Key Points a Systems Engineer Needs to Know about Software Engineering

Lead Author: *Dick Fairley,* ***Contributing Author:*** *Alice Squires*

The field of software engineering is extensive and specialized. Its importance to modern systems makes it necessary for systems engineers to be knowledgeable about software engineering and its relationship to systems engineering.

□

Contents

Key Concepts a Systems Engineer Needs to Know about Software Engineering

The following items are significant aspects that systems engineers need to know about software and software engineering. Most are documented in (Fairley and

Willshire 2011):

1. **For the time, effort, and expense devoted to developing it, software is more complex than most other system components** - Software complexity arises because few elements in a software program (even down to the statement level) are identical, as well as because of the large number of possible decision paths found even in small programs, with the number of decision paths through a large program often being astronomical. There are several detailed references on software complexity. The SWEBOK (Bourque and Fairley 2014) discusses minimizing complexity as part of software construction fundamentals. Zuse (1991) has a highly cited article on software complexity measures and methods. Chapters 2 and 3 of the SWEBOK also have further references.
2. **Software testing and reviews are sampling processes** - In all but the simplest cases, exhaustive testing of software is impossible because of the large number of decision paths through most programs. Also, the combined values of the input variables selected from a wide combinatorial range may reveal defects that other combinations of the variables would not detect. Software test cases and test scenarios are chosen in an attempt to gain confidence that the testing samples are representative of the ways the software will be used in practice. Structured reviews of software are an effective mechanism for finding defects, but the significant effort required limits exhaustive reviewing. Criteria must be established to determine which components (or sub-components) should be reviewed. Although there are similar concerns about exhaustive testing and reviewing of physical products, the complexity of software makes software testing, reviews, and the resulting assurance provided more challenging. Other points include:
    1. All software testing approaches and techniques are heuristic. Hence, there is no universal "best" approach, practice, or technique for testing, since these must be selected based on the software context.
    2. Exhaustive testing is not possible.
    3. Errors in software tend to cluster within the software structures; therefore, any one specific

approach or a random approach to testing is not advised.

4. Pesticide paradox exists. As a result, running the same test over and over on the same software-system provides no new information.
5. Testing can reveal the presence of defects but cannot guarantee that there will be no errors, except under the specific conditions of a given test.
6. Testing, including verification and validation (V&V), must be performed early and continually throughout the lifecycle (end to end).
7. Even after extensive testing and V&V, errors are likely to remain after long term use of the software.
8. Chapter 4 of the SWEBOK discusses software testing and provides a bibliography.

3. **Software often provides the interfaces that interconnect other system components** - Software is often referred to as the *glue* that holds a system together because the interfaces among components, as well as the interfaces to the environment and other systems, are often provided by digital sensors and controllers that operate via software. Because software interfaces are behavioral rather than physical, the interactions that occur among software components often exhibit emergent behaviors that cannot always be predicted in advance. In addition to component interfaces, software usually provides the computational and decision algorithms needed to generate command and control signals. The SWEBOK has multiple discussions of interfaces: Chapter 2 on Software Design is a good starting point and includes a bibliography.
4. **Every software product is unique** - The goal of manufacturing physical products is to produce replicated copies that are as nearly identical as much as possible, given the constraints of material sciences and manufacturing tools and techniques. Because replication of existing software is a trivial process (as compared to manufacturing of physical products), the goal of software development is to produce one perfect copy (or as nearly perfect as can be achieved given the constraints on schedule, budget, resources, and technology). Much of software development

involves altering existing software. The resulting product, whether new or modified, is uniquely different from all other software products known to the software developers. Chapter 3 of the SWEBOK provides discussion of software reuse and several references.

