S E I / C M M P r o p o s e d

Software Evaluation and Test KPA

SEI/CMM Proposed Software Evaluation and Test KPA

By Richard Bender,
Bender RBT Inc.
17 Cardinale Lane
Queensbury, NY 12804
518-743-8755
rbender@BenderRBT.com

KPA REVIEW GROUP

The following have been gracious enough to be reviewers of this proposed Testing KPA. I want to thank them for their insights and contributions. However, any problems or omissions the reader may find with this document I take full responsibility for. This is very much a work in progress. Please feel free to contact me with suggestions for improving it.

Boris Beizer - Independent testing consultant
Greg Daich - STSC
Dave Gelperin - SQE
Bill Hetzel - SQE
Capers Jones - SPR
John Musa - ATT
William Perry - QAI
Robert Poston - IDE

The original version of the Evaluation and Testing KPA was sponsored by Xerox Corporation. They have graciously allowed us to distribute it to the software community. The key contact is:

David Egerton
800 Phillips Road
Building 129
Webster, NY 14580
(716) 422-8822

August 2001
(Revision #6)

TABLE OF CONTENTS

1. Introduction

The objective of this document is to present a proposal that Evaluation and Test become a Key Process Area (KPA) in the SEI Capability Maturity Model (CMM). The first section addresses the scope of what is meant by evaluation and test. The second section identifies the justifications for making this a separate KPA. The third section presents the proposed KPA definition including: definition, goals, commitment to perform, activities performed, measurements and analysis, and verifying implementation. The final section addresses integrating this KPA with the existing KPAs. This includes identifying which level to assign it to and some repackaging suggestions for existing KPAs.

2. Defining Evaluation and Test

Evaluation is the activity of verifying the various system specifications and models produced during the software development process. Testing is the machine based activity of executing and validating tests against the code. Most software organizations define evaluation and test very narrowly. They use it to refer to just the activities of executing physical test cases against the code. In fact, many companies do not even assign testers to a project until coding is well under way. They further narrow the scope of this activity to just function testing and maybe performance testing.

This view is underscored in the description of evaluation and test in the current CMM. It is part of the Software Product Engineering KPA. The activities in this KPA, activities 5, 6, and 7, only use code based testing for examples and only explicitly mention function testing. Other types of testing are euphemistically referenced by the phrase "...ensure the software satisfies the software requirements."

People who build skyscrapers, on the other hand, thoroughly integrate evaluation and test into the development process long before the first brick is laid. Evaluations are done via models to verify such things as stability, water pressure, lighting layouts, power requirements, etc. The software evaluation and test approach used by many organizations is equivalent to an architect waiting until a building is built before testing it and then only testing it to ensure that the plumbing and lighting work.

The CMM further compounds the limited view of evaluation and test by making a particular evaluation technique, peer reviews, its own KPA. This implies that prior to the delivery of code the only evaluation going on is via peer reviews and that this is sufficient. The steps in the evaluation and test of something are: define the completion/success criteria, design cases to cover this criteria, build the cases, perform/execute the cases, verify the results, and verify that everything has been covered. Peer reviews provide a means of *executing* a paper based test. They do not inherently provide the success criteria nor do they provide any formal means for defining the cases, if any, to be used in the peer review. They are also fundamentally subjective. Therefore, the same misconceptions that lead a programmer to introduce a defect into the product may cause them to miss the defect in the peer review.

A robust scope for evaluation and test must encompass every project deliverable at each phase in the development life cycle. It also address each desired characteristic of each deliverable. It must address each of the evaluation/testing steps. Let's look at two examples: evaluating requirements and evaluating a design.

A requirements document should be complete, consistent, correct, and unambiguous. One step is to validate the requirements against the project/product objectives (i.e., the statement of "why" the project is being done). This ensures that the right set of functions are being defined. Another evaluation is to walk use-case scenarios through the functional rules, preferably aided by screen prototypes if appropriate. A third evaluation is a peer review of the document by domain experts. A fourth is to do a formal ambiguity review by non-domain experts. (They cannot read into the document assumed functional knowledge. It helps ensure that the rules are defined explicitly, not implicitly.) A fifth evaluation is to translate the requirements into a Boolean graph. This identifies issues concerning the precedence relationships between the rules as well as missing cases. A sixth is a logical consistency check with the aid of CASE tools. A seventh is the review, by domain experts, of the test scripts derived from the requirements. This "bite-size" review of the rules often uncovers functional defects missed in reviewing the requirements as a whole.

