

## Defining the Terms Architecture, Design, and Implementation

Rick Kazman and Amnon Eden

### Introduction

Over the past 10 years many practitioners and researchers have sought to define software architecture. At the SEI, we use the following definition:

*The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

Definitions of software architecture abound. We've been collecting definitions from visitors to our Web site (<http://www.sei.cmu.edu/architecture/definitions.html>) and already have received dozens. However, we are interested not only in understanding the term "software architecture" but in clarifying the difference between architecture and other related terms such as "design" and "implementation." The lack of a clear distinction among these terms is the cause of much muddled thinking, imprecise communication, and wasted, overlapping effort. For example, "architecture" is often used as a mere synonym for "design" (sometimes preceded with the adjective "high-level"). And many people use the term "architectural patterns" as a synonym for "design patterns."

Confusion also stems from the use of the same specification language for both architectural and design specifications. For example, UML is often used as an architectural description language. In fact, UML has become the industry de facto standard for describing architectures, although it was specifically designed to manifest detailed design decisions (and this is still its most common use). This merely contributes to the confusion, since a designer using UML has no way (within UML) of distinguishing architectural information from other types of information.

Confusion also exists with respect to the artifacts of design and implementation. UML class diagrams, for instance, are a prototypical artifact of the design phase. Nonetheless, class diagrams may accumulate enough detail to allow code generation of very detailed programs, an approach that is promoted by CASE tools such as Rational Rose and System Architect. Using the same specification language further blurs the distinction between artifacts of the design (class diagrams) and artifacts of the implementation (source code). Having a unified specification language is, in many ways, a good thing. But a user of this unified language is given little help in knowing if a proposed change is "architectural" or not.

Why are we interested in such distinctions? Naturally, a well-defined language improves our understanding of the subject matter. With time, terms that are used interchangeably lose their meaning, resulting inevitably in ambiguous descriptions given by developers, and significant effort is wasted in discussions of the form “by design I mean...and by architecture I mean...”

Seeking to separate architectural design from other design activities, definers of software architecture in the past have stressed the following:

1. “*Architecture* is concerned with the selection of architectural elements, their interaction, and the constraints on those elements and their interactions...*Design* is concerned with the modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satisfy the requirements.”
2. Software architecture is “concerned with issues...beyond the algorithms and data structures of the computation.”
3. “Architecture...is specifically not about...details of implementations (e.g., algorithms and data structures.)...Architectural design involves a richer collection of abstractions than is typically provided by OOD” (object-oriented design).

In suggesting typical “architectures” and “architectural styles,” existing definitions consist of examples and offer anecdotes rather than providing clear and unambiguous notions. In practice, the terms “architecture,” “design,” and “implementation” appear to connote varying degrees of abstraction in the continuum between complete details (“implementation”), few details (“design”), and the highest form of abstraction (“architecture”). But the amount of detail alone is insufficient to characterize the differences, because architecture and design documents often contain detail that is not explicit in the implementation (e.g., design constraints, standards, performance goals). Thus, we would expect a distinction between these terms to be qualitative and not merely quantitative.

The ontology that we provide below can serve as a reference point for these discussions.

## The Intension/Locality Thesis

To elucidate the relationship between architecture, design, and implementation, we distinguish at least two separate interpretations for abstraction in our context:

1. Intensional (vs. extensional) design specifications are “abstract” in the sense that they can be formally characterized by the use of logic variables that range over an unbounded domain. For example, a layered architectural pattern does not restrict the architect to a specific number of layers; it applies equally well to 2 layers or 12 layers.
2. Non-local (vs. local) specifications are “abstract” in the sense that they apply to *all* parts of the system (as opposed to being limited to some part thereof).

Both of these interpretations contribute to the distinction among architecture, design, and implementation, summarized as the “intension/locality thesis”:

1. Architectural specifications are intensional and non-local
2. Design specifications are intensional but local
3. Implementation specifications are both extensional and local

Table 1 summarizes these distinctions.

**Table 1.** The Intension/Locality Thesis

Architecture	<i>Intensional</i>	<i>Non-local</i>
Design	<i>Intensional</i>	<i>Local</i>
Implementation	<i>Extensional</i>	<i>Local</i>

## Implications

What are the implications of such definitions? They give us a firm basis for determining what is architectural (and hence crucial for the achievement of a system's quality attribute requirements) and what is not.

Consider the concept of a strictly layered architecture (an architecture in which each layer is allowed to use only the layer immediately below it). How do we know that the architectural style "layered" is really architectural? To answer that we need to answer whether this style is intentional and whether it is local or non-local. First of all, are there an unbounded number of implementations that qualify as layered? Clearly there are. Secondly, is the layered style local or non-local? To answer that, we need only consider a violation of the style, where a layer depends on a layer above it, or several layers below it. Since this would be a violation wherever it occurred, the notion of a layered architecture must be non-local.

What about a design pattern, such as the factory pattern? This is intentional, because there may be an unbounded number of realizations of a factory design pattern within a system. But is it local or non-local? One may use a design pattern in some corner of the system and not use it (or even violate it) in a different portion of the same system. So design patterns are local.

Similarly, it is simple to show that the term "implementation" refers only to artifacts that are extensional and local.

