better understand the ramifications of changes, and thereby more accurately estimate costs of modifications. Moreover, architectural descriptions separate concerns about the functionality of a component from the ways in which that component is connected to (interacts with) other components, by clearly distinguishing between components and mechanisms that allow them to interact. This separation permits one to more easily change connection mechanisms to handle evolving concerns about performance and reuse.

5. *Analysis:* Architectural descriptions provide new opportunities for analysis, including system consistency checking [3, 25], conformance to constraints imposed by an architectural style [1], conformance to quality attributes [9], dependence analysis [42], and domain-specific analyses for architectures built in specific styles [10, 15, 26].

6. *Management:* Experience has shown that successful projects view achievement of a viable software architecture as a key milestone in an industrial software development process. Critical evaluation of an architecture typically leads to a much clearer understanding of requirements, implementation strategies, and potential risks [7].

7. *Communication:* An architectural description often serves as a vehicle for communication among stakeholders. For example, explicit architectural design reviews allow stakeholders to voice opinions about relative weights of features and quality attributes when architectural tradeoffs must be considered [9].

3. PRECURSORS

The notion of providing explicit descriptions of system structures goes way back. In the 1960's and 1970's there were active debates about criteria on which to base modularization of software [12, 45]. Programming languages began to provide new features for modularization and the specification of interfaces.

In 1975 DeRemer and Kron [11] argued that creating program modules and connecting them to form larger structures were distinct design efforts. They created the first module interconnection language (MIL) to support that connection effort. In an MIL, modules import and export resources, which are named programming-language elements such as type definitions, constants, variable, and functions. A compiler for an MIL ensures system integrity using inter-module type checking. Since DeRemer and Kron's proposal, other MILs have been developed for specific programming languages such as Ada and Standard ML, and have provided a base from which to support software construction, version control, and system families [33,46]. Enough examples are available to develop models of the design space [47].

These early efforts to develop good ways to talk about system structures and to provide criteria for software modularization focused primarily on the problem of code organization, and relationships between the parts based on interactions such as procedure call and simple data sharing. The key question was how to partition the software into units that could be implemented separately by software developers, and that would provide downstream benefits in support of extensibility, maintenance, and system understandability.

Today's view of software architecture builds on the insights and concepts from the early days of software structuring, but goes much further by also considering architectural representations that capture a system's run-time structures and behavior. By representing architectures as interacting components (viewed as actual run-time entities), these representations more directly facilitate reasoning about system properties such as performance, security, and reliability. Additionally, modern views of software architecture provide a much richer notion of interaction (than procedure call and simple data sharing), permitting new abstractions for the "glue" that allows components to be composed.

4. A NEW DISCIPLINE EMERGES

Initially architectural design was largely an ad hoc affair. Architectural definitions relied on informal box-and-line diagrams, which were rarely maintained once a system was constructed. Architectural choices were made in an 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 virtually no way to check that a given system implementation faithfully represented its architectural design.

However, despite their informality, architectural descriptions were central to 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. The 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 the concern with exploiting commonalities in specific domains to provide reusable frameworks for product families. 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).

Much has changed in the past decade. Although there is wide variation in the state of the practice, broadly speaking, 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.

5. ARCHITECTURE DESCRIPTION LANGUAGES AND TOOLS

The informality of most box-and-line depictions of architectural designs leads to a number of problems. The meaning of the design may not be clear. Informal diagrams cannot be formally 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.

To alleviate these problems there have been number of important developments. First has been the emergence of practitioner guidelines [2] and published standards for architectural documentation [44, 48]. These have helped to codify best practices and provide some uniformity to the way architectures are documented.

A second development has been the creation of formal notations for representing and analyzing architectural designs. Sometimes referred to as "Architecture Description Languages" or “Architecture Definition Languages” (ADLs), these notations usually provide both a conceptual framework and a concrete syntax for characterizing software architectures [19, 30]. They also typically provide tools for parsing, displaying, compiling, analyzing, or simulating architectural descriptions.

Examples of ADLs include Acme[18], 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: Acme supports interchange of architectural descriptions, 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.

