Research Directions in Software Architecture[1]

DAVID GARLAN

Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213, ⟨garlan@cs.cmu.edu⟩.

What is Software Architecture?

A critical aspect of the design for any large software system is its high-level organization of computational elements and interactions between those elements. Broadly speaking, this is the software architectural level of design [Garlan and Shaw 1993; Perry and Wolf 1992]. The structure of software has long been recognized as an important issue (e.g., [Dijkstra 1968; Parnas et al. 1985]) and recently software architecture has begun to emerge as an explicit field of study for software engineering practitioners and researchers. There is a large body of recent work in areas such as module interface languages, domain-specific architectures, architectural description languages, design patterns and handbooks, formal underpinnings for architectural design, and architectural design environments [Garlan 1995; Garlan and Perry 1995].

Although there is increasing agreement about the issues addressed by architectural design, there is currently no single, widely accepted definition of "software architecture." Indeed, the term is used in quite different ways, including: (a) the architecture of a particular system, as in "the architecture of this system consists of the following components," (b) an architectural style, as in "this system adopts a client-server architecture," and (c) the general study of architecture, as in "the papers in this journal are about architecture."

Within software engineering, most uses of the term "software architecture" focus on the first of these interpretations. The following definition (developed in a software architecture discussion group at the SEI in 1994) is typical:

> *The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.*

But definitions such as this tell only a small part of the story. More important is the current locus of effort in research and development centered on software architecture.

This effort has been prompted by two distinct trends. The first is the recognition that over the years designers have begun to develop 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 are rarely assigned precise definitions, they permit designers to describe complex systems using abstractions that make the overall system intelligible.

The second trend is the concern with exploiting specific domains to provide reusable frameworks for product fami-

---

[1] This article was adapted from the Editors' Introduction to the *IEEE Transactions on Software Engineering* Special Issue on Software Architecture, 1995, by David Garlan and Dewayne Perry.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1995 ACM 0360-0300/95/0600-0257 $03.50

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).

Generalizing from these trends, it is possible to identify three salient distinctions between software architecture and other areas of computer science.

- *Focus of concern*: The first is between traditional concerns about design of algorithms and data structures, on the one hand, and architectural concerns about the organization of a large system, on the other. In particular, software architectural design is concerned with gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.
- *Nature of representation*: The second distinction is between system description based on definition-use structure and architectural description based on graphs of interacting components [Allen and Garlan 1994]. The former modularizes a system in terms of source code, usually making explicit the dependencies between use sites of the code and corresponding definition sites. The latter modularizes a system as a graph, or configuration, of "components" and "connectors." Components define the application-level computations and data stores of a system. Examples include clients, servers, filters, databases, and objects. Connectors define the interactions between those components. The interactions can be as simple as procedure calls, pipes, and event broadcast, or be much more complex, including client-server protocols, database accessing protocols, and so on.
- *Design methods versus architectures*: The third distinction is between software design methods—such as object-oriented design and structured analysis—and software architecture. Although both design methods and architectures are concerned with bridging the gap between requirements and implementations, there is a significant difference. Without either software design methods or a discipline of software architecture design, the implementor is typically left to develop a solution using whatever *ad hoc* techniques may be at hand (Figure 1a). Design methods improve the situation by providing a path between some class of system requirements and some class of system implementations (Figure 1b). Ideally, a design method defines the steps that take a system designer from the requirements to a solution. The extent to which such methods are successful often depends on their ability to exploit constraints on the class of problems they address and the class of solutions they provide. One way is to focus on certain styles of architectural design. For example, object-oriented methods usually lead to systems formed out of objects, while others may lead more naturally to systems with an emphasis on dataflow. In contrast, the field of software architecture is concerned with the space of architectural designs (Figure 1c). Within this space object-oriented and dataflow structures are but two of the many possibilities. Architecture is concerned with the tradeoffs between the choices in this space—the properties of different architec-