Evaluating a design can also take a number of tacks. One is walking tests derived from the requirements through the design documents. Another is building a model to verify design integrity (e.g., a model built of the resource allocation scheme for an operating system to ensure that deadlock never occurs). A third is building a model to verify performance characteristics. A fourth is comparing the proposed design against existing systems at other companies to ensure that the expected transaction volumes and data volumes can be handled via the configuration proposed in the design.

Only some of the above evaluations were executed via peer reviews. None of the above were code based. Neither of the above examples of evaluation was exhaustive. There are other evaluations of requirements and designs that can be applied as necessary. The key point is that a deliverable has been produced (e.g., a requirements document); before we can say it is now complete and ready for use in the next development step we need to evaluate it for the desired/expected characteristics. Doing this requires more sophistication than just doing peer reviews.

That is the essence of evaluation and test. A pre-defined set of characteristics, defined as explicitly as possible, is validated against a deliverable. For example, when you were in school and took a math test the instructor compared your answers to the expected answers. The instructor did not just say they look reasonable or they're close enough. The answer was supposed to be 9.87652. Either it was or it was not. Also, the instructor did not wait until the end of the semester to review papers handed in early in the course. They were tested as they were produced. With the stakes so much higher in software development, can we be any less rigorous and timely?

Among the items which should be evaluated and tested are Requirements Specifications, Design Specifications, Data Conversion Specifications and Data Conversion code, Training Specifications and Training Materials, Hardware/Software Installation Specifications, Facilities Installation Specifications, Problem Management Support System Specifications, Product Distribution Support System Specifications, User Manuals, and the application code. Again this is not a complete list. The issue is that every deliverable called for in your project life cycle must be tested.

The evaluation and test of a given deliverable may span multiple phases of the project life cycle. More and more software organizations are moving away from the waterfall model of the life cycle to an iterative approach. For example, a Design Specification might be produced via three iterations. The first iteration defines the architecture - is it manual or automated, is it centralized or distributed, is it on-line or batch, is it flat files or a relational data base, etc. The second iteration might push the design down to identifying all of the modules and the inter-module data path mechanisms. The third iteration might define the intra-module pseudo-code. Each of these iterations would be evaluated for the appropriate characteristics.

The types of evaluation and test must be robust. This includes, but is not limited to, verifying functionality, performance, reliability-availability-serviceability, security, usability, portability, ability to localize (i.e., different languages), maintainability, testability, and extendibility.

In summary, each deliverable at each phase in its development should be evaluated/tested for the appropriate characteristics via formal, disciplined techniques.

3. The Justification for a Separate Evaluation and Test KPA

There are five significant reasons which justify having a separate Evaluation and Test KPA: evaluation and test's role in accelerating the cultural change towards a disciplined software engineering process, the role of evaluation and test in project tracking, the portion of the development and maintenance budget spent on evaluation and test, the impact of evaluation and test disciplines on the time and costs to deliver software, and the impact of residual defects in software.

3.1 Accelerating Cultural Change

Electrical engineers and construction engineers are far more disciplined than software engineers. Electrical engineers produce large scale integrated circuits at near zero defect even though they contain millions of transistors. What is often lost in the widely discussed defect in the Pentium processor is that it was one defect in 3,100,000 transistors. When was the last time you saw software which had only one defect in 3,100,000 lines of code? The hardware engineers do not achieve better results because they are smarter than the software engineers. They achieve quality levels orders of magnitude higher than software because they are more disciplined and rigorous in their development and testing approach. They are willing to invest the time and effort required to ensure the integrity of their products. They recognize the impact that defects have, economic and otherwise.

Construction engineers face similar challenges in constructing sky scrapers. In their world a "system crash" means the building collapsed. In regions of the world which have and enforce strict building codes that just does not happen. Again, this can be traced to the discipline of their development and testing approach.

Software, on the other hand, is a different matter. Gerald Weinberg's statement that "if builders built buildings the way software people build software, the first woodpecker that came along would destroy civilization" is on the mark.

We have to recognize that the software industry is very young as compared to other engineering professions. You might say that it is fifty-five years old, if you start with Grace Hopper as the first programmer. (A bit older if you count Ada Lovelace as the first.) However, a more realistic starting date is about 1960. That is just over forty years. By contrast, the IEEE celebrated their 100th anniversary in 1984. That means that in 1884 there were enough electrical engineers around to form a professional society. In 1945, by contrast, Ms. Hopper would have been very lonely at a gathering of software engineers.