Although these languages (and their tools) differ in many respects, a number of key insights have emerged through their development.

The first insight is that good architectural description benefits from multiple views, each view capturing some aspect of the system [2, 24, 44, 48]. Two of the more important classes of view are:

- **Code-oriented views**, which describe how the software is organized into modules, and what kinds if implementation dependencies exist between those modules. Class diagrams, layered diagrams, and work breakdown structures are examples of this class of view; and
- **Execution-oriented views**, which describe how the system appears at run time, typically providing one or more snapshots of a system in action. These views are useful for documenting and analyzing execution properties such as performance, reliability, and security.

A second insight is that architectural description of execution-oriented views, as embodied in most of the ADLs mentioned earlier, requires the ability to model the following as first class design entities:

- **Components** represent the computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of software architectures. Examples of components include clients, servers, filters, blackboards, and databases. Components may have multiple interfaces, each interface defining a point of interaction between a component and its environment. A component may have several interfaces of the same type (e.g., a server may have several active http connections).
- **Connectors** represent interactions among components. They provide the “glue” for architectural designs, and correspond to the lines in box-and-line descriptions. From a run-time perspective, connectors mediate the communication and coordination activities among components. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. Connectors may also represent complex interactions, such as a client-server protocol or a SQL link between a database and an application. Connectors have interfaces that define the roles played by the participants in the interaction.
- **Systems** represent graphs of components and connectors. In general, systems may be hierarchical: components and connectors may represent subsystems that have their own internal architectures. We will refer to these as *representations.* When a sys-

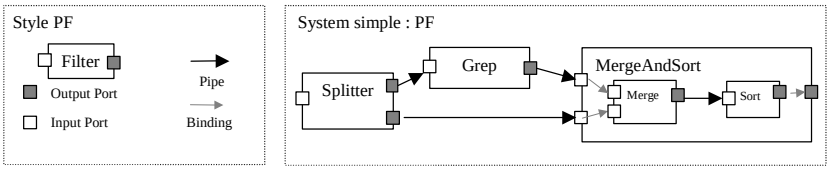

Figure 2. A system in the pipe-filter style

tem or part of a system has a representation, it is also necessary to explain the mapping between the internal and external interfaces.

- **Properties** represent additional information (beyond structure) about the parts of an architectural description. Although the properties that can be expressed by different ADLs vary considerably, typically they are used to represent anticipated or required extra-functional aspects of an architectural design. For example, some ADLs allow one to calculate system throughput and latency based on performance estimates of the constituent components and connectors. In general, it is desirable to be able to associate properties with any architectural element in a description (components, connectors, systems, and their interfaces). For example, a property of an interface might describe an interaction protocol.
- **Styles** represent families of related systems. An architectural style typically defines a vocabulary of design element types as a set of component, connector, port, role, binding, and property types, together with rules for composing instances of the types. We will describe some of the more prominent styles later in this article.

To illustrate the use of these modeling constructs, consider the example shown in Figure 2. The system defines an execution-oriented view of a simple string-processing application that extracts and sorts text. The system is described in a pipe-filter style, which provides a design vocabulary consisting of a filter component type and pipe connector type, input and output interface (port) types, and a single binding type. In addition, there would likely be constraints (not shown) that ensure, for example, that the reader/writer roles of the pipe are associated with appropriate input/output ports. The system is described hierarchically: *MergeAndSort* is defined by a representation that is itself a pipe-filter system. In complementary documentation, properties of the components and connectors might list, for example, performance characteristics used by a tool to calculate overall system throughput.

6. PRODUCT LINES AND ARCHITECTURAL STANDARDS

As noted earlier, an important trend 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 domain-specific architectural standards for cross-vendor integration.

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.

However, in a product line approach, 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.

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 [22, 13].) Moreover, organizations such as the CMU Software Engineering Institute are well on their way towards providing concrete guidelines and processes for the use of a product line approach [37].

