SEMAT Workshop
March 17-19, 2010 Zürich Switzerland

Capturing the Essence of Software Engineering -- A Reflection on SEMAT Vision Statement

Shihong Huang
(shihong@fau.edu)
Dept. of Computer Science & Engineering
Florida Atlantic University

1. A Reflection on Vision Statement

The Software Engineering Method and Theory (SEMAT) initiative [1] aims at addressing some of the prevalent problems of software engineering theory and practices, to revitalize this discipline. Some of the problems mentioned in the Purpose and Scope of the Vision Statement [2] include prevalence of fads, lack of a sound and widely accepted theoretical basis, large number of methods and their variants with differences little understood and artificially magnified, the need of credible empirical evaluation and validation, and the gap between industry and academia.

To address these problems, SEMAT sets five goals: defining the basic definition of software engineering, providing a strong mathematical basis to the discipline, identifying the truly *universal* elements to be integrated into the *kernel*, defining a kernel language that describes the "method elements" – practices, patterns and methods, and providing assessment techniques to evaluating software practice and theories, including the results of SEMAT.

According to the Vision Statement, the two immediate tasks are identifying the Universals and defining the Kernel Language, with whose vision I tend to agree as these two tasks will lay the foundation for the other three tasks.

This position paper discusses some of the candidate aspects and attributes of the Universals to complement what was discussed in Appendix 3. The goal of the Universal track is to identify "the universal elements of software engineering, which must be integrated into the SEMAT kernel", and in the meantime, to "keep the kernel concrete, focused and small". Identifying Universals should be based on the definition of software engineering and their essential concepts of the discipline (Task 1). To some extent, Universals and Definitions are mutually tightly coupled and restrained – Definition defines the scope of Universals, and Universals codify Definition. So, our first task is to have a basic understanding of what "software engineering" is and what the uniqueness of software engineering is from other engineering disciplines.

2. The Difference between Science and Engineering

By simply parsing the term "software engineering", we understand there are two parts which constitute this discipline – one is "software", the other is "engineering", hence the obvious definition: "Software engineering is the application of engineering methods and discipline to the field of software" [A1.1]. Whether this definition is sufficient or precise [3], we leave it to Track 1 to answer. However, one would agree that software engineering is indeed an engineering discipline. Therefore we should treat software engineering the engineering way. Having said that, there is significant difference between science and engineering. In Henry Petroski's recent book [4] he points out the difference of the two endeavors as:

- Science seeks to understand what is, whereas
- Engineering seeks to create what never was

It is not appropriate to describe engineering as mere applied science. There are some extra-scientific components to engineering, something often referred to as the creative nature, situated

culture and particularity to a specific application domain. In engineering, we often do the design first, and then the hypothesized structure can be given a scientific or mathematical litmus test [4]. It is essential to keep in mind the similarities and differences between science and engineering when we give the definition of software engineering and define the Universals. In engineering, analysis follows synthesis and observation --- not the other way around. Unlike science that deals with the universal laws, which are context and time independent and true everywhere, engineering deals with situated culture, needs to have constant learning, refinement and adaptation to meet the environmental requirements.

3. The Proposed Universals Hierarchy

Given the nature of software engineering discipline, the Universals should be a hierarchical structure which consists of three layers of components from general to specific: Engineering Universals, Software Engineering Universals, and Domain Specific Software Engineering Universals. They are shown in Figure 1.

