Running Head: ARCHITECTURE VERSUS ENGINEERING

MetaTech Consulting, Inc.

White Paper

Comparative Analysis of Architecture and Engineering as Software Disciplines

Jim Thomas

July 26, 2003

Abstract

This paper presents a discussion of the emergence of the Software Architecture discipline in comparison to the more mature Software Engineering discipline. A treatment of the civil construction metaphor is provided as a foundation to the rationalization of the software disciplines as rigorous activities worthy of being associated with the term *engineering*. That is followed by a discussion on the practical application of the metaphor – how practitioners view the partitioning of the disciplines and their relationship to traditional engineering fields. A discussion on the relative maturity of the disciplines is also provided.

Comparative Analysis of Architecture and Engineering as Software Disciplines

The software industry undertakes efforts of striking magnitude and complexity with the best of intentions only to achieve total success on a fraction of the starts. The rate that technology continues to evolve easily surpasses any software development team's ability to assimilate it. Additionally, the complexity of the systems being built is such that it is simply not possible to manage all details of its creation in one's mind. Nearly a half-century ago it was realized that building software required the same discipline and rigor as had been applied in more traditional engineering fields for centuries. Simply adopting a metaphor does not resolve all the woes of an industry. The not-quite-perfect fit of the civil engineering metaphor has left the emerging disciplines with not-quite-trivial issues that continue to plague them. Research on the discipline of Software Engineering and Software Architecture is ongoing and has been well documented by practitioners and researchers such as W. Maier, M. Shaw, F. Brooks, A. Bryant, and D. Garlan. Individually, and in combination, works by these authors have profoundly affected the maturation of Software Engineering and Software Architecture. The purpose of this paper is to bring together under a single cover some of the salient points from these and other authors regarding the interrelationship of the disciplines as well as their continued maturation.

The Metaphor

Metaphors are a technique humans use to aid in communications as well as to assist in assimilating foreign ideas, thoughts, and concepts. They are a powerful tool that can provide cognitive context through abstractions. They provide a firm foundation which understanding and learning can build on. In describing the thing, this author has himself founded his assertions on a metaphor, albeit unintentionally. Yes, metaphors are both prevalent and a powerful construct. It

is not surprising one was chosen to aid in understanding the emergent software practices more than 40 years ago.

The foundation metaphor on which the software industry has evolved is based on construction. *Engineering*, a term lifted directly from the civil construction, have become the discipline classifiers within the software industry. References to applications of this metaphor to the infant software discipline are present as far back as 1958 (Brooks, 1987). The language of the discipline has clearly accommodated the metaphor as typified by terms such as *requirements, specification, independent verification and validation, interfaces,* etc.

Architecture¸ a natural extension to *Engineering*, was first suggested for use as a classifier within the software industry in the early 1960s though it did not achieve general use until its use at a 1969 NATO conference (Bryant, 2000). Though the term entered general use, only through its roots from the like-named civil discipline did it have general meaning. It was not until 2002 that consensus was achieved of the meaning of the term in the domain of software (Maier, Emery, & Hillard, 2001). For practitioners entering the profession within the past 20 years, any alternate verbiage seems wholly inappropriate and unsuitable.

Formal Definitions

To support further discussion on the topic, it is necessary to make explicit the formal definitions of the relevant terms. Appreciation for the following terms will ensure a consistent context for subsequent analysis.

Software Engineering. The IEEE Standards Collection: Software Engineering (as cited by Pressman, 2001, p. 20) defines Software Engineering as "(1) The application of a systematic,

disciplined, quantifiable approach to the development, operation, and maintenance or software: that is, the application of engineering to software. (2) The study of approaches as in (1)."

Software Architecture. In describing architecture as "a representation of overall structure, or a representation that captures the essential features of a system while masking the unimportant details", Maier (1996) likens systems architecture to civil architecture in that it communicates style, decoration, and patterns of organization. The IEEE Standards Collection: Software Engineering (as cited by Maier, 2001, p. 108) defines architecture as "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution."

The preceding paragraph may be interpreted to say that architecture is design. Clements (n.d.) supports this generalization though he adds a caveat that aims to make clear that not all design is architecture. He distinguishes the two as follows:

> The architect draws the boundary between architectural and nonarchitectural design by making those decisions that need to be bound in order for the system to meet its development, behavioral, and quality goals. (Decreeing what the modules are achieves modifiability, for example.) All other decisions can be left to downstream designers and implementers.

Architectural description. Maier (2001, p. 108) describes an architectural description as "a concrete artifact" that communicates the architecture. The IEEE Standard 1471 places normative requirements of these artifacts. These descriptions, contained in one or more *architectural views*, must specify the systems stakeholders and address their architectural concerns including: (a) functionality, (b) performance, (c) security, and (d) feasibility.

