V&V LIFECYCLE METHODOLOGIES

by David F. Rico

ABSTRACT

This paper introduces a new notion called Verification & Validation (V&V) Lifecycle Methodologies, examining what V&V is, and expanding its scope to the entire software lifecycle (much beyond traditional methods of software testing). V&V Lifecycle Methodologies are streamlined and efficient, yet highly holistic and concisely measured step-by-step threaded and sequenced approaches to predictively and deterministically building-in software quality as it is developed across the entire software lifecycle. V&V is recast in a much more holistic definition and approach, based on a rapidly maturing V&V discipline, solid empirical data, and recent innovations in metrics-based software quality and reliability management and engineering.

WHAT IS V&V?

According to the IEEE Standard Glossary of Software Engineering Terminology [1], V&V is defined as the process of determining whether:

- Requirements for a system or component are complete and correct.
- Products of each development phase fulfill the requirements or conditions imposed by the previous phase.
- Final systems or components comply with specified requirements.

What this means is that if deliverable software products, usually non-deliverable by-products of the software process, and final integrated systems all satisfy their specified requirements, successful V&V has occurred. A simpler way of saying the same thing, is that V&V is the process of successfully eliminating software defects from all software lifecycle artifacts on a stage-by-stage basis, before testing begins

MYTHS & MISCONCEPTIONS

Unfortunately, there are many different perceptions of what V&V is, and new definitions are being created across the world all of the time. The author has been retained to derive new definitions of V&V for international standardization despite the plethora of existing definitions. But, the challenge of successfully defining, managing, and using V&V doesn't end there, as many myths and misconceptions abound about what V&V is (making it one of the most difficult and confusing software disciplines to be a part of):

- <u>V&V, quality, and testing are often equated</u>: V&V is the process of eliminating defects from software artifacts across the software lifecycle. Quality is a process of ensuring that all specified software processes have been properly executed. Testing is a process of dynamically executing software after it is complete. Many practitioners equate these three forms of processes, designate them as a single labor category, and often reduce V&V and quality to the mere act of testing.
- <u>Quality and testing are often equated</u>: Once again, quality processes and labor categories are often reduced to testing.
- <u>Testing and V&V are often equated</u>: V&V processes and labor categories are also often reduced to testing.
- <u>Testing is believed to be sufficient</u>: Even when V&V and quality processes and roles are well understood, testing itself is perceived to sufficiently replace them.
- <u>V&V is often confused with IV&V</u>: The role of V&V is an inherent part of software management and engineering, and even specialized labor categories internal to software lifecycles. IV&V is the process of V&V

carried out by an independent group other than the principal software managers and engineers, for the purpose of ensuring objectivity. IV&V is often designated to be carried out by an independent third party contractor or wholly external organization, also for the purpose of ensuring objectivity. Unfortunately, IV&V is often, but not always, reduced to the process of testing.

- IV&V is believed to be better than V&V Lifecycle Methodologies: It is also popularly held that IV&V is a necessary part of mission critical software development. The notion being that IV&V can successfully be used to objectively eliminate defects, that software managers and engineers are unable to, based on an objective viewpoint.

At the very least, this paper will attempt to define what V&V and testing are, differentiating between them sharply. This paper will minimally alert the reader to the notion that quality, testing, V&V, and IV&V processes are not equal in definition, execution, or effectiveness. And, hopefully, this paper will begin to dispel some of these deeply ingrained myths and misconceptions concerning V&V.

V&V APPROACHES

Three unique V&V approaches will be identified and briefly explained here. The objective is to call out important software industry V&V perspectives. These three V&V approaches will be placed in the proper context, and the most appropriate and attractive form of V&V will be illuminated. Appropriateness and attractiveness will be defined from the perspective of the software under development, and ultimately the customer or user of the final software product. The approaches that will be examined here are:

- Testing (Post Process): Testing is a popular though extremely limited V&V approach that encompasses dynamic execution of hard-coded software.