5. **In many cases, requirements allocated to software must be renegotiated and reprioritized** - Software engineers often see more efficient and effective ways to restate and prioritize requirements allocated to software. Sometimes, the renegotiated requirements have system-wide impacts that must be taken into account. One or more senior software engineers should be, and often are, involved in analysis of system-level requirements. This topic is addressed in the SWEBOK in Chapter 1, with topics on the iterative nature of software and change management.
6. **Software requirements are prone to frequent change** - Software is the most frequently changed component in complex systems, especially late in the development process and during system sustainment. This is because software is perceived to be the most easily changed component of a complex system. This is not to imply that changes to software requirements and the resulting changes to the impacted software can be easily done without undesired side effects. Careful software configuration management is necessary, as discussed in Chapter 6 of the SWEBOK, which includes extensive references.
7. **Small changes to software can have large negative effects** (A corollary to frequently changing software requirements: *There are no small software changes*) - In several well-known cases, modifying a few lines of code in very large systems that incorporated software negatively impacted the safety, security, and/or reliability of those systems. Applying techniques such as traceability, impact analysis, object-oriented software development, and regression testing reduces undesired side effects of changes to software code. These approaches limit but do not eliminate this problem.
8. **Some quality attributes for software are subjectively evaluated** - Software typically provides the interfaces to systems that have human users and operators. The intended users and operators of these

systems often subjectively evaluate quality attributes, such as ease of use, adaptability, robustness, and integrity. These quality attributes determine the acceptance of a system by its intended users and operators. In some cases, systems have been rejected because they were not judged to be suitable for use by the intended users in the intended environment, even though those systems satisfied their technical requirements. Chapter 10 of the SWEBOK provides an overview of software quality, with references.

9. **The term *prototyping* has different connotations for systems engineers and software engineers** - For a systems engineer, a prototype is typically the first functioning version of hardware. For software engineers, software prototyping is primarily used for two purposes: (1) as a mechanism to elicit user requirements by iteratively evolving mock-ups of user interfaces, and (2) as an experimental implementation of some limited element of a proposed system to explore and evaluate alternative algorithms. Chapter 1 of the SWEBOK discusses this and provides excellent references.
10. **Cyber security is a present and growing concern for systems that incorporate software** - In addition to the traditional specialty disciplines of safety, reliability, and maintainability, systems engineering teams increasingly include security specialists at both the software level and the systems level in an attempt to cope with the cyber-attacks that may be encountered by systems that incorporate software. Additional information about System Security can be found in the Systems Engineering and Quality Attributes Part.
11. **Software growth requires spare capacity** - Moore's Law no longer fully comes to the rescue (Moore, 1965). As systems adapt to changing circumstances, the modifications can most easily be performed and upgraded in the software, requiring additional computer execution cycles and memory capacity (Belady and Lehman 1979). For several decades, this growth was accommodated by Moore's Law, but recent limits that have occurred as a result of heat dissipation have influenced manufacturers to promote potential computing power growth by slowing down the processors and putting more of them on a chip. This requires software developers to revise their

programs to perform more in parallel, which is often an extremely difficult problem (Patterson 2010). This problem is exacerbated by the growth in mobile computing and limited battery power.

12. **Several Pareto 80-20 distributions apply to software** - These refers to the 80% of the avoidable rework that comes from 20% of the defects, that 80% of the defects come from 20% of the modules, and 90% of the downtime comes from at most 10% of the defects (Boehm and Basili 2001). These, along with recent data indicating that 80% of the testing business value comes from 20% of the test cases (Bullock 2000), indicate that much more cost-effective software development and testing can come from determining which 20% need the most attention.
13. **Software estimates are often inaccurate** - There are several reasons software estimates are frequently inaccurate. Some of these reasons are the same as the reasons systems engineering estimates are often inaccurate: unrealistic assumptions, vague and changing requirements, and failure to update estimates as conditions change. In addition, software estimates are often inaccurate because productivity and quality are highly variable among seemingly similar software engineers. Knowing the performance characteristics of the individuals who will be involved in a software project can greatly increase the accuracy of a software estimate. Another factor is the cohesion of the software development team. Working with a team that has worked together before and knowing their collective performance characteristics can also increase the accuracy of a software estimate. Conversely, preparing an estimate for unknown teams and their members can result in a very low degree of accuracy. Chapter 7 of the SWEBOK briefly discusses this further. Kitchenam (1997) discusses the organizational context of uncertainty in estimates. Lederer and Prasad (1995) also identify organizational and management issues that increase uncertainty; additionally, a recent dissertation from Sweden by Magazinus (2012) shows that the issues persist.
14. **Most software projects are conducted iteratively** - "Iterative development" has a different connotation for systems engineers and software engineers. A fundamental aspect of iterative software development is that each iteration of a software

development cycle adds features and capabilities to produce a next working version of partially completed software. In addition, each iteration cycle for software development may occur on a daily or weekly basis, while (depending on the scale and complexity of the system) the nature of physical system components typically involves iterative cycles of longer durations. Classic articles on this include (Royce 1970) and (Boehm 1988), among others. Larman and Basili (2003) provide a history of iterative development, and the SWEBOK discusses this in life cycle processes in Chapter 8.