Additionally, an architectural description must make explicit the rational for making key architectural decisions.

Architectural views. The IEEE Standards Collection: Software Engineering (as cited by Maier, 2001, p. 108) defines architectural views as "a collection of models that represent one aspect of an entire system." Such a model is relevant only to the system being defined. It does not provide a generalization suitable across multiple systems.

Practical Application of the Metaphor

There is a tendency to attempt to partition the roles of architecture and engineering in such a way that there is clear distinction between them. While doing so makes it far easier to argue the need for the coexistence of the disciplines, it is not a desirable model. Rather, the roles of system engineer and system architect coexist on a single continuum that is on one end is quantitative and the other end is qualitative (Maier, 1996). An architect, being the trusted agent for the client, generally follows an inductive path to help articulate the objective capabilities of what is to be built. The architect produces one or more views that of the objective system, *architectural views*, that communicate to the engineer what is to be built and what is necessary to ensure customer satisfaction. The engineer, being concerned with quantifiable costs and then follows a deductive path by applying mathematics and hard science to achieve technical optimization in designing the objective system.

The architect is active in the conceptualization, scoping, partitioning, specification, and certification of a system. Clients and engineers generally do not share a common vocabulary. Clients state requirements in terms of their business needs. Engineers require far more specificity before they can perform detailed design with the rigor mandated by their discipline.

The architect must be able to interpret the client's business needs and translate them into high level specifications. She will capture the relevant details and represent them in architectural views that address each of the clients concerns. Multiple iterations will likely be needed to ensure that all of the scenarios that characterize the business needs. The architect will partition the architectural components in what is believed to be the best configuration – this will generally be in accordance with well established architectural styles that have performance characteristics appropriate for the clients needs. Once complete, the architectural views are conveyed to the engineers such that they can translate them into specifications suitable for implementation. The architect continues her involvement to ensure the intent of the client's needs are given deference when any ambiguity in the specifications exist or when latitude exists for engineering or implementation trades. It is the architect, on behalf of the client, that is responsible certifying the design and resulting system satisfies the client's needs. Having done so, the architect will greatly increases the probability of ultimate acceptance by the client.

Furthermore, on the subject of increasing probability of success, Maier and Rechtin (2002) emphasize the necessity to have both absolute consistency and completeness of interface descriptions and a disciplined methodology when undertaking a large, complex effort. Again, the architect is active throughout to lifecycle to provide the necessary consistency.

Maturity of the Disciplines

The path to maturity of a given software technology follows a general progression from its inception to the point that it is ready for popular use. Redwine and Riddle (as cited in Shaw, 2001) identified six phases to that maturity path. Those are (a) basic research, (b) concept formalization, (c) development and extension, (d) internal enhancement and exploration, (e)

external enhancement and exploration, and (f) popularization. These phases are neither exclusive nor measured. It is common for activities identified with two or more phases to occur concurrently. And, the time which passes for a given technology to fully transition through a given phase is independent from the time necessary for it to transition through other phases and from the time other technologies pass through that same phase. Redwine and Riddle asserted that the time needed for full maturity of software technology is approximately 15 to 20 years.

System Engineering

As a discipline and, arguably, a technology, Software Engineering has existed since 1968 (Bryant, 2000). By the 1980s professional certifications and university courses supporting the discipline emerged. It has matured from the world of theory to practice and practice having achieved full popularization so that there is little dispute that the engineering processes and rigor are ubiquitous in the software industry.

Software Architecture

Mary Shaw (2001), in describing the technology in the aforementioned maturity context, has stated Software Architecture has matured to the third phase – development and extension. This is substantiated through the recognition that through Software Architecture was initially thought useful only in providing qualitative descriptions of various organizations of software components, it now has been extended to include formal notations, specialized tools, and discrete analysis techniques. Shaw asserts the degree of maturity of Software Architecture, having been initiated in 1992 with the release of the first publications on the topic, continues to progress within the 15-20 year timeframe offered by Redwine and Riddle (1985). Shaw summarizes the maturity of software architecture as follows:

> We see software architecture is reaching the point of growing from its adolescence in research laboratories to the responsibilities of maturity. This brings with it additional responsibility to show not just that ideas are promising (a sufficient grounds to continue research) but also that they are effective (a necessary grounds to move into practice). (p. 661)

Heuristics. Over time and through experience man learns. In the absence of effective communications, experience gained by one generation of practitioners is lost to the next. In the early days of the Software Architecture discipline, there was no effective means of communicating lessons learned (i.e. heuristics) beyond ones immediate associates. As there were neither academic nor professional organizations to facilitate propagation of knowledge among the community of system architects (let alone software architects) until the mid-1980s, learning across the community was generally slow. Maier & Rechtin (2002, p. 22) chronicles an effort by a University of Southern California graduate course to address this deficiency undertaken in the late-1980s that "gathered together lessons learned throughout the West Coast aerospace, electronics, and software industries and expressing them in heuristic form for use by architects, educators, researchers, and students." Through these abstractions of experiences individual architects can begin to learn from the collective whole of the community and, through contributions to the forum, extend the learning beyond their associates and to those who follow.