- Lifecycle Frameworks (In Process): Lifecycle frameworks are V&V approaches that manifest themselves as encyclopedic taxonomies of abundant software defect elimination techniques.

- Lifecycle Methodologies: Lifecycle methodologies are holistic and integrated step-by-step metrics and measurement-based approaches to V&V.

Testing (Post Process)

Testing is a popular, yet extremely limited form of V&V, which encompasses dynamic execution of hard-coded software. It is referred to as post process, because testing generally occurs at the very end of software lifecycles, usually after all resources have been exhausted, and much too late to eliminate the population of latent software defects. Testing is also the most ineffective and expensive of the three approaches discussed in this paper. Testing is considered less effective because it requires an order of magnitude more resources than lifecycle methodologies, and would take too long to eliminate the population of latent software defects.

Testing Techniques

Testing is broadly classified and divided into two categories, white box testing and black box testing. White box testing is internal or structural testing designed to examine and even dynamically exercise hard-coded software, to include these popular testing techniques:

- Flow Graph Notation (Basis Path).
- Cyclomatic Complexity (Basis Path).
- Deriving Test Cases (Basis Path).
- Graph Matrices (Basis Path).
- Condition (Control Structure).
- Dataflow (Control Structure).
- Loop (Control Structure).

Black box testing is external testing designed to analyze and exercise hard-coded software without examining its internal structure:

- Graph Based Methods (Specification).
- Equivalence Partitioning (Specification).
- Boundary Value Analysis (Specification).
- Comparison (Specification).
- Interface Misuse (Interface).
- Interface Misunderstanding (Interface).
- Timing (Interface).
- Serialization (Interface).
- Requirements (Operational).
- Scenario (Operational).
- Use (Operational).

An explanation of each of these eighteen white and black box testing techniques may be found in software engineering overview texts [2, 3]. Testing techniques are beyond the scope of this paper, as this paper is designed to illuminate life cycle methodologies, not testing.

Testing Documents

The IEEE Standard for Software Test Documentation [4] describes a set of standard test documents. The purpose of this IEEE standard is to act as a test document framework, in order to introduce structure into the test process. IEEE recognizes that adding structure through framework standards is a good method to act as a basis for creating rational software testing processes. The testing techniques previously described may be captured in these documents:

- Test Plan.
- Test Design.
- Test Case.
- Test Procedure.
- Test Item Transmittal.
- Test Log.
- Test Summary.

Testing Levels

Software can be constructed and viewed as a structured hierarchy of ever increasing granularity [3]. Software may then be tested by starting with the lowest level of granularity, proceeding up through software system and acceptance testing:

- Unit Testing.
- Module Testing.
- Component Testing.
- Subsystem Testing.
- Integration Testing.
- System Testing.
- Acceptance Testing.

Once again, testing levels have been structured in an attempt to identify, define, and promote organized, documented, and rational testing. Structured testing processes are believed to increase the likelihood that software defects will be eliminated.

What does Testing Do?

According to the IEEE Standard Glossary of Software Engineering Terminology [1] testing may be defined in these increasing levels of granularity:

- Testing is the process of determining whether final systems or components comply with specified requirements.
- Testing is an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.

For most software projects, software isn't produced until the end of software lifecycles. Therefore testing can't begin until the end of software lifecycles. By this time all software defects have been committed. It would take more time to eliminate all of the software defects during testing than it would to finish all of the software processes before testing begins.

Most organizations don't construct and execute the structured testing techniques, documents, and levels described in the previous section. Many software projects don't perform testing at all. And, the ones that do testing, usually resort to operational testing until software executes reasonable well, ignoring most of the latent software defect population. Defect laden software is then passed directly to customers and users.

Exhaustive software testing standards, frameworks, and techniques, tend to reduce the interest in highly structured testing, not increase it. The standards and frameworks promote the notion that a robust variety of testing techniques and structures will increase the likelihood that software defects will be eliminated. However, for software projects that don't already do ad hoc testing very well, highly structured testing is not very high on their agenda of priorities either. While this attitude tends to be changing rapidly, testing will never be a good method for V&V.