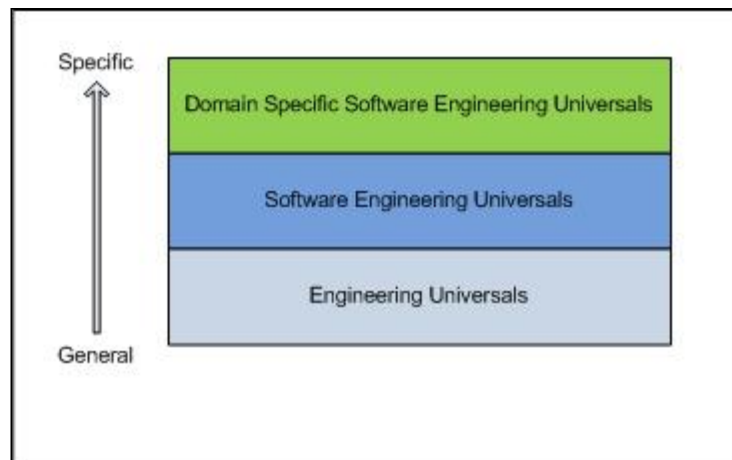

Figure 1: Universals Hierarchy

(1) Engineering Universals: this layer includes the best practices of engineering disciplines, such as civil engineering, mechanical engineering, electrical engineering, that apply to software engineering. This is the foundation of the "engineering" aspect of the Universals (e.g., project, team, system, quality etc as mentioned in A3.4)

(2) Software Engineering Universals: this layer includes the unique best practices to software engineering, such as extensibility, interoperability, evolveability, reusability, maintainability etc. This is the "software" aspect of the Universals.

(3) Domain Specific Software Engineering Universals: this layer addresses domain specific universals, such as for real-time systems, self-adaptive systems, self-management systems, Web application systems etc. This is the "variability" equivalent to software product line approach.

The following is further elaboration on the hierarchy.

Since software engineering is an engineering discipline, it possesses some of the engineering fundamentals. For example, it should follow some of the ingredients of engineering project management [5] for a project, i.e., transformation, flow, and value generation. From a management perspective, it should have the stages of planning, execution and controlling.

While software engineering follows the engineering fundamentals, there are some unique features of software engineering and software products that are different from general engineering and engineering products and must be addressed in order to reflect the nature of software. For example, for the engineering model, there are full specification, followed by design, manufacture, test, install, maintain; whereas for the software model, often we have incomplete specifications, and the first 3 stages (design, manufacture and test) are often blurred. The final product has no physical system and software doesn't wear out.

One prominent feature that a software product is different from a general engineering products is that software needs to be consistently maintained and evolved to meet new business requirements [6]. The evolution is more important in software than in other engineering disciplines. Software engineering rarely involves "green field" development because most organizations have substantial legacy systems. The legacy systems represent significant assets containing valuable components that can be reused (through reverse engineering [7]) as the system evolves over time to meet changing requirements and new business challenges. The cost incurred in changing the software after its deployment are likely to exceed the development cost by a factor of 3 or 4 [8].

Given the malleable nature of software, a good collection of Universals should include not only the general engineering universals that capture the core practices of engineering disciplines, but also some of the unique features of software, such as changeability, extensibility, interoperability, evolveability, and software reuse [9].

The uniqueness of software determines some of the Universal attributes we must have, these attributes differentiate software engineering from other engineering disciplines.

Besides following the common practices of engineering disciplines and uniqueness features of software engineering, the last layer would be domain specific software engineering Universals that reflect and address the knowledge of different more situated application domains.

References:

[1] Software Engineering Method and Theory (SEMAT) online at www.semat.org

[2] SEMAT Vision Statement online at http://www.semat.org/pub/Main/WebHome/SEMAT-vision.pdf

[3] A. Cockburn. The end of software engineering and the start of economic gaming. http://alistair.cockburn.us/The+end+of+software+engineering+and+the+start+of+economic-cooperative+gaming).

[4] Henry Petroski *The Essential Engineer: Why Science Alone Will Not Solve Our Global Problems*, Knopf, February 23, 2010

[5] L. Koskela, Lauri and G. Howell. The underlying theory of project management is obsolete. http:// www.leanconstruction.org/pdf/ObsoleteTheory.pdf

[6] K. Bennett, V. Rajlich. "Software maintenance and evolution: a roadmap." International Conference on Software Engineering Proceedings of the Conference on The Future of Software Engineering (ICSE 2000: Limerick, Ireland). pp. 73-87.

[7] H. Müller, J. Jahnke, D. Smith, M-A. Storey, S. Tilley, and K. Wong. "Reverse engineering a roadmap." International Conference on Software Engineering Proceedings of the Conference on The Future of Software Engineering (ICSE 2000: Limerick, Ireland). pp. 47-60.

[8] I. Sommerville. *Software Engineering* 8th Edition, Addison Wesley June 2006.

[9] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley Professional June 1, 1997.