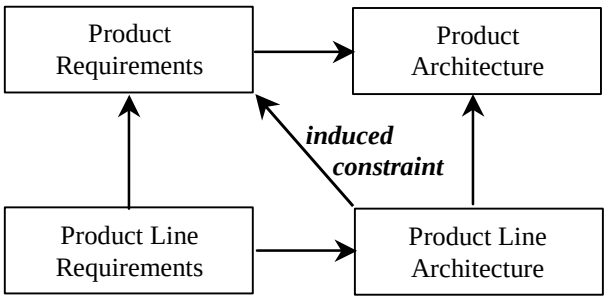

Figure 3: Product Line Architectures

Like product line approaches, domain-specific architectural standards for cross-vendor integration provide frameworks that permit system developers to configure a wide variety of specific systems by instantiating that framework. But more importantly, such standards support the integration of parts provided by multiple vendors. A number of these have been sanctioned as formal international standards (such as those sponsored by IEEE or ISO), while others are ad hoc or de facto standards promoted by one or more industrial leaders.

A good example of the former is the High Level Architecture (HLA) for Distributed Simulation [4]. Initially proposed by the US Defense Modeling and Simulation Office as a standard to permit the integration of simulations produced by many vendors, it now has become an IEEE Standard (IEEE P1516.1/D6). The HLA 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.

7. CODIFICATION AND DISSEMINATION

One early impediment to the emergence of architectural design as an engineering discipline 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, 44] and courses [21].

A common theme in these books and courses is the use of standard architectural *styles*. An architectural style typically specifies a design vocabulary, constraints on how that vocabulary is used, and semantic assumptions about that vocabulary [1]. For example, a pipe-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-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 previously-defined 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 a 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, some of which are illustrated in Figure 4 (due to

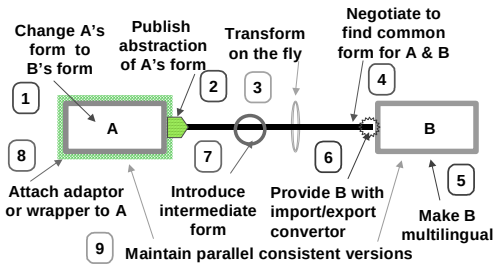

Figure 4: Some mismatch repair techniques

Mary Shaw).

8. RELATED AREAS

There are a number of closely related areas.

9.1 Software Development Methods

One of the hallmarks of software engineering progress has been the development of methods and processes for software development. Like software architecture, methods attempt to provide a path from requirements to code that eliminates some of the ad hoc development practice of the past.

Methods complement software architecture: the former attempt to provide a set of regular steps for software development, while the latter attempts to provide a basis for developing and analyzing certain design models along that path.

To the extent that they support conceptual design of systems, they also address architectural concerns. On the other hand, most methods tend to favor a particular architectural style. For example, object-oriented methods naturally favor architectural designs based on interacting objects, while other methods favor other styles.

9.2 Object-Oriented Design and Modeling

There are a number of 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 [31]. 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, DCOM, and 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 [14].

Given these similarities it is worth asking the question: what are the important differences between the two fields? To shed light on the issue, it is helps to view the relationship between architecture and object-oriented methods from at least three distinct perspectives.

1. *Object-oriented design 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. *Object-oriented design 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. *Object-oriented design as an architectural modeling notation:* This perspective treats a notation such as UML as a suitable notation for all architectural descriptions [8, 24]. Proponents of this perspective have advocated various ways of using object modeling, including class diagrams, collaboration diagrams, and package diagrams [17, 24, 29]. From this perspective, architecture is viewed as a sub-activity of object-oriented design.

Elaborating on the relationship between ADLs and object-oriented modeling notations, such as UML, Figure 5 shows some of the paths that might be followed. Path A-D is one in which an ADL is used as the modeling language. Path B-E is one in which UML is used as the modeling notation. Path A-C-E, is one in which an architecture is first represented in an ADL, but then transformed into UML before producing an implementation.