There is some reprieve for testing enthusiasts, as streamlined testing methodologies have been created [5]. These testing methodologies cut through the bureaucracy of encyclopedic testing standards and frameworks, identifying only the most essential and effective testing techniques. According to these testing methodologies, boundary analysis is an excellent technique for eliminating software defects. Automated test case generation for boundary analysis is now possible in many instances. However, many of these techniques would be better served by performing boundary analysis on highly structured analytical models, before software is hard-coded, not after. Furthermore, these techniques are still not the preferable V&V method. Testing still happens much too late in software lifecycles, is not holistic and in process, and would result in the necessity to eliminate inordinately large latent defect populations.

Lifecycle Frameworks (In Process)

Lifecycle frameworks are a much more progressive form of V&V than testing [6]. Lifecycle frameworks recognize the existence and potential selection and use of multiple techniques for eliminating software defects at each software lifecycle stage. Lifecycle frameworks are referred to as in process for this reason. Lifecycle frameworks in their attempt to be thorough and holistic have evolved to be encyclopedic taxonomies of every conceivable software defect elimination technique [7]. Lifecycle frameworks are non-methodological in the sense that each software project must tailor a new and unproven lifecycle methodology by picking and choosing from hundreds of vaguely defined and measured techniques.

<u>More Lifecycle Framework Tasks</u>

The IEEE Standard for Software Verification and Validation Plans [6] identifies thirty-one optional or additional V&V techniques that are not considered mandatory. The objective of these additional V&V techniques is to reinforce the encyclopedic nature of lifecycle frameworks, just in case something is missed. The philosophy is that more is better, and less is bad. This paper will examine an innovative notion that less is more than adequate, when it comes to V&V, and that more is unnecessary and inundating overkill. It's not necessary to enumerate the additional thirty-one V&V techniques here.

<u>What do Lifecycle Frameworks Do?</u>

According to the IEEE Standard Glossary of Software Engineering Terminology [1] V&V may be selectively defined as:

- V&V is the process of determining whether products of each development phase fulfill the requirements or conditions imposed by the previous phase.

IEEE claims that V&V frameworks provide uniform and minimum requirements for V&V. IEEE goes on to say that V&V frameworks provide for a comprehensive evaluation throughout each phase of software projects to help ensure that:

- Errors are detected and corrected as early as possible in the software life cycle.
- Project risk, cost, and schedule effects are lessened.
- Software quality and reliability are enhanced.
- Management visibility into the software process is improved.
- Proposed changes and their consequences can be quickly assessed.

Lifecycle frameworks are definitely a step in the right direction, extending V&V out of testing, and ushering it right into the beginning of software lifecycles. However, like exhaustive testing frameworks, encyclopedic lifecycle frameworks are perceived as overly bureaucratic, unnecessary, and unaffordable. If most software projects won't employ highly structured testing, they certainly will not consider even more extensive lifecycle frameworks. The next few sections will introduce streamlined and more effective approaches to inundating lifecycle frameworks, as well as late and ineffective testing.

Lifecycle Methodologies (In Process)

Lifecycle methodologies, like lifecycle frameworks, address V&V across all software lifecycle stages, eliminating defects from software artifacts [8]. That's where the similarities end. Lifecycle methodologies are streamlined to include only the most effective and bare minimum software defect elimination techniques recommended by lifecycle frameworks. Lifecycle methodologies go on to add holistic and integrated step-by-step metrics and measurement approaches, thus earning the term *methodology* [9]. Lifecycle methodologies also add predictive statistical parametric models, for accurately estimating software lifecycle resources, software defects, and software defect elimination effectiveness [10]. Lifecycle methodologies are much more desirable than lifecycle frameworks, and are literally, orders of magnitude more desirable than testing. Lifecycle methodologies enable quality to focus on ensuring that processes are followed (not being reduced to testing), and are more effective and cost-efficient than lifecycle frameworks, testing, and even IV&V.