## Conclusions

Since the inception of architecture as a distinct field of study, there has been much confusion about what the term "architecture" means. Similarly, the distinction between architecture and other forms of design artifacts has never been clear. The intension/locality thesis provides a foundation for determining the meaning of the terms architecture, design, and implementation that accords not only with intuition but also with best industrial practices. A more formal and complete treatment of this topic can be found in our paper, "Architecture, Design, Implementation." But what are the consequences of precisely knowing the differences among these terms? Is this an exercise in definition for definition's sake? We think not. Among others, these distinctions facilitate

1. determining what constitutes a uniform program (e.g., a collection of modules that satisfy the same architectural specifications)
2. determining what information goes into architecture documents and what goes into design documents

3. determining what to examine and what not to examine in an architectural evaluation or a design walkthrough
4. understanding the distinction between local and non-local rules (i.e., between the design rules that are enforced throughout a project versus those that are of a more limited domain, because the architectural rules define the fabric of the system and how it will meet its quality attribute requirements, and the violation of architectural rules typically has more far-reaching consequences than the violation of a local rule)

Furthermore, in the industrial practice of software architecture, many statements that are said to be “architectural” are in fact local (e.g., *both tasks A and B execute on the same node*, or *task A controls B*). Instead, a truly architectural statement would be, for instance, *for each pair of tasks A,B that satisfy some property X, A and B will execute on the same node and the property Control(A,B) holds*.

More generally, for each specification we should be able to determine whether it is a *design* statement, describing a purely local phenomenon (and hence of secondary interest in architectural documentation, discussion, or analysis), or whether it is an instance of an underlying, more general rule. This is a powerful piece of information.

## References

- [Bass 98] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*. Reading, MA: Addison Wesley Longman, Inc., 1998.
- [Booch 99] Booch, G.; Jacobson, I.; & Rumbaugh, J. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.
- [Eden 03] Eden, A. & Kazman, R. “Architecture, Design, Implementation.” *Proceedings of the 25th International Conference on Software Engineering (ICSE 25)*, Portland, OR, May 2003.
- [Garlan 93] Garlan, D. & Shaw, M. “An Introduction to Software Architecture,” 1–39. *Advances in Software Engineering and Knowledge Engineering*, Vol. 2. Edited by V. Ambriola and G. Tortora. New Jersey: World Scientific Publishing Company, 1993.
- [Kazman 99] Kazman, R. “A New Approach to Designing and Analyzing Object-Oriented Software Architecture.” Invited talk, *Conference*

*On Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Nov. 1–5, 1999, Denver, CO.

- [Monroe 97] Monroe, R. T.; Kompanek, A.; Melton, R.; & Garlan, D. “Architectural Styles, Design Patterns, and Objects.” *IEEE Software* 14, 1 (January 1997): 43–52.
- [Perry 92] Perry, D. E. & Wolf, A. L. “Foundation for the Study of Software Architecture.” *ACM SIGSOFT Software Engineering Notes* 17, 4 (1992): 40–52.
- [Popkin Software 00] Popkin Software. *System Architect 2001*. New York, NY: McGraw-Hill, 2000.
- [Quatrani 99] Quatrani, T. *Visual Modelling with Rational Rose 2000 and UML, Revised*. Reading, MA: Addison Wesley Longman, Inc., 1999.
- [Schmidt 00] Schmidt, D. C.; Stal, M.; Rohnert, H.; & Buschmann, F. *Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*. New York, NY: John Wiley & Sons, Ltd., 2000.

## About the Authors

**Rick Kazman** is a senior member of the technical staff at the SEI, where he is a technical lead in the Architecture Tradeoff Analysis Initiative. He is also an adjunct professor at the Universities of Waterloo and Toronto. His primary research interests within software engineering are software architecture, design tools, and software visualization. He is the author of more than 50 papers and co-author of several books, including a book recently published by Addison-Wesley titled *Software Architecture in Practice*. Kazman received a BA and MMath from the University of Waterloo, an MA from York University, and a PhD from Carnegie Mellon University.

**Dr. Eden** is a faculty member in the Department of Computer Science at the University of Essex and a research scholar at the Center for Inquiry. His research focuses on formalizing the informal narrative describing software design and architecture. He received a Ph.D. in 2000 from Tel Aviv University, and has substantial industrial experience in object-oriented analysis and design and C++ programming. He has held positions at the Tel Aviv College of Management, Technion—Israel Institute of Technology (Israel), Uppsala University (Sweden), and Concordia University (Canada).

---

The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University. This article is intended to stimulate further discussion about this topic.

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.

® Capability Maturity Model, Capability Maturity Modeling, Carnegie Mellon, CERT, CERT Coordination Center, CMM, and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

SM Architecture Tradeoff Analysis Method; ATAM; CMM Integration; COTS Usage Risk Evaluation; CURE; EPIC; Evolutionary Process for Integrating COTS Based Systems; Framework for Software Product Line Practice; IDEAL; Interim Profile; OAR; OCTAVE; Operationally Critical Threat, Asset, and Vulnerability Evaluation; Options Analysis for Reengineering; Personal Software Process; PLTP; Product Line Technical Probe; PSP; SCAMPI; SCAMPI Lead Assessor; SCAMPI Lead Appraiser; SCE; SEI; SEI-Europe; SEPG; Team Software Process; and TSP are service marks of Carnegie Mellon University.

TM Simplex is a trademark of Carnegie Mellon University.