Future Trends of Software Technology and Applications: 
Software Architecture

Paul C. Clements
Software Engineering Institute, Carnegie Mellon University
clements@sei.cmu.edu

Abstract

This paper argues that software architecture has always been the enabler for the ever-upward spiral in software engineering of more powerful problem-space languages, and will continue to be so.

1. Position statement

Software architecture has enjoyed phenomenally rapid growth as a field of study and a field of practice over the last ten years or so. A recently published retrospective highlighted its path of growth and made a few predictions about its future [1]. Today no one would dream of building a complex software system without paying careful attention to its architecture. An architecture is the primary carrier of a software system’s quality attributes – its modifiability, performance, security, availability, and the like. An architecture serves as the structuring blueprint for the software but also the development project that is producing the software [2]. The right architecture is essential. The wrong one is a recipe for disaster. It is fitting that a panel on software technology and applications should specifically include software architecture as an explicit part of its deliberations.

In order to predict its future, it helps to understand architecture’s role up to this point. When programs became too large and complex to understand (or to engineer) by working only with source code, architecture became an indispensable concept. It helped people think in terms of problem-domain abstractions that the programming languages of the day did not support. Architecture provided large-scale conceptual “chunks” of functionality wired together as a solution, chunks that addressed particular parts of the problem. Generally useful or commonly recurring chunks led to pre-packaging and pre-wiring of those chunks, which in turn led to languages in which those chunks became primitives.

Software engineering itself can be seen as a series of cycles in which the programming languages—that is, the languages in which we describe to the computing systems the solution we want—become more and more sophisticated and more able to support problem-space abstractions. Binary and assembly language gave way to FORTRAN, whose ilk gave way to C++ and Java, which are now yielding to “e-business platforms” in which an enterprise’s entire business can be programmed, from its front-end web site to its shipping facility and everything in between. And software architecture can be seen as the catalyst that enables this ever-upward spiral by giving us the means to identify the primitives of the next generation’s languages and how those primitives will interact.

I predict it will continue to be that catalyst. As the solutions written using these new mega-programming languages become more and more complex, architecture will once again step in to help “solution architects” (the current term for those who program in these super-languages) to impose order on the chaos, to make sure quality attributes are achieved, and to structure the solution being given to the underlying computing systems. We will see continued movement towards supporting larger, more complex, and more varied problem-space abstractions, backed up by more sophisticated translation or compiler technology that will allow today’s “specification languages” to continue to be tomorrow’s programming languages.

2. References