<u>Basis for Lifecycle Methodology</u>

The quantitative basis for lifecycle methodologies is predictive statistical parametric Rayleigh models [11]. Rayleigh models are a special case of the Weibull distribution probability density curve or function [9, 10, 11]. Rayleigh models are used to predict defect populations of the total set of software life cycle artifacts before software lifecycles begin.

If true, which ample quantitative empirical results display [11], then lifecycle methodologies enable what no other reliability model claims to do. That is, accurately predict the number of software defects, and allocate the exact number of resources necessary to eliminate software defects, on a schedule, no less. In addition accurately portrayed defect populations, statistically correlated to other management indicators, such as cost, effort, and schedule, serve as the basis for accurately determining required software project resources, before software lifecycles begin [9, 11].

The news keeps on getting better, experience with lifecycle methodologies indicates that, not only can software defects be largely eliminated before testing begins, but that dependency on late and ineffective testing can be minimized [8, 9, 10, 11]. Contrary to the opinion of some practitioners, Rayleigh models can be successfully managed to become asymptotic before software lifecycles end [8, 9, 10, 11]. That is software defects may be completely eliminated before software is actually delivered to customers or users [9, 11].

However, there still must be an underlying process for eliminating and counting defects, in order to manage software lifecycles using lifecycle methodologies. These processes will be described in the next two sections.

Software Inspection Process

Inspections are a highly structured and measured, early lifecycle, in process multiple person review of software artifacts for the exclusive purpose of identifying software defects for elimination [11, 12, 13, 14, 15]. Inspections were invented by Michael E. Fagan of IBM in 1972, and first made public in a seminal paper in a 1976 issue of the IBM Systems Journal [15]. Inspections may be used to identify defects in all software lifecycle artifacts throughout software lifecycles. Therefore, Inspections are the foundational process for lifecycle methodologies, especially those exploiting the predictive and management enabling capabilities of Rayleigh models. Ironically, Inspections are but one of hundreds of techniques recommended by lifecycle frameworks, and are more effective and efficient than many, if not most lifecycle framework V&V techniques. And, Inspections are an order of magnitude more effective and efficient than testing, minimizing the need for testing-intensive V&V.

Inspection Process Measurability

Inspections are a highly structured, well defined, step-by-step process for identifying defects for elimination. Because Inspections can be so concisely characterized and described, all aspects of Inspections can be measured, including total effort required, as well as the efficiencies of every aspect of Inspections. On average, a single Inspection may take 15 hours and uncover as many as 15 to 45 defects in software lifecycle artifacts. Whereas, testing may require more than 45 hours to identify the same number defects in hard-coded software alone. Edward Weller [12], Glenn Russell [13], and Robert Grady [16] have given the earliest expositions of Inspection econometrics and models, but other research available on the World-Wide-Web has offered explicit econometric Inspection models [17].

Lifecycle Methodology Accuracy

As previously mentioned, lifecycle methodologies are based on the use of Rayleigh models. Rayleigh models are quantitative techniques for accurately predicting final software product quality within a tenth of a defect, expressed in defect density metrics and measures, once they have been calibrated to empirical data [9, 11].

Rayleigh model-based lifecycle methodologies have been successfully applied on large-scale monolithic software lifecycles, most notably between 1986 and 1996 [8, 9, 10, 11]. Rayleigh model-based lifecycle methodologies resulted in the development, conversion, and maintenance of [8, 9, 10, 11]:

- Eight system-level software products.
- Five compilers.
- Five major system utilities.
- Eleven million lines of online help information.
- Thirty-two thousand pages of manuals.
- A five hundred thousand line automated help utility.
- Eleven hundred lines of questions and answers.

- Native support for twenty-five international languages.
- Five and a half million new sources lines of code.
- Thirty billion converted source lines of code.