As a further contrast construction engineering goes back over 5,000 years. The initial motivation for creating nations was not self defense; it was the necessity to manage large irrigation construction projects. We even know the names of some of these engineers. For example, in 2650 BC Imhotep is the chief engineer for the step pyramid of Djoser (aka Zoser) in Egypt. In fact he did such a good job they made him a god.

The electrical engineers and construction engineers did not start out with inherently disciplined approaches to their jobs. The discipline evolved over many years. It evolved as they came to understand the need for discipline and the implications of defects in their work products. Unfortunately, we do not have thousands of years or even a hundred years to evolve the software profession. We are already building business critical and safety critical software systems. Failures in this software are causing major business disruptions and even deaths at an alarmingly increasing rate. (See "Risk To The Public" by Peter Neumann.)

Moving the software industry from a craftsman approach to a true engineering level of discipline is a major cultural shift. The objective of the CMM is, first and foremost, a mechanism for inducing this cultural change for software engineers. However, a culture does not change voluntarily unless it understands the necessity for change. It must fully understand the problems being solved by evolving to the new cultural paradigm.[1] This, finally, brings us to the role of testing in accelerating the cultural change to a disciplined approach (I know you were beginning to wonder when I would tie this together).

[1] My degree is in mathematics; however, my minors were archaeology and anthropology. I have always found these far more useful than math in helping organizations install software engineering disciplines and tools.

In the late 1960's, IBM was one of the first major organizations to begin installing formal software engineering techniques. This began with the use of the techniques espoused by Edsger Dijkstra and others. Ironically, it was not the software developers who initiated this effort. It was the software testers. The initial efforts were started in the Poughkeepsie labs under a project called "Design for Testability" headed by Philip Carol.

Phil was a system tester in the Software Test Technology Group. This group was responsible for defining the software testing techniques and tools to be used across the entire corporation. Nearly thirty years ago they began to realize that you could not test quality into the code. You needed to address the analysis, design, and coding processes as well as the testing process. They achieved this insight because as testers they thoroughly understood the problem since testing touches all aspects of software development. Testers inherently look for what is wrong and try to understand why.

It was this understanding of the problem and the ability to articulate the problem to developers that allowed for a rapid change in the culture. As improved development and test techniques and tools were installed, the defect rate in IBM's OS operating system dropped by a factor of ten in just one release. This is a major cultural shift occurring in a very short time, especially given that it involved thousands of developers in multiple locations.

The rapidity of the change was aided by another factor related to testing in addition to the problem recognition. This was the focused feedback loop inherent in integrating the testing process with the development process. As the development process was refined, the evaluation and test process was concurrently refined to reflect the new success criteria. As developers tried new techniques they got immediate feedback from testers as to how well they did because the testers were specifically validating the deliverables against the new yardstick.

A specific example is the installation of improved techniques for writing requirements that are unambiguous, deterministic, logically consistent, complete, and correct. Analysts are taught how to write better requirements in courses on Structured Analysis and courses in Object-Oriented Analysis. If ambiguity reviews are done immediately after they write up their first functional descriptions, the next function they write is much clearer out of the box. The tight feedback loop of write a function, evaluate the function, accelerates their learning curve. Fairly quickly the process moves from defect detection to defect prevention - they are writing clear, unambiguous specifications.

Contrast this to the experience of the software industry as a whole. The structured techniques and the object-oriented techniques have been available for over thirty years (yes, O-O is that old). Yet the state of the practice is far behind the state of the art. The issue is an organization does not fully accept nor understand a solution (e.g., the software engineering tools and techniques) unless it understands the problem being solved. Integrated evaluation and test is the key to problem comprehension. "Integrated evaluation and test" is defined here as integrating testing into every step in the software development process. It is thus the key to the necessary feedback loops required to master a technique. Any process without tight feedback loops is a fatally flawed process. Evaluation and test is then the key to accelerating the cultural change.

3.2 The Role of Evaluation and Test in Project Tracking

A project plan consists of tasks, dependencies, resources, schedules, budgets, and assumptions. Each task should result in a well defined deliverable. That deliverable needs to be verified that it is truly complete. If you do not evaluate/test the task deliverables for completeness you cannot accurately track the true status of the project.