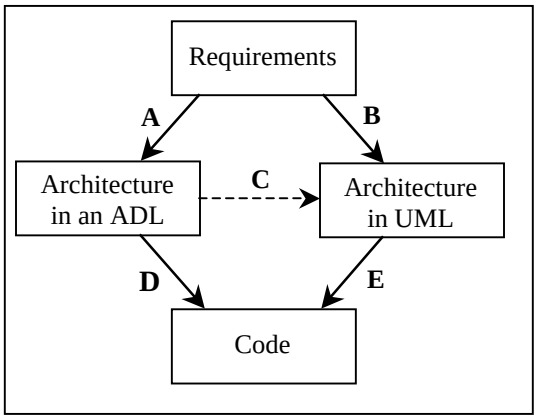

Figure 5: ADLs versus Object Modeling

Using a more general modeling language such as UML has the advantages of providing a notation that practitioners are more likely to be familiar with, and providing a more direct link to object-oriented implementations and development tools. But general-purpose object languages suffer from the problem that the object conceptual vocabulary may not be ideally suited for representing architectural concepts, and there are likely to be fewer opportunities for automated analysis of architectural properties.

9.3 Component-Based Systems

Component-based systems are closely related to object-oriented systems insofar as both are based on the construction of systems from encapsulated entities that provide well-defined interfaces to a set of services. However, most component-based systems have a strong intrinsic architectural flavor in that they are usually coupled with an integration framework that prescribes what kinds of interfaces the components must have, and ways in which components can interact at run-time [43].

From an architectural perspective component-based systems such as DCOM, CORBA, and JavaBeans define architectural styles that are predominantly object-oriented. In addition, they may support other forms of interaction such as event publish-subscribe. However, component integration standards typically go beyond architectural modeling by providing run-time infrastructure and (in many cases) considerable support for generating code from more abstract descriptions.

9. FUTURE PROSPECTS

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. As architectural design matures into an engineering discipline that is universally recognized and practiced, there are a number of significant challenges that will need to be addressed. Many of the solutions to these challenges are likely to arise as a natural consequence of dissemination and maturation of the architectural practices and technology that we know about today. Other challenges arise because of the shifting landscape of computing and the needs for software: these will require significant new innovations. This article has attempted to provide a high-level overview of the terrain – illustrating where software architecture has come over the past few years, and outlining relationships between software architecture and other aspects of software engineering.

10. ACKNOWLEDGEMENTS

The author would like to acknowledge a number of colleagues and students who have helped clarify his ideas on software architecture, including Barry Boehm, Dewayne Perry, John Salasin, Mary Shaw, Dave Wile, Alex Wolf, and past and present members of the ABLE research group at Carnegie Mellon University.

11. 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. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Software Architecture Documentation in Practice*, Addison Wesley Longman, 2001.

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

4. F. Kuhl, R. Weatherly, J. Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture.* Prentice Hall, 2000.

5. L. Bass, P. Clements and R. Kazman. *Software Architecture in Practice.* Addison Wesley, 1999, 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. Frank DeRemer and Hans H. Kron. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Trans. on Software Engineering*, SE-2(2):80-86, June 1976.

12. E. W. Dijkstra. The structure of the "THE" – multi-programming 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 and A. J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Proceedings of the Third International Conference on the Unified Modeling Language. 2000*.

18. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, Gary T. Leavens and Murali Sitaraman (eds), Cambridge University Press, 2000 pp. 47-68.

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. J. Rumbaugh, I Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.

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. P. Clements and L. Northrop. "Software Product Lines: Practices and Patterns." Addison Wesley Longman, 2001..

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. International Organization for Standardization. *ISO/IEC 10746 1-4 Open Distributed Processing – Reference Model (Parts 1-4)*, July 1995. ITU Recommendation X.901-904.

45. D. Parnas. On the Criteria To Be Used in Decomposing Systems Into Modules. *Communications of the ACM*, 15(12):1053-1058, December 1972.

46. L. W. Cooprider. The representation of software families. Ph.D. Thesis, Technical Report CMU-CS-79-116. Carnegie Mellon University, 1979.

47. D. E. Perry. Software interconnection models. In *Proceedings of the 9th International Conference on Software Engineering*. IEEE Computer Society Press, March 1987.

48. *IEEE Std 1471-2000. Recommended Practice for Architectural Description of Software-Intensive Systems*, October 2000.