But this wasn't the only program, as Rayleigh model-based V&V lifecycle methodologies have been applied on many different software lifecycles. These software lifecycles ranged from seventy to seventeen hundred new source line of code systems, all managed to within a tenth of a defect of accuracy by the time software was delivered, extending into field use and measurement [11]. To deny the existence, applicability, usefulness, and utility of Rayleigh-based lifecycle methodologies would be a matter of personal choice, but certainly not invalidity.

What do Lifecycle Methodologies Do?

Lifecycle methodologies are compliant with the generic definition of V&V by the IEEE Standard of Glossary of Software Engineering Terminology, which states that V&V is the process of determining whether:

- Requirements for a system or component are complete and correct.
- Products of each development phase fulfill the requirements or conditions imposed by the previous phase.
- Final systems or components comply with specified requirements.

However, lifecycle methodologies are both more and less than lifecycle frameworks, especially those recommended by IEEE standards [6, 7]. Lifecycle methodologies are *more*, in the sense that they are predictive, measurable, and highly effective. Lifecycles are *less*, in the sense that they are lean, streamlined, and reduced to only the most essential, but highly–effective V&V techniques. Popular literature fails to even begin to recognize the existence and applicability of lifecycle methodologies [2, 3]. However, there is a new wave of literature expounding the benefits of even more streamlined and efficient forms of lifecycle methodologies emerging [18, 19, 20, 21, 22, 23].

COSTS & BENEFITS

The most thorough examinations of the costs and benefits of lifecycle methodologies were done by Stephen H. Kan [11], Robert Grady [16], and even some recent work [17, 23]. Take for example, an estimated defect population for ten thousand source lines of code expressed in defect density metrics. One thousand defects would not be uncommon. Watts Humphrey has found the estimated defect population for this size of program to be as much as fifteen to one hundred and fifty percent in many cases [19]. It would take approximately seven hundred and eight Inspection hours to eliminate as much as nine hundred defects in a well run Inspection, while yielding 1.27 defects per hour. Ninety more defects could be found by testing using over eleven hundred hours, for a total cost of eighteen hundred and fifty-two V&V hours using lifecycle methodologies.

Testing alone would require over eleven thousand hours to eliminate almost nine hundred defects from the same population, in an extremely well run testing process. Testing would result in the delivery of over one hundred defects, should an unwary software manager commit to performing over eleven thousand hours of testing.

Contemporary lifecycle methodologies would only require four hundred hours to eliminate all of the defects from the same population [17]. This effort would include development time as well. This would be an improvement of seventy eight percent over traditional lifecycle methodologies, and an improvement of ninety seven percent over testing. The empirical results are overwhelming and undeniable.

Costs of Methodologies

Assuming that Rayleigh theory is correct, it would not be advisable to use testing-based V&V methodologies. Rayleigh models predict that an overwhelming population of defects exists in all software lifecycle artifacts by the time testing begins, if lifecycle methodologies are not employed [9, 11, 21, 22]. Using testing alone would overwhelm the testing process, software lifecycle resources, and inevitably result in passing a large defect population onto software customers and users.

Hewlett Packard

Hewlett-Packard reports saving over three hundred and fifty million dollars in software lifecycle costs by the corporate-wide use of Inspections, between 1989 and 1998 [16]. Inspections are the linchpin of lifecycle methodologies [8, 11, 17].

Raytheon

Raytheon plans on using the aid of lifecycle methodologies to save as much as three and a half billion dollars in corporate operating expenses [24]. Raytheon will be highly dependent on the defect elimination properties of lifecycle methodologies described in this paper.