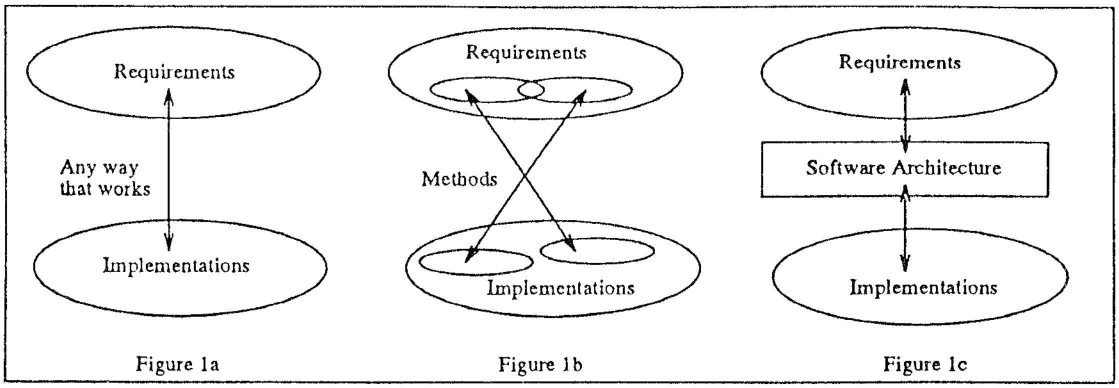

Figure 1. Design methods versus software architecture.

tural designs and their ability to solve certain kinds of problems. Thus design methods and architectures complement each other: behind most design methods are preferred architectural styles, and different architectural styles can lead to new design methods that exploit them.

Why is Software Architecture Important?

Principled use of software architecture can have a positive impact on at least five aspects of software development.

Understanding. Software architecture simplifies our ability to comprehend large systems by presenting them at a level of abstraction at which high-level design can be understood [Garlan and Shaw 1993; Perry and Wolf 1992]. Moreover, at its best, architectural description exposes the high-level constraints on system design, as well as the rationale for making specific architectural choices.

Reuse. Architectural descriptions support reuse at multiple levels. Current work on reuse generally focuses on component libraries. Architectural design supports, in addition, both reuse of large components and also frameworks into which components can be integrated. Existing work on domain-specific software architectures, reference frameworks, and design patterns provide evidence for this [Gamma et al. 1994; Mettala and Graham 1992].

Evolution. Software architecture can expose the dimensions along which a system is expected to evolve. By making explicit the "load-bearing walls," system maintainers can better understand the ramifications of change, and thereby more accurately estimate the costs of modification [Perry and Wolf 1992]. Moreover, architectural descriptions can separate functionality from the ways in which a component is connected to (interacts with) other components. This allows one to change the connection mechanism to handle evolving concerns about performance, interoperability, prototyping, and reuse.

Analysis. Architectural descriptions provide new opportunities for analysis, including high-level forms of system consistency checking [Allen and Garlan 1994], conformance to an architectural style [Abowd et al. 1993], conformance to quality attributes [Clements et al. 1995], and domain-specific analyses for architectures that conform to specific styles [Garlan et al. 1994].

Management. There is a strong rationale for making viable software architecture a key milestone in industrial software development. Achieving this involves specifying a software system's initial operational capability requirements, the dimensions of anticipated growth, the software architecture, and a rationale that demonstrates that the architecture will satisfy the system's initial requirements and anticipated growth. If one proceeds to develop a software product without satisfying these conditions, there is significant risk that the system will be either inadequate or unable to accommodate change.

Research Directions

Although application of good architectural design is becoming increasingly important to software engineering, much common practice leads to architectural designs that are informal, ad hoc, unanalyzable, unmaintainable, and handcrafted. Consequently architectural design is only vaguely understood by developers; architectural choices are based more on default than on solid engineering principles; architectural designs cannot be analyzed for consistency or completeness; architectures are not enforced as a system evolves; and there are virtually no tools to help architectural designers.