For example, Requirements Specifications always seem to be "done" on schedule. This is because many organizations do not formally evaluate the Requirements Specification. Later in the project they find themselves completing the definition of the requirements during design, coding, testing, and even production. What, therefore, did it really mean to say that the task of writing the requirements was completed?

Incomplete "completed" tasks can also have a ripple effect on the completion status of subsequent tasks. In the above scenario, what is the impact of finding requirements deficiencies during code based testing? The "completed" Requirements Specification must be revised. The "completed" Design Specification must be revised. The "completed" code must be revised. The "completed" User Manuals must be revised. The "completed" Training Materials must be revised. The "completed" test cases must be revised.

The objective of project tracking is to give management and the project team a clear understanding of where the project stands. Without evaluation/testing integrated into every step in the project you can never be sure of what is and is not really completed. Given that Software Project Tracking and Oversight is a KPA and it depends on evaluation and test to perform the tracking, then evaluation and test as a KPA is a necessary preceding activity.

3.3 Evaluation and Test as a Percentage of the Project Costs

A major pragmatic factor in determining what should and should not be a separate KPA is what portion of the software development budget and staff are involved in the activity. The more significant the activity is in these terms the more focus it should receive.

There have been numerous studies documenting how project costs are allocated across the various activities. In these studies just the code based testing accounts for 35% to 50% of the project costs. This is true for both software development and for software maintenance. Factor in the effort to perform evaluations and this number is higher.

Organizations using any level of discipline in their testing have a tester to developer ratio of at least 1:3. More and more software vendors are moving to a 1:1 ratio. At times the NASA Space Shuttle project has had a ratio of 3:1 and even 5:1!

Simply put, any activity which consumes a third to a half of the budget and a fourth to a half of the resources should definitely be addressed by its own KPA.

3.4 Impact of Evaluation and Test on Development Schedules and Project Costs

Numerous studies show that the majority of defects have their root cause in problems with the requirements definition. In one study quoted by James Martin, over 50% of all software defects are caused by incomplete, incorrect, inaccurate, and/or ambiguous requirements. Even more telling is that over 80% of the costs of defects have their roots in requirements based errors.

Other studies show that the earlier you find a defect the cheaper it is to fix. A defect found in production can cost 2,000 times more than the same defect found in an evaluation of the requirements.

The issue is scrap and rework. This is the primary cause of cost and schedule overruns on projects. The plan may have identified the initial set of tasks to be done. However, due to defects found later, “completed” tasks must now be redone. The “re-do” task was not in the original plan. As the number of tasks requiring rework grows, the cost and schedule overruns accumulate. Integrating evaluation and test throughout the project life cycle minimizes scrap and rework, bringing the costs and schedules back under control.

Integrated evaluation and test can further shorten schedules by allowing for more concurrent activities. When Requirements Specifications are not formally evaluated, the design and coding activities often result in numerous changes to the scope and definition of the functions being delivered. For this reason, work does not start on the User Manuals and Training Materials until code based testing is well underway. Until then no one is confident enough in the system definition.

Similarly, poorly defined requirements do not provide sufficient information from which to design test scripts. The design and building of test cases often does not start until coding is well underway.

These two scenarios force the development process to be linear: requirements, then design, then code, then test, then write manuals. If the Requirements Specification is written at a deterministic level of detail (i.e., given a set of inputs and an initial system state you should be able to determine the exact outputs and the new system state by following the rules in the specification), then test case design and the writing of the manuals can go on concurrently with the system design. This in turn shortens the elapsed time required to deliver the system. However, creating deterministic specifications requires formal evaluation of that specification.

In summary, integrated evaluation and test reduces schedules and project costs by minimizing scrap and rework and allowing more activities to be performed concurrently. These types of gains can not be accomplished without integrated evaluation and test. Since time to market and cost to market are key issues for any software organization and testing is the key to achieving improvements in this area, then evaluation and test should be a KPA.

3.5 The Cost of Defects

The cost of defects is rising at an exponential rate. This has two causes. The first is that our dependence on software is greater than ever. When it fails its impact is proportionate to that dependence. The second cause is litigation. There is a significant increase in the number of lawsuits concerning software quality. These are usually multi-million dollar exercises.

The support costs for software vendors is a growing concern. Microsoft receives over a 100,000 calls per day at an average cost per call of somewhere between $50 to $100. You also have to factor in the costs for developers to fix the defects and the opportunity loss caused by efforts going into fixing defects instead of creating new functionality.