CONCLUSION

At least three conclusive positions can be identified by this paper:

- Testing is inefficient and happens too late in the lifecycle: Testing is at least ten times less efficient than Inspection-based life cycle methodologies [12, 13]. In addition, testing begins far too late to begin addressing the latent defect populations predicted by Rayleigh models [9, 11]. Most contemporary software managers would not concede to performing minimal testing, much less carry out a program of highly structured testing (based on the author's experience).
- Lifecycle frameworks are inundating, non-methodological, and not easily understood: Lifecycle frameworks are encyclopedic and taxonomic in nature [6, 7]. Lifecycle frameworks are non-methodological, requiring them to be tailored from a catalog of techniques [6]. Lifecycle frameworks lack metrics and measurement-intensive processes that Inspections have [6, 7]. There is not one published or publicly available case study on the quantitative benefits of lifecycle frameworks (based on the author's experience).
- Lifecycle methodologies are fast, efficient, measurable, and accurate: The costs and benefits of lifecycle methodologies are some of the most measurement intensive industrial examples of software lifecycles [11, 12, 17, 21, 23]. The inefficiencies of testing-based V&V approaches are extremely well documented [12, 13]. Lifecycle frameworks have been obsoleted by upstart lifecycle methodologies (based on the author's experience).

BIBLIOGRAPHY

[1] "IEEE Standard Glossary of Software Engineering Terminology," IEEE Standard 610.12-1990.

[2] "Software Engineering: A Practitioner's Approach (Fourth Edition)," Roger S. Pressman, McGraw-Hill, 1997.

[3] "Software Engineering (Fifth Edition)," Ian Sommerville, Addison-Wesley, 1997.

[4] "IEEE Standard for Software Test Documentation," ANSI/IEEE Standard 829-1983 (Reaff 1991).

[5] "Using Models for Test Generation and Analysis," Mark R. Blackburn, Proceedings of the IEEE Digital Avionics Systems Conference, 1998

[6] "IEEE Standard for Software Verification and Validation Plans," IEEE Standard 1012-1986.

[7] "IEEE Guide for Software Verification and Validation Plans," IEEE Standard 1059-1993.

[8] "A New Development Rhythm for AS/400 Software," Richard A. Sulack, et al, IBM Systems Journal, Volume 28, Number 3, 1989.

[9] "Modeling and Software Development Quality," Stephen H. Kan, IBM Systems Journal, Volume 30, Number 3, 1991.

[10] "AS/400 Software Quality Management," Stephen H. Kan, et al., IBM Systems Journal, Volume 33, Number 1, 1994.

[11] "Metrics and Models in Software Quality Engineering," Stephen H. Kan, Addison-Wesley, 1995.

[12] "Lessons Learned from Three Years of Inspection Data," Edward F. Weller, IEEE Software, September 1993.

[13] "Experience with Inspection in Ultralarge-Scale Developments," Glenn W. Russell, IEEE Software, January 1991.

[14] "Advances in Software Inspections," Michael E. Fagan, IEEE Transactions on Software Engineering, Volume SE-12, Number 7, July 1986.

[15] "Design and Code Inspections to Reduce Errors in Program Development," Michael E. Fagan, IBM Systems Journal, Volume 15, Number 3, 1976.

[16] "Successful Software Process Improvement," Robert B. Grady, Prentice-Hall, 1997.

[17] "Software Process Improvement: Impacting the Bottom Line by Using Powerful Solutions," David F. Rico, Software Process Improvement (SPI) Webpage, http://members.wbs.net/homepages/d/a/v/david frico/spibrief.pdf, April 1998.

[18] A Discipline for Software Engineering," Watts S. Humphrey, Addison-Wesley, 1995.

[19] "Using a Defined and Measured Personal Software Process," Watts S. Humphrey, IEEE Software, May 1996.

[20] "Introduction to the Personal Software Process," Watts S. Humphrey, Addison-Wesley, 1996.

[21] "Results of Applying the Personal Software Process," Pat Ferguson, IEEE Computer, May 1997.

[22] "The Personal Software Process: An Empirical Study of the Impact of PSP on Individual Engineers," Will Hayes, CMU/SEI-97-TR-001.

[23] "Personal Software Process (PSP)," David F. Rico, Software Process Improvement (SPI) Webpage, http://members.wbs.net/homepages/d/a/v/davidfrico/psp.pdf, December 1998.

[24] "High Hopes Riding on Six Sigma at Raytheon," Anthony L. Velocci, Jr., Aviation Week & Space Technology, November 16, 1998.