15. **Teamwork within software projects is closely coordinated** - The nature of software and its development requires close coordination of work activities that are predominately intellectual in nature. Certainly, other engineers engage in intellectual problem solving, but the collective and ongoing daily problem solving required of a software team requires a level of communication and coordination among software developers that is of a different, more elevated type. Highsmith (2000) gives a good overview.
16. **Agile development processes are increasingly used to develop software** - Agile development of software is a widely used and growing approach to developing software. Agile teams are typically small and closely coordinated, for the reasons cited above. Multiple agile teams may be used on large software projects, although this is highly risky without an integrating architecture (Elssamadisy and Schalliol 2002). Agile development proceeds iteratively in cycles that produce incremental versions of software, with cycle durations that vary from one day to one month, although shorter durations are more common. Among the many factors that distinguish agile development is the tendency to evolve the detailed requirements iteratively. Most agile approaches do not produce an explicit design document. Martin (2003) gives a highly cited overview.
17. **Verification and validation (V&V) of software should preferably proceed incrementally and iteratively** - Iterative development of working product increments allows incremental verification, which ensures that the partial software product satisfies the technical requirements for that

incremental version; additionally, it allows for the incremental validation (ISO/IEC/IEEE 24765) of the partial product to make certain that it satisfies its intended use, by its intended users, in its intended environment. Incremental verification and validation of working software allows early detection and correction of encountered problems. Waiting to perform integration, verification, and validation of complex systems until later life cycle stages, when these activities are on the critical path to product release, can result in increased cost and schedule impacts. Typically, schedules have minimal slack time during later stages in projects. However, with iterative V&V, software configuration management processes and associated traceability aspects may become complex and require special care to avoid further problems. Chapter 4 of the SWEBOK discusses software testing, and provides numerous references, including standards. Much has been written on the subject; a representative article is (Wallace and Fujii 1989).

18. **Performance trade-offs are different for software than systems** - Systems engineers use “performance” to denote the entire operational envelope of a system; whereas software engineers use “performance” to mean response time and the throughput of software. Consequentially, systems engineers have a larger design space in which to conduct trade studies. In software, performance is typically enhanced by reducing other attributes, such as security or ease of modification. Conversely, enhancing attributes such as security and ease of modification typically impacts performance of software (response time and throughput) in a negative manner.
19. **Risk management for software projects differs in kind from risk management for projects that develop physical artifacts** - Risk management for development of hardware components is often concerned with issues such as supply chain management, material science, and manufacturability. Software and hardware share some similar risk factors: uncertainty in requirements, schedule constraints, infrastructure support, and resource availability. In addition, risk management in software engineering often focuses on issues that

result from communication problems and coordination difficulties within software development teams, across software development teams, and between software developers and other project members (e.g., hardware developers, technical writers, and those who perform independent verification and validation). See (Boehm 1991) for a foundational article on the matter.

20. **Software metrics include product measures and process measures** - The metrics used to measure and report progress of software projects include product measures and process (ISO/IEC/IEEE 24765) measures. Product measures include the amount of software developed (progress), defects discovered (quality), avoidable rework (defect correction), and budgeted resources used (technical budget, memory and execution cycles consumed, etc.). Process measures include the amount of effort expended (because of the people-intensive nature of software development), productivity (software produced per unit of effort expended), production rate (software produced per unit time), milestones achieved and missed (schedule progress), and budgeted resources used (financial budget). Software metrics are often measured on each (or, periodically, some) of the iterations of a development project that produces a next working version of the software. Chapter 8 and Chapter 7 of the SWEBOK address this.
21. **Progress on software projects is sometimes inadequately tracked** - In some cases, progress on software projects is not adequately tracked because relevant metrics are not collected and analyzed. A fundamental problem is that accurate tracking of a software project depends on knowing how much software has been developed that is suitable for delivery into the larger system or into a user environment. Evidence of progress in the form of working software is one of the primary advantages of the iterative development of working software increments.