Standards. One may expect an essay on architecture and engineering to include a significant number of references to industry or international standards. Though individual technologies related to Software Engineering and Architecture have numerous standards to provide guidance, the disciplines themselves do not. Shaw and Garlan (1996) address the absence of standards for notations, theories, and analytical techniques as well as the need for a

well-accepted taxonomy. A consensus on good architectural description practices, documented in IEEE Standard 1471 was approved by Computer Society in 2000. This is one of the first actions that is characteristic of a maturing discipline.

Conclusions

As the discipline of Software Architecture matures, and as more certifications and university degrees become available, one may assume that it is becoming ever easier for a practitioner to recognize when he is prepared to be an architect. When is an engineer prepared to become an architect? With only little more than cursory consideration for that question, it may become apparent that it is deceptively simple. Sathyanarayana Pandurange (n.d.) asserts that an engineer is prepared to become an architect when he has gained enough breadth and depth in the relevant domain and technologies, and has the capacity to envision a software system before it is developed." This implies knowledge and skill – both of which can be obtained through formal and informal training – coupled with an artful, enlightened ability to envision what is possible. This last attribute is beyond education. To some extent, one must be predisposed to think like an architect – that is an innate ability difficult to impose through training. This ability, coupled with the wisdom that comes only through experience, is key to success as an architect.

The imagery conjured up through the engineering/architecture metaphor has finite limits. In fact, there continues to be wide debate if software engineering constitutes a true engineering discipline at all. Bryant (2000) encapsulates the debate when he quotes I. Summerville as follows: "Software engineering differs from other forms of engineering since it is not constrained by materials governed by physical laws or by manufacturing processes" (p. 78). While the metaphor has served the discipline well for several decades, the question asking if it is time to

evolve to a new metaphor – a new paradigm that provides a more accurate representation of the agile business environment that is driving so much of the software industry of today.

The waterfall model that provided an effective methodology for so many software-intensive projects as the software industry matured is less and less used by practitioners. That model requires full understanding of requirements prior to initiating structured analysis and design. Furthermore, any change to those requirements prior to project completion resulted in a dramatic increase of cost and schedule together with an increased probability of programming errors. Pressman (2001, p.198) illustrates that level of effort required to correct an error (e.g. a new or missed requirement) identified prior to field operations can be as high as 70 times that if it is identified during the requirements phase (see Figure 1). Monolithic projects of great scale and complexity, particularly those that also provide for safety or security of there intended users, continue to mandate the rigor or the waterfall model and the structured analysis and design methodologies. Such an undertaking is done with the full belief that the requirements will be complete and unchanging throughout the lifecycle.

But, what about the remaining development activities? In this day and age when time-to-market is a critical factor in attaining and maintaining market share, and when it is common for requirements to evolve much more quickly than the time it takes to build a system, alternate methodologies have become far more prevalent. Pressman (2001), together with many other texts on the subject of Software Engineering, document a wide variety of alternate methodologies (i.e. evolutionary, linear sequential, prototyping, etc.). Bryant (2000) suggests that as the industry continues to shift away from linear sequential (e.g. the waterfall model) toward evolutionary (e. g. spiral and incremental) methodologies, another metaphor may now be more appropriate. He suggests one based on *growth* is now more appropriate than the existing

one that is based on *construction*. The notion of shifting metaphors, suggested first in the mid-1980s, continues to prompt debate within academia and among practitioners.

References

Brooks, F. P. (1987, April). No silver bullet: Essences and accidents of software engineering. *IEEE Computer*, 10-19.

Bryant, A. (2000). *It's engineering Jim.…but not as we know it: Software engineering – solution to the software crisis, or part of the problem?* Proceedings, 22nd international conference on Software engineering, 77-86. ACM Press New York, NY, USA.

Clements, S. (n.d.). What's the difference between architecture and design? Essays on Software Architecture. Retrieved July 26, 2003 from http://www.sei.cmu.edu/architecture/essays.htm

Maier, M. W. (1996, October). Developments in System Architecting. *2nd IEEE International Conference on Engineering of Complex Computer Systems*, 139 – 142.

