The Philosophy of Software Architecture

Amit Goel, RMIT University, Australia

ABSTRACT

Computer Software Intensive systems have become ingrained in our daily life. Apart from obvious scientific and business applications, various embedded devices are empowered with computer software. Such a diverse application of Computer Software has led to inherent complexity in building such systems. As civilizations moved forward, the concept of architectural thinking and practice was introduced to grapple with the complexity and other challenges of creating buildings, skyscrapers, townships, and cities. The Practice of Software Architecture is an attempt to understand and handle similar challenges in Software Intensive Systems. This paper introduces software architecture and the underlying philosophy thereof. This paper provokes a discussion around the present and future of Software Architecture. The authors discuss skills and roles of Software Architect.

Keywords: *Computer Software Intensive Systems, Software Architect, Software Architecture, Software Engineering, Software-Intensive Systems*

INTRODUCTION

The invention of automated machines dates back to seventeenth century. These automated machines were run by mechanical and electrical control mechanisms and performed simple tasks. The advent of electronics based computing machines increased the potential of these machines making them complex. The concept of software has manifested in all forms of computing machines whether mechanical, electrical or electronic, being the lifeline thereof.

As the computer systems became more powerful and smaller in size, their usage diversified from scientific computations to business systems. It wasn't long before they were used to automate various devices such as Phones, Airplanes and Cruise Control Systems in cars. Today most of our devices are embedded with a computer of one kind or another. The diverse usage, heterogeneous systems and structure of computers systems lead to further complexity for software, which has now become the essential part of any computer system, large or small.

In order to manage complexity, a journey of abstractions was observed which passed through machine language (language of 0 and 1), assembly language (language of instructions and mnemonics such as add, load), high level languages the (C, C++, Java) and fourth generation (4GL) or domain specific languages (DSL). From another viewpoint this complexity was being addressed by using concepts such as top-down and bottom-up software development approach. The theory of software design

DOI: 10.4018/jwp.2010100103

and design patterns was formed during these developments. As the complexity increased, the need was felt to make decisions at much higher levels of abstraction, and to make strategic decisions before making tactical (as in design) or operational decisions (as in code). The theory of Software Architecture started taking shape in order to manage the complexity at higher levels of abstraction and to embed strategic decision making in the building of software systems.

In this paper we explore few fundamental thoughts on software architecture to provoke discussion around some basic questions. We start by discussing the meaning and definition of the term 'Software Architecture' in section 2. We ask "Why do we need to do Software Architecture?" in section 3 and hence outline the rationality for doing the software architecture. Section 4 discusses what skills and qualities are required by a software engineer engaged in the practice of software architecture. Section 5 discusses the software architecture metaphor and how is it similar to or different from art, engineering and science. This section leads us to think whether software architecture is an art, science or engineering or a mix of these. We conclude by providing a summary and future direction.

This paper covers few key issues about philosophy of software architecture in breadth. Hence the discussion is brief. However, we point the reader to various references to dive deeper into details of various concepts presented in this paper.

The Pursuit of Software Architecture

Software architecture is a generally overused term. However, if we ask someone about software architecture generally the conversation is like the one below:

"What is software architecture?"

"The set of decisions an architect makes."

"What are these decisions?"

"The architecturally significant ones"

"Ok. What is architecturally significant?"

"The architect decides".

Kent Beck articulated such situation humorously that "Software architecture is what software architects do and therefore by implication what software architects do is, well, they architect software" (Booch, 2006).

Let us first understand the meaning of word architecture in context of computer software. Software engineering community has a common understanding that architecture enables transformation of requirements to code or working application. Yet another view is that architecture is the glue between Business and IT and closes the Business-IT alignment gap. Hence software architecture is positioned in the middle of requirements/code (Figure 1) or business/IT (Figure 2). We do not deny the importance of architecture in both these roles, but mainly software architecture sits in the middle of strategy and implementation (Figure 3). Strategy is the owner's vision and implementation is the execution of strategy. Architecture, positioned in the middle, is architect's blueprint which allows owners to implement or execute their strategy. Positioning architecture in the middle of strategy and execution allows it to scale conceptually from software architecture to enterprise architecture, and relates architecture to important aspects discussed in section one, i.e., strategic decision making and higher levels of abstraction.

We find many definitions of Software Architecture in the literature. Let us have a look at some of them:

- "Software architecture is a set of architectural (or design) elements that have a particular form." (Perry & Wolf, 1992).
- "Software architecture is a collection of computational components—or simply components-—together with a description of the interactions between these