References

Works Cited

Belady, L. and M. Lehman. 1979. "Characteristics of large systems." In P. Wegner (ed.), *Research Directions*

in Software Technology. Cambridge, MA, USA: MIT Press.

Boehm, B. 1991. "Software risk management: Principles and practices." *IEEE Software*. 8(1):32-41.

Boehm, B. and V. Basili. 2001. "Software defect reduction Top 10 List." *Computer*. 34(1):135-137.

Brooks, F. 1995. *The Mythical Man-Month, Anniversary Edition*. Boston, MA, USA: Addison Wesley Longman Inc.

Bullock, J. 2000. "Calculating the value of testing." *Software Testing and Quality Engineering*, May-June, 56-62.

DeMarco, T. and T. Lister. 1987. *Peopleware: Predictive Projects and Teams*. New York, NY, USA: Dorset House.

Elssamadisy, A. and G. Schalliol. 2002. "Recognizing and responding to 'bad smells' in extreme programming." *Proceedings, ICSE 2002*, ACM-IEEE, 617-622.

Fairley, R.E. and M.J. Willshire. 2011. "Teaching software engineering to undergraduate systems engineering students." *Proceedings of the 2011 American Society for Engineering Education (ASEE) Annual Conference and Exposition*. 26-29 June 2011. Vancouver, BC, Canada.

Kitchenham, B. 1997. "Estimates, uncertainty, and risk." *IEEE Software*. 14(3): 69-74.

Larman, C. and V.R. Basili. 2003. "Iterative and incremental developments: A brief history." *Computer*. 36(6): 47-56.

Lederer, A.L. and J. Prasad. 1995. "Causes of inaccurate software development cost estimates." *Journal of Systems and Software*. 31(2):125-134.

Magazinius, A. 2012. *Exploring Software Cost Estimation Inaccuracy*. Doctoral Dissertation. Chalmers University of Technology. Goteborg, SE.

Martin, R.C. 2002. *Agile Software Development: Principles, Patterns and Practices*. Upper Saddle River, NJ, USA: Prentice Hall.

Moore, G.E. 1965. "Cramming more components onto integrated circuits," *Electronics Magazine*, April 19, 4.

Patterson, D. 2010. "The trouble with multicore." *IEEE Spectrum*, July, 28-32, 52-53.

Royce, W.W. 1970. "Managing the development of large software systems." *Proceedings of IEEE WESCON*. August 1970.

Wallace, D.R. and R.U. Fujii. 1989. "Software verification and validation: An overview." *IEEE Software*. 6(3):10-17.

Zuse, Horst. 1991. *Software complexity: Measures and methods*. Hawthorne, NJ, USA: Walter de Gruyter and Co.

Primary References

Bourque, P. and R.E. Fairley (eds.). 2014. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. Los Alamitos, CA, USA: IEEE Computer Society. Available at: http://www.Swebok.org.

Brooks, Fred 1995. The Mythical Man-Month, Anniversary Edition. Reading, Massachusetts: Addison Wesley.

Fairley, R.E. 2009. *Managing and Leading Software Projects.* Hoboken, NJ, USA: John Wiley & Sons.

PMI. 2013A. *A Guide to the Project Management Body of Knowledge|(PMBOK® Guide)*. 5th ed. Newtown Square, PA, USA: Project Management Institute (PMI).

Additional References

PMI 2013B. *Software Extension to the PMBOK® Guide*, Fifth Edition, Newtown Square, PA, USA: Project Management Institute (PMI) and Los Alamitos, CA, USA: IEEE Computer Society.

Pyster, A., M. Ardis, D. Frailey, D. Olwell, A. Squires. 2010. "Global workforce development projects in software engineering." *Crosstalk - The Journal of Defense Software Engineering*, Nov/Dec, 36-41. Available at: http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA535633 Accessed 02 Dec 2015.

< Previous Article | Parent Article | Next Article >
SEBoK v. 2.9, released 20 November 2023

04"