Current research in software architecture attempts to address all of these issues; among the more active areas are

- *Architecture description languages* address the need for expressive notation in architectural design and architectural styles. In particular, much of this research is on providing precise descriptions of the "glue" for combining components into larger systems.
- *Formal underpinnings of software architecture* addresses the current imprecision of architectural description by providing formal models of architectures, mathematical foundations for modularization and system composition, formal characterizations of extra-functional properties (such as performance and maintainability), and theories of architectural connection.
- *Architectural analysis techniques* are being developed for determining and predicting properties of architectures. In particular, progress is being made in understanding the relationships between architectural constraints and the ability to perform specialized analyses, as well as abstraction techniques that make analysis practical for large systems.
- *Architectural development methods* become imperative to integrate architectural activities smoothly into the broader methods and processes of software development.
- *Architecture recovery and reengineering* to handle legacy code is critical for large systems with long lifetimes. Research is beginning to address extraction of architectural design from existing systems, unification of related architectural designs, and abstraction, generalization, and instantiation of domain-specific components and frameworks. There is also increasing research on issues of interoperability: techniques for detecting component mismatch and bridging them.
- *Architectural codification and guidance* to codify design expertise so that nonexperts can use it. This has led to an interest in rules and techniques for selecting architectural styles, handbooks of patterns and elements, and curricula for educating software architects.
- *Tools and environments for architectural design* to support architectural design with new tools and environments. Current work addresses architectural analysis tools, architectural design environments, and application generators.
- *Case studies* of architectural design, including retrospective analyses of successful (and sometimes unsuccessful) architectural development, to increase our understanding of architectural design, as well as to provide model problems against which other researchers can gauge their effectiveness.

ACKNOWLEDGMENTS

We thank Barry Boehm, Paul Clements, Robert Monroe, Mary Shaw, and Jeannette Wing for their insightful comments on earlier drafts of this work

REFERENCES

ABOWD, G., ALLEN, R., AND GARLAN, D. 1993. Using style to give meaning to software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering, Softw. Eng. Not. 18*, 5 (Dec.), 9–20.

ALLEN, R., AND GARLAN, D. 1994. Formalizing architectural connection. In *Proceedings of ICSE'16* (May).

ALLEN, R., AND GARLAN, D. 1994. Beyond definition/use. Architectural interconnection. In *Proceedings of the ACM Interface Definition Language Workshop. SIGPLAN Not. 29*, 8 (Aug.).

CLEMENTS, P., BASS, L., KAZMAN, R., AND ABOWD, G. 1995. Predicting software quality by architecture-level evaluation. In *Proceedings of the Fifth International Conference on Software Quality* (Austin, Tex., Oct.).

DIJKSTRA, E. W. 1968. The structure of the "THE"-multiprogramming system. *Commun. ACM 11*, 5, 341–346.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Design.* Addison-Wesley, Reading, Mass.

GARLAN, D. (ED.). 1995. *Proceedings of the First International Workshop on Software Architecture* (April 1995). CMU Tech. Rep. CMU-CS-95-151, May.

GARLAN, D., ALLEN, R., AND OCKERBLOOM, J. 1994. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, ACM Press (Dec.).

GARLAN, D., AND SHAW, M. 1993. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering, Vol. I*, World Scientific Publishing.

GARLAN, D., PERRY, D. (EDS.) 1995. Special Issue on Software Architecture. *IEEE Trans. Softw. Eng.*

METTALA, E., AND GRAHAM, M. H. 1992. The domain-specific software architecture program. Tech. Rep. CMU/SEI-92-SR-9, CMU Software Engineering Institute, June.

PARNAS, D. L., CLEMENTS, P. C., AND WEISS, D. M. 1985. The modular structure of complex systems, *IEEE Trans. Softw. Eng. SE-11*, (March) 259–266.

PERRY, D. E., AND WOLF, A. L. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Soft. Eng. Notes 17.*