Intersecting Definitions of V&V

Mark C. Paulk
Advanced Research Center on Software Testing and Quality Assurance
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75080
Mark.Paulk@utdallas.edu

***Abstract*—Verification and validation are two terms with three overlapping definitions in the closely related fields of systems and software engineering: 1) fulfill the conditions imposed by the previous phase; 2) reflect the requirements; and 3) fulfill the intended use. All three concepts are important and address distinct needs. One can argue that the first, the traditional software engineering definition of verification, has been largely superseded by agile methods.**

***Keywords*— verification, validation**

When the Capability Maturity Model Integration for Development (CMMI-DEV) was being developed, I noticed something odd about the Verification and Validation process areas. Coming out of a software engineering background, I was familiar with the definitions from IEEE 610 [1]:

- **verification.** (1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
- **validation**. The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.
- **verification and validation (V&V).** The process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements.

For verification, "conditions imposed at the start of that phase" implied, for example, that the code implemented the design. The Capability Maturity Model for Software (Software CMM) [2], which was one of the source documents for CMMI, used the IEEE 610 definitions. In CMMI-DEV [3], however, the purposes of the process areas and the definitions of the terms are:

1. The purpose of **Validation** (VAL) is to demonstrate that a product or product component fulfills its intended use when placed in its intended environment.
   - **Validation**. Confirmation that the product or service, as provided (or as it will be provided), will fulfill its intended use. In other words, validation ensures that you built the right thing.
2. The purpose of **Verification** (VER) is to ensure that selected work products meet their specified requirements.
   - **Verification**. Confirmation that work products properly reflect the requirements specified for them. In other words, verification ensures that you built it right.

These definitions were consistent with those used in the Systems Engineering CMM [4] and EIA 731 [5] but differed, perhaps significantly, from those used in software engineering. They intersected, albeit with different terms for the same concept, but there were really three concepts in systems and software engineering captured by these two terms.

Concept	Software Engineering	Systems Engineering
fulfill the conditions imposed by the previous phase	verification	--
reflect the requirements	validation	verification
fulfill the intended use	--	validation

One might consider these differences to reflect the different foci of systems engineering and software engineering. Traditionally systems engineering, at least for hardware/software systems, has had the lead on interfacing with the customer and users. Such an interpretation of responsibilities leaves a gap for software-only systems, which also need systems thinking even if systems engineering may not have a formal role.

A challenge for any project is the gap between what the user needs and the requirements that supposedly express those needs. The systems engineering sense of validation attacks the problem of missing and misunderstood requirements.

One might argue that a system's fulfilling its intended use in its operational environment is primarily a requirements elicitation problem – what is the role of "validation" against such an amorphous target? In requirements inspections, however, we expect the developers to ask probing questions into whether a correct and complete set of requirements has been elicited, using checklists and scenarios to confirm that the stakeholders have articulated their needs and desires effectively. Teams using agile methods, for example, maintain a dialog with the customer and users throughout the development process to ensure that the project gets timely feedback on how accurately the system is addressing the customer's (evolving) needs.

The mechanisms we use are not necessarily constrained by our jargon. The systems engineering sense of validation could be addressed by requirements inspections; design inspections could address software engineering validation and systems engineering verification; code inspections could address systems and software engineering verification. One can easily conclude that all three concepts are important and should be addressed, even if the terminology may be context dependent. Unfortunately many of the organizations that adopt inspections focus on code inspection rather than requirements and design inspection, where the greatest benefit accrues.

Perhaps, however, software engineering is evolving in the direction of the systems engineering perspective. Consider the growth of agile methods over the last twenty years. In the agile world, design documentation is de-emphasized. Design is emphasized as a verb rather than a noun. As iterative and incremental development move to very short increments where requirements analysis, design, code, and test are performed in a compressed manner, does it make sense to worry about IEEE 610's 1990 view of verification?

As a side note, I agree with agile's de-emphasis on detailed design documentation because my own experience with detailed design documents is that they quickly move out of synch with the code as the requirements change. Even maintaining an up-to-date requirements document can be a challenge. Detailed design documents are rarely useful in maintenance because they don't reflect what the code is actually doing. I am, however, an advocate for architecture as useful top-level design documentation that can and should be maintained as a system evolves.

If one takes an agile perspective, then fulfilling the conditions imposed by the previous phase becomes a meaningless concept. The "previous phase" is compressed down to an activity in a short iteration. Maintaining traceability from requirements, perhaps in the form of user stories, to code and then test cases remains a worthwhile exercise, but arguably the systems engineering sense of V&V should dominate.

