SEMAT Position Paper

Donald Firesmith (Software Engineering Institute)

I believe that the SEMAT initiative must address the following:

- The kernel must be widely applicable.
- Software engineering is a subdiscipline of system engineering.
- Poorly engineered requirements are a major cause of project failure.
- Quality is far more than merely a lack of defects.

1 The kernel must be widely applicable.

1.1 The Challenge

Although it is important to derive the benefits of a de facto industry standard kernel, it is critical that this kernel be sufficiently flexible to address the many different situations we face as software engineers. We may argue about the details of the following taxonomy, but it is nevertheless clear that software engineering is used to engineer an extremely wide range of systems having very different stakeholders and organizations as well as projects with quite different characteristics:

- **System Variability:**
  - **Complexity –** Software-reliant systems vary greatly from the trivially simple to the ultra complex in terms of size, number of different missions supported or performed, number of requirements implemented, number and complexity of their component subsystems, number and complexity of the relationships, interfaces, and interactions between these subsystems, heterogeneity of the subsystems in terms of application domain and technologies, as well as complexity of these technologies.
  - **Criticality –** Software-reliant systems vary greatly in criticality in terms of business-, mission-, safety-, and security-criticality.
  - **Negative Emergence –** Although all software-reliant systems have emergent behavior and characteristics, they vary greatly in terms of the amount and severity of unexpected detrimental emergence.
  - **Evolution –** Software-reliant systems vary greatly in terms of their rate of evolution due to changing requirements, the identification and correction of defects and vulnerabilities, the technical refresh rate of updating evolving technologies, changes in external interfacing systems, the degree of independent governance of subsystems, and the addition, modification, and removal of subsystems.
  - **External Coupling –** Software-reliant systems vary greatly in terms of the degree to which they are coupled to and must interface and interoperate with external systems.
  - **Intelligence –** Software-reliant systems vary greatly in intelligence and the degree to which they operate autonomously without human control.
  - **Physical Distribution –** Software-reliant systems vary greatly in terms of the degree to which their instances and subsystems are physically distributed from contiguous systems to systems, the components of which are distributed across the planet and even across the Solar System.
  - **Requirements Risk –** Software-reliant systems vary greatly in terms of completeness, quality, and management.

  - **Reuse –** Software-reliant systems vary greatly in terms of the degree to which components and their technologies are reused as well as the type of reuse (e.g., COTS, GOTS, MOTS, and open source).
  - **Size –** Software-reliant systems vary greatly in terms of size from the trivially small to the ultra-large scale.
  - **Technology Maturity –** Software-reliant systems vary greatly in terms of technology maturity from low-risk, stable, and mature technologies to those that are rapidly evolving, immature, and bleeding-edge.
  - **Variability –** Software-reliant systems vary greatly in terms of variability from single instance systems to product lines of multiple variants, to highly configurable, internationalized and even personalized systems.
- **Quality Characteristics and Attributes** – Software-reliant systems vary greatly in their quality characteristics and attributes in terms of the degree to which these qualities are required, whether or not these they are well-defined in an associated quality model, and the degree to which the stakeholders agree on their meanings and importance.
- **Stakeholder Characteristics** – Software-reliant systems vary greatly in terms of the number, types, authority of, volatility of, and degree of trust in and between the stakeholders as well as the degree to which the software engineers have access to them.
- **Organizational Characteristics** – Software-reliant systems vary greatly in terms of their associated organizations such as number, types, and sizes of the organizations, the management and engineering culture, the geographic distribution of the organizations and their staffs, as well as staff experience and expertise.
- **Project Characteristics** – Software-reliant systems vary greatly in terms of their associated projects such as contract types, number, scope, duration, and amount and adequacy of schedule and funding.

I contend that the above variability makes it impossible for any single software engineering method to be sufficiently flexible and tailorable to meet all situations. In fact, individual software engineering efforts may need multiple methods for different subsystems as well as for different organizations (e.g., prime contractor, system integrator, and subcontractors).

1.2 A Proposed Solution

With all of this variability, no single software or system development method can possibly be optimal for every situation. That is why I am greatly in favor of taking a situational method engineering approach so that the process is appropriate for the specific needs of the organization, stakeholders, system, etc. That is why I also favor capturing industry best practices in the form of reusable method components.

Thus to meet this need for extreme flexibility while still deriving the benefits of standardization, I suggest that the kernel either is (or at least contains) a software engineering method framework based on situational method engineering. Specifically, this framework should contain the following four parts: (1) an ontology defining the concepts and terminology of software engineering, (2) a metamodels defining the foundational abstract classes of method components, (3) a repository of free, open-source, reusable method components such as work products (models, documents, software) to be produced, work units (activities, tasks, and techniques) to be performed to produce the work products, and workers (engineers, teams, and tools) to perform the work units , and (4) a metamethod for selecting appropriate method components, tailoring them as necessary, adding additional method components when necessary, integrating them to produce one or more appropriate situation-specific methods, verifying these methods, and publishing them to the relevant stakeholders. This

gives us the benefits of both flexibility (in selection and tailoring of method components) as well as standardization (of ontology, method components, and metamethod).

Hopefully, this approach will help us get past "My method is better than your method", regardless of the situation.

2 Software engineering is a subdiscipline of system engineering.

In many domains, it is impossible to address software without considering hardware and other types of system components. For example, the so-called "software" quality characteristics are actually a function of both software and hardware (and often people) and are therefore system quality characteristics. This places software engineering as a subdiscipline within systems engineering. This raises the question of "What is the relationship between SEMAT and SystemsEMAT?" Just as hardware engineers sometimes ignore software engineering to the disadvantage of the project, system, and its stakeholders, software engineers sometimes do the same. Also, what is the relationship between software engineering and specialty engineering areas such as reliability engineering, safety engineering, and security engineering? While SEMAT should bound itself so that we don't try to drink the ocean, I think we will do ourselves a disservice if we try to look at methods and theory from only a pure software viewpoint. Our methods and theory have to be consistent with and incorporated into a framework of systems methods and theory.

3 Poorly engineered requirements are a major cause of project failure.

Numerous studies have shown that poor requirements are a major root cause of project failure in terms of cost overruns, schedule slippages, poor quality, functionality not delivered, and systems delivered not being utilized or utilized fully. Whatever we do, we must make sure that SEMAT addresses the reasons why we constantly produce requirements that do not have the appropriate characteristics (e.g., correct, complete, verifiable, unambiguous, feasible,…). It is impossible to review requirements specifications in practice and not find 5-10 of defects per page. As bad as the situation is with regard to functional requirements, it is far worse with other requirements such as data requirements, interface requirements, architecture/design/implementation constraints, and especially quality requirements which are notorious for being missing, infeasible, ambiguous, and unverifiable. Requirements defects are the most expensive and difficult to fix, and yet they are constantly being injected at an unacceptably high rate. Poor requirements are also a major (if not the major) root cause of accidents involving software-reliant systems. I predict that our work will be judged a failure if we do not do something about the current miserable state of affairs regarding requirements.

4 Quality is far more than merely a lack of defects.

One often hears that quality is synonymous with meeting requirements. Yet as the above issue on requirements incompleteness and poor quality, you cannot build quality purely on the concept of correctness. There are many types of quality and they must be properly defined and agreed upon in a system/project quality model of the quality characteristics (ilities), their component quality attributes, and the associated measurement scales and methods for determining the [required and actual] levels of the different quality attributes. In spite of ISO standardization, quality models are either rarely or very poorly used in practice. SEMAT must improve software engineering's poor record on defining, mandating, and achieving required levels of these different types of quality.