Quality and the lack thereof also moves market share. Ashton-Tate went from being the industry leader in PC based data base software to being out of business due to large numbers of defects in one release of their main product. Market share for dBase went from 90%+ to less than 45%. Their acquisition by Borland did not stop the slide. Furthermore, only one year after their acquisition only 2% of all the people who had worked for Ashton-Tate still had jobs at Borland.

One area where software quality has become highly visible is on the web. When a problem occurred in mainframe batch systems it could be corrected in a few hours without much impact. With client server applications more people were immediately impacted, but these were still internal to the company. When there are defects in web based applications used by the customers, the defects are immediately visible and shake their confidence in your company. The competition is only a few mouse clicks away. The quality criteria for web based applications is at least two orders of magnitude higher than for even client-server applications.

The direct costs of defects can be staggering for the end users of the software. Both United Airlines and American Airlines estimate that they lose $20,000 a minute in unrecoverable income when their reservation system goes down. A large manufacturer estimates they lose $50,000 a minute when their assembly line goes down. A large credit card company estimates they lose over a $160,000 a minute when their credit authorization system goes down. Million dollar defects are now common place. For example, if GM has a defect in the firmware that requires reloading the control program in an EPROM it could effect 2.5 million automobiles at an average cost to GM of $100 per car. There has even been an instance of a BILLION dollar loss due to a single defect. It was caused by a round-off error.

Some estimates place the average cost of a severity one defect in production in the tens of thousands and even the hundreds of thousands on some applications. You can do a lot of evaluation and test for a $100,000. You could add an additional senior tester to the organization and, counting their salary and overhead costs, the break even point occurs when they find one or two defects that would have slipped through to production.

When you are dealing with safety critical systems how do you cost out the value of a human life? There have been hundreds and hundreds of deaths due to software defects. With software playing a bigger role in transportation and in the medical profession, the risk of deaths is rapidly increasing.

The legal profession is beginning to take note of these costs. Many feel we should be held to the same standards as other engineering professions. This leads to the exposure of software product liability and professional malpractice. The financial exposure in such suits is enormous. To date the issue of setting legal precedents in this area is still in a state of flux. However, the trend is clear. Software professionals and their products will be held to the same standards of care and professionals as other engineers and their products.

Currently, most of the lawsuits related to software quality are being brought to court on the grounds of breach of contract. MY prior company, Bender & Associates (since merged with TBI), was involved in a number of these as expert witnesses, and never lost a case. This is because in each instance Bender & Associates was been on the side of the software user, not the producer.

Few software vendors can demonstrate that they have applied a reasonable level of due diligence in the evaluation and test of their software. The emphasis in most vendors is on dates and functionality, not quality. The result is that in half of the cases we have testified in the vendor has gone out of business as a direct result of the cost of litigation and the cost of the award to the customer.

If the CMM was addressing the medical profession, there is no doubt that the avoidance of malpractice suits would be a KPA. Well this issue is now on our doorsteps as software professionals. It requires a disciplined approach to evaluation and test to minimize this exposure.

The net is that the direct and indirect cost of defects is already huge and rising dramatically. Defect detection and defect avoidance require fully integrated evaluation and test. This alone is sufficient to justify an evaluation and test KPA.

4. The Proposed Software Evaluation and Test KPA

Evaluation is the activity of ensuring the integrity of the various system specifications and models produced during the software development process. Testing is the machine-based activity of executing tests against the code. The purpose of Software Evaluation and Test is to validate (i.e., is this what we want) and verify (i.e., is this correct) each of the software project deliverables, identifying any defects in those deliverable in a timely manner.

Software Evaluation and Test involves identifying the deliverables to be evaluated/tested; determining the types of evaluations/tests to be performed; defining the success criteria for each evaluation/test; designing, building, and executing the necessary evaluations/tests; verifying the evaluation/test results; verifying that the set of tests fully cover the defined evaluation/test criteria; creating and executing regression libraries to re-verify deliverables that have been modified; and logging, reporting, and tracking defects identified.

The initial deliverable to be evaluated is the software requirements. Subsequently, the majority of the evaluation and test is based on the validated software requirements.

The software evaluation and test may be performed by the software engineering group and/or an independent test organization(s), plus the end user and/or their representatives.