Not all projects, however, are good candidates for agile methods. In life critical or high reliability systems, the need for thoughtful analysis of requirements changes implies that the project cannot be agile in the sense of responding rapidly to changing requirements even if it has adopted agile practices elsewise. The individual practices of the agile world derive for the most part from good software engineering and management practices that have been cranked up to an extreme implementation, so a project could in principle adopt an agile methodology even if it is not "agile" in the sense of responding rapidly to change. Projects in a high reliability environment are likely to use a plan-driven approach along with good practices inspired by their agile counterparts.

If we accept that many projects will continue to use a plan-driven style, does the IEEE 610 kind of verification continue to add value? Arguments can be made on both sides of this question; my point is that we should understand the terms that we use and impose on others. If V&V evolves to a more useful set of terms, that should be a conscious decision on the part of those defining and using those terms.

In discussing this point with the authors of various standards and models, I have so far found no one who was concerned about these intersecting definitions. If a project applies peer reviews (in particular inspections) throughout the life cycle, then all three senses of V&V can be addressed via a single mechanism, rendering the distinction moot.

CMMI-DEV characterizes verification as an inherently incremental process occurring throughout product development. By implication, peer reviews will take into consideration the inputs to each engineering process, e.g., design as an input to code, in their search for defects. Thus the model that originally sparked this question implicitly addresses the intersecting definitions.

In recent years standards writing bodies have tended to follow the path laid out by CMMI; entities originally focused on software engineering standards have evolved to address both systems and software engineering. The current IEEE glossary, for example, is IEEE 24765 [6], which addresses the vocabulary of both software and systems engineering. It includes five definitions of validation from various standards, including one from ISO 12207 (Software Life Cycle Processes) [7]: validation is, in a life cycle context, the set of activities ensuring and gaining confidence that a system is able to accomplish its intended use, goals and objectives.

Thus we see that software and systems engineering have become thoroughly intertwined. Once useful distinctions have become blurred and/or evolved. We should not, however, allow jargon to cloud our consideration of these three intersecting concepts. It is our responsibility as professionals to consider whether and how each of these three should be appropriately addressed on our projects. This can become a particularly pointed question if our organizations are undergoing appraisals against CMMI-DEV, assessments against ISO 15504, or other kinds of “certification”. Understanding the relevant definitions of V&V can prevent unwelcome surprises.

References

[1] “IEEE Standard Glossary of Software Engineering Terminology,” IEEE 610.12, September 1990.

[2] M.C. Paulk, C.V. Weber, B. Curtis, and M.B. Chrissis, The Capability Maturity Model: Guidelines for Improving the Software Process, Addison-Wesley, Boston, 1995.

[3] M.B. Chrissis, M.D. Konrad, and S. Shrum, CMMI for Development: Guidelines for Process Integration and Product Improvement, Third Edition, Addison-Wesley, Upper Saddle River, NJ, 2011.

[4] R. Bate, D. Kuhn, C. Wells, J. Armitage, G. Clark, K. Cusick, S. Garcia, M. Hanna, R. Jones, P. Malpass, I. Minnich, H. Pierson, T. Powell, and A. Reichner, “A Systems Engineering Capability Maturity Model, Version 1.1,” Carnegie Mellon University, Software Engineering Institute, CMU/SEI-95-MM-003, November 1995.

[5] “Systems Engineering Capability Model,” Electronic Industries Alliance, EIA 731.1, August 2002.

[6] "Systems and Software Engineering – Vocabulary,” ISO/IEC/IEEE 24765, December 2010.

[7] “Systems and Software Engineering - Software Life Cycle Processes,” ISO/IEC/IEEE 12207, January 2008.

Author Biography

Dr. Mark Paulk teaches software engineering and is a member of the Software Testing and Quality Assurance Center in Computer Science Department at the University of Texas at Dallas. He was a Senior Systems Scientist at the Institute for Software Research at Carnegie Mellon University from 2002 to 2012. From 1987 to 2002, Dr. Paulk was with the Software Engineering Institute at Carnegie Mellon, where he led the work on the Capability Maturity Model for Software. He was co-project editor of ISO/IEC 15504-2 (Software Process Assessment: Baseline Practices Guide) and a contributor to ISO and IEEE standards. He is a co-author of the eSourcing Capability Model for Service Providers. Dr. Paulk received his PhD in industrial engineering from the University of Pittsburgh. He is a Fellow of the ASQ and a Senior Member of the IEEE.