Software Architecture: Practice, Potential, and Pitfalls

Panel Introduction

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA

Dewayne Perry
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974 USA

1 What is software architecture?

A critical aspect of the design for any large software system is its gross structure – that is, its high-level organization of computational elements and interactions between those elements [3, 6]. Broadly speaking, we refer to this as the software architectural level of design. Recently software architecture has begun to emerge as an important field of study for software engineering practitioners and researchers. This emergence is evidenced by a large body of recent work in areas such as module interface languages, domain specific architectures, architectural description languages, formal underpinnings for architectural design, and architectural design environments.

What do we mean by the term "software architecture"? If we look at the current uses of the term "architecture", we find that it is used in different ways, often making it difficult to understand what aspect is being addressed. Among the various meanings are (a) the architecture of a particular artifact, as in "the blueprints describe this building," (b) an architectural style, as in "that church is an example of Gothic architecture," and (c) the general study of architecture, as in "he has an advanced degree in architecture."

To clarify the meaning of the term "architecture" with respect to software systems, it is helpful to observe that the recent emergence of interest in software architecture 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 patterns as a "pipeline", a "blackboard-oriented design", or a "client-server system". Although these terms rarely have precise definitions, they permit designers to describe complex systems using abstractions that make the overall system intelligible. Moreover, they provide significant semantic content that inform others about the kinds of properties that the system will have: the expected paths of evolution, its overall computational paradigm, and its relationship to similar systems.

The second trend is the recent interest in exploiting specific domains to provide reusable frameworks for product families. This 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 by "instantiating" the shared infrastructure. 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 a different layers of abstraction), fourth generation languages (which exploit the common patterns of business information processing), and user interface toolkits and frameworks.

Generalizing from these trends, it is possible to identify three salient distinctions. The first distinction is between traditional concerns about design of algorithms and data structures, on the one hand, and architectural concerns about the gross modularization of a large system, on the other. The former has been the traditional focus of much of computer science, while the latter is emerging as a significant and different design level that requires its own notations, theories, and tools.

The second distinction is between system description based on definition-use structure and architectural description based on graphs of interacting components. The former modularizes a system in terms of source code, usually making explicit 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 primary points of computation in the system, while connectors define the interactions between those components. These interactions can be as simple as procedure calls or data sharing, or can be as complex as pipes, event broadcast, client-server protocols, database accessing protocols, etc.

The third distinction is between architectural instance and architectural style. An architectural instance refers to the architecture of a specific system. Box and line diagrams that accompany system documentation describe architectural instances since they apply to individual systems. An architectural style, however, defines constraints on the form and structure of a family of architectural instances. For example, a "pipe and filter" architectural style might define the family of system architectures that are constructed as a graph of incremental stream transformers. Architectural styles typically prescribe such things as a vocabulary of components and connectors (*e.g.,* filters and pipes), topological constraints (*e.g.,* the graph must be acyclic), semantic constraints (*e.g.,* filters cannot share state), and specific instances of components or connectors (*e.g.,* there must be a database in the system). Different stylistic categories range from abstract architectural patterns and idioms (such as "client-server" organization) to concrete "reference architectures" (such as the ISO OSI communication model and the traditional linear decomposition of a compiler).

2 Significance to software engineering

Architectural design of large systems has always played a significant role in determining the ultimate success of a system: choosing an inappropriate architecture can have a disastrous effect. However, traditionally architectural design has been largely informal and ad hoc, with the result that it has been difficult to communicate, analyze, and compare architectural designs and principles. We believe that the current interest in software architecture signals the emergence of a more disciplined basis for architectural design that has the potential to significantly improve our ability to construct effective software systems.

Specifically, a principled use of software architecture can have a positive impact on at least four aspects of software development.

1. **Understanding:** Software architecture simplifies our ability to comprehend large systems by presenting them at a level of abstraction in which the whole system can be understood [3, 6]. Moreover, at its best, architectural description exposes the high level constraints on system design as well as the rationale for making specific architectural choices.

2. **Reuse:** Architectural description supports reuse at multiple levels. While most current work on reuse focuses on component libraries, architectural design supports, in addition, both reuse of large components (such as subsystems), and also the complementary need for reusable frameworks into which components can be integrated. Existing work on domain-specific software architectures and reference frameworks have already begun to provide evidence for this [5].

3. **Evolution:** Software architecture can expose the dimensions along which a system is expected to evolve. By making explicit what are the "load-bearing walls" of a system, system maintainers can better understand the ramifications of changes, and thereby more accurately estimate costs of modifications [6].

4. **Analysis:** Architectural description provides new opportunities for analysis [6], including high-level forms of system consistency checking [2, 4], conformance to an architectural style [1], and domain-specific analyses for architectures that conform to specific styles.

3 Purpose of the panel

Whatever the long-term impact of software architecture may turn out to be, an appropriate starting point is a concrete appraisal of the current state of the practice in the use of software architecture. It is the purpose of this panel to take a step in this direction. By assembling a panel of experts with a broad base of experience in the area, we hope to provide concrete examples of what is now possible when architectural principles are applied to industrial problems in systematic ways, to consider the potential impact of software architecture over the next few years, and to suggest steps that should be taken to bring this about.

References

[1] Gregory Abowd et al. Using style to give meaning to software architecture. In *Proc. of SIGSOFT'93: Foundations of Software Engineering*, December 1993.

[2] Robert Allen and David Garlan. Formalizing architectural connection. In *Proc. of ICSE'16*, May 1994.

[3] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering, Volume I.* World Scientific Publishing Company, 1993.

[4] David C. Luckham et al. Partial orderings of event sets and their application to prototyping concurrent timed systems, Unpublished draft of March 1992.

[5] Erik Mettala and Marc H. Graham. The domain-specific software architecture program. Technical Report CMU/SEI-92-SR-9, CMU Software Engineering Institute, June 1992.

[6] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), 1992.