Maier, M. W., (2001, April) Software architecture: Introducing IEEE standard 1471. *Computer*, 107-109.

Maier, W.M. & Rechtin, E. (2002). *The art of systems architecting (2nd ed.)*. CRC Press LLC., Boca Raton, FL, 33431.

Panduranga, S. (n.d.). Are you a software architect yet? Essays on Software Architecture. Retrieved July 26, 2003 from http://www.sei.cmu.edu/architecture/essays.htm

Pressman, R. S. (2001). *Software engineering: A practitioner's approach (5th ed.)*. McGraw-Hill, New York, NY, 10020.

Redwine, S. & Riddle, W. (1985, May). Software technology maturation. *Proceedings of the 8th International Conference on Software Engineering*, pp. 189 – 200. IEEE Computer Society Washington, DC, USA.

Shaw, M. (2001). *The coming-of-age of software architecture*. Proceedings of the 23rd international conference on Software engineering, 656 – 663.

Shaw, M. & Garlan, D. (1996). *Software architecture: Perspectives on an emerging discipline*. Prentice-Hall, Inc. Upper-Saddle River, New Jersey 07458.

Figure 1

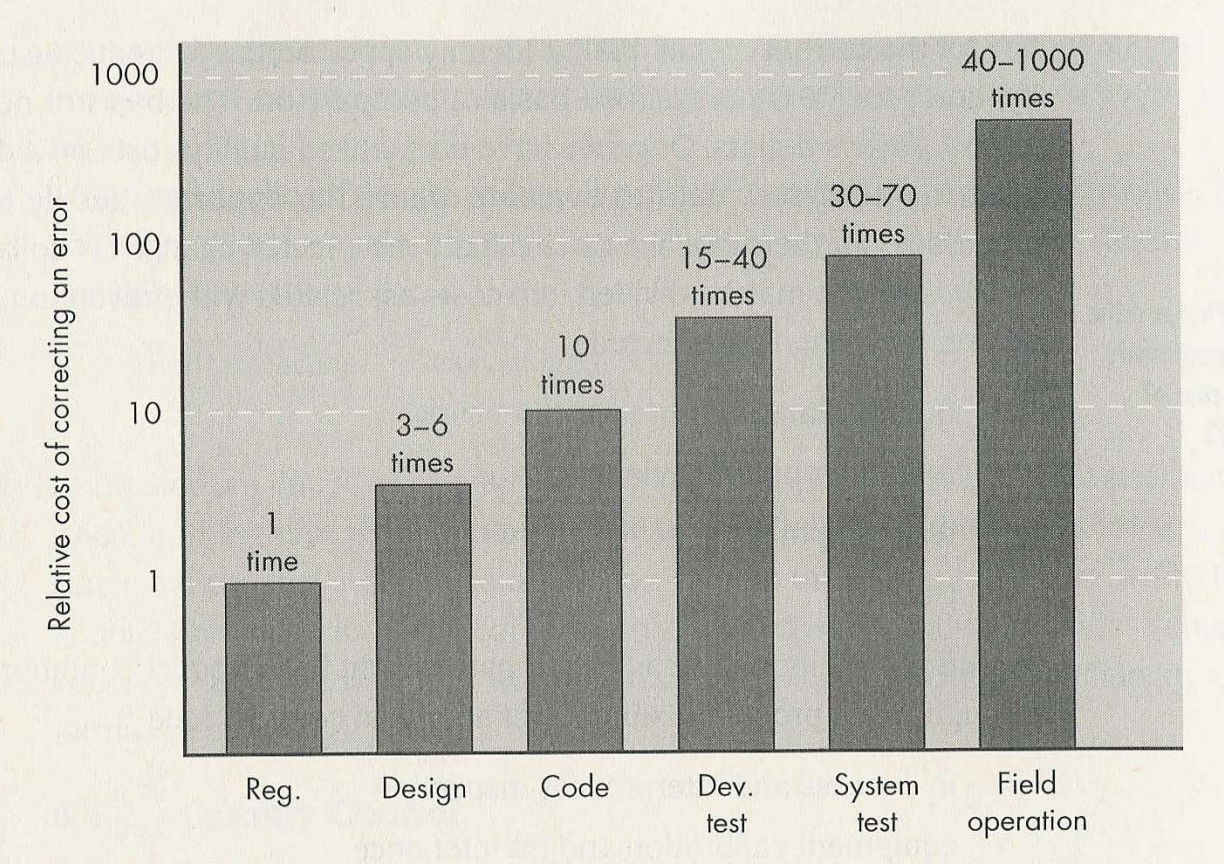

Figure 1. Relative cost of correcting an error (Pressman, 2001, p 198)