4.1 Goals

Goal 1	Quantitative and qualitative evaluation/test criteria are established for each of the software project deliverables.
Goal 2	Evaluations/tests are executed in a timely manner to verify that the success criteria has been met.
Goal 3	Evaluation/testing is sufficiently effective to minimize the impact of defects such as scrap and rework during development and operational disruptions after implementation.
Goal 4	Defects and other variances identified are logged and tracked through to their successful closure.

4.2 Commitment to Perform

Commitment 1 — The project follows a written organizational policy for evaluating/testing the software project deliverables.

This policy typically specifies:

1. The organization identifies a standard set of software project deliverables to be evaluated/tested, the characteristics to be evaluated/tested, and the levels of verification criteria to be considered.

Examples of deliverables to be evaluated and tested include:

- requirements specifications
- design specifications
- user manuals
- training materials
- data conversion specifications and support systems
- code

Examples of characteristics to evaluate/test for are:

- functional integrity
- performance
- usability
- security

Examples of levels of verification criteria are (using code based testing as the example):

- 100% of all statements and branch vectors
- 100% of all predicate conditions
- 100% of all first order simple set-use data flows
- 100% of all first order compound set-use data flows

Examples of levels of verification criteria are (using requirements based testing as the example):

- 100% of all equivalence classes
- 100% of all functional variations
- 100% of all functional variations, sensitized to guarantee the observability of defects

2. The organization has a standard set of methods and tools for use in evaluation/testing and defect tracking.

3. Each project identifies the deliverables to be evaluated/tested, the phase(s) in which they will be evaluated/tested, and how they will be evaluated/tested in each phase.

4. Evaluations and tests are performed by trained testers.

5. Evaluations and testing focuses on the software project deliverables and not on the producer.

Commitment 2 — Senior Management supports and enforces that projects must meet their pre-defined success criteria before installation into production in the users/customers environment.

1. Senior management reviews and approves the overall evaluation and testing objectives for the software system.

2. Senior management reviews and approves that the system has met that criteria prior to installation.

Author's note: One of the biggest enemies of quality is unreasonable schedules. If the team is going to measured solely on just meeting dates, then the test plan will be bypassed. Management must measure functionality, resources, schedules, and quality in determining a project's success, not just dates.

4.3 Ability to Perform

Ability 1 — Adequate resources and funding are provided for planning and executing the evaluation and testing tasks.

1. Sufficient numbers of skilled individuals are available for performing the evaluation and testing activities, including:
- overall evaluation/test planning
- evaluation/test coordination
- evaluation/test case design
- evaluation/test case implementation
- evaluation/test execution
- evaluation/test results verification
- evaluation/test coverage analysis
- defect logging and tracking

2. Tools to support the testing effort are made available, including:
- test case design tools
- test data generators
- test drivers
- test coverage monitors

3. A test environment configuration is made available, including:
- hardware and software, dedicated to the testers, which mirrors the intended production configuration

Ability 2 — Members of the software testing staff receive required training to perform their technical assignments.

Examples of training for evaluation and test include:
- evaluation and test planning
- criteria for evaluation/test readiness and completion
- use of the evaluation/testing methods and tools
- performing peer reviews

Ability 3 — Members of the software engineering staff whose deliverables will be evaluated and tested receive training on how to produce testable deliverables and orientation on the overall evaluation and testing disciplines to be applied to the project.

Refer to Ability 5 for an example of a testable deliverable.

Ability 4 The project manager and all of the software managers receive orientation in the technical aspects of the evaluation/testing criteria and disciplines to be applied to the project.

Examples of orientation include:

- the evaluation/testing methods and tools to be used
- the entry and exit criteria for the various levels of evaluation/testing
- the defect resolution process

Ability 5 The software engineers produce testable deliverables.

An example of a testable deliverable would be a requirements specification that had the following characteristics:

- the functional rules are written at a deterministic level of detail (i.e., given a set of inputs and an initial system state you should be able to follows the rules in the specification and determine the outputs and the final system state)
- the specification is non-redundant
- the specification is unambiguous
- the various requirements follow a consistent standard (e.g., standards for user interface definitions are followed which define function keys, intra-screen navigation, inter-screen navigation)

4.4 Activities Performed

Activity 1 The overall evaluation and testing effort is planned and the plans are documented.

These plans:

1. Identify the risks and exposures if defects propagate through the various project phases and into production. This information is used to determine how much evaluation and testing needs to be done.

Examples of risks to be evaluated are:

- the potential scrap and rework and resulting cost and schedule overruns which might be caused by defects in the requirements specifications
- the potential cost per unit of time for system down time in production
- the potential cost to customers and end users of inaccurate processing
- the potential risk to human lives in safety critical applications

Note: The premise here is that testing is essentially an insurance policy. The overall evaluation and test strategy and its associated costs should be proportional to the potential bottom line risks which defects could cause.

2. Identify the software project deliverables to be evaluated/tested.

Examples of software project deliverables to be evaluated/tested are:

- requirements specifications
- design specifications
- code
- user manuals and built in help facilities
- training manuals, courseware, and training support systems
- data conversion procedures and data conversion support systems
- hardware/software installation procedures and support systems
- production cutover procedures and support systems (e.g., code that creates a temporary bridge between an existing system and its replacement, allowing some sites to run on the old and some on the new until full cutover is complete)
- production problem management procedures and support systems (e.g., the production help desk)
- product distribution procedures and support systems (i.e., the mechanisms for distributing updates and new releases, especially to widely distributed end users)
- publications procedures and support systems (e.g., the mechanisms for physically publishing all of the copies of the manuals needed to support the system in production)

3. For each deliverable to be evaluated/tested determine the characteristics to be tested.

Examples of characteristics to be evaluated/tested are:

- functional integrity
- performance
- usability
- reliability, availability, serviceability
- portability (i.e., can this one code line be easily ported from one platform to another)
- security (e.g., who can access which functions and what data)
- maintainability (i.e., can fixes and minor incremental improvements be easily made)
- testability (e.g., are test diagnostics part of the system)
- extendibility (i.e., can major additions be made to the system without causing a major rewrite)

4. Determine the qualitative and quantitative success criteria for each deliverable and each characteristic evaluated and tested for the deliverable.

An example of the functional test criteria for code could be:

- the code is tested to verify that 100% of all functional variations derived from the requirements, fully sensitized for the observability of defects, have been run successfully
- 100% of the code's statements and branch vectors have been executed

5. Determine the methods and tools required to evaluate/test each deliverable for each of its desired characteristics.

An example of evaluating a requirements specifications might involve:

- performing an ambiguity review
- walking use-case scenarios through the requirements to validate completeness
- building screen prototypes to validate the completeness
- creating cause-effect graphs from the functional requirements to validate that the precedence rules are clear
- doing a peer review with domain experts to validate completeness and accuracy
- doing a logical consistency check of the rules via a CASE tool
- reviewing the test cases designed from the functional requirements with developers and end user / customers to validate the completeness and accuracy of the specifications from which they were derived

Examples of testing tools include:

- test case design tools
- test data generators
- capture/playback tools
- test drivers
- test coverage monitors
- test results compare utilities
- memory leak detection tools
- debuggers
- defect tracking tools

6. Determine the stages (sometimes called levels) of testing and refine the quantitative and qualitative test criteria into entry and exit criteria for each phase of testing.

Examples of stages of code based testing include:

- unit testing with primary emphasis on white box structural testing, usually done by the coder
- component testing with primary emphasis on black box functional testing and inter-unit interface testing, with some initial performance testing and initial usability testing
- system testing with primary emphasis on inter-component interface testing, full thread functional testing, full performance testing, full usability testing, and full reliability/recoverability testing
- inter-system integration testing with primary emphasis on inter-application interface testing and inter-application performance testing
- acceptance testing (a.k.a. beta testing) with emphasis on final validation of functional robustness, usability, and configuration testing

An example of refining the success criteria by test stage is:

- the entry criteria into unit testing is a peer review of the code
- the exit criteria from unit test is correct execution of 100% of the code statements and branch vectors
- the entry criteria into component test is 100% execution of the "go right" statements and branches
- the exit criteria from component test is 100% execution of all functional variations derived from the requirements specification

Note that the entry criteria into component test is less stringent than the exit criteria from unit test. This allows these activities to overlap in a controlled manner.

7. For each deliverable, decompose it into units for evaluation and test and determine the optimal sequence for evaluating/testing the units.

> For example, the unit testing of the code might be done in a sequence which minimizes the need for building scaffolding code to emulate interfaces to code not yet tested.

8. Define the methods and procedures for defect reporting and tracking to be used by the project.

Activity 2 — Reconcile the evaluation/test plan with the overall development plan.

1. Verify the evaluation and test resources and schedules against the project schedules and constraints.

2. Reconcile the desired sequencing of units for evaluation and test against the availability of those units as defined in the development plan.

3. Get concurrence on the defect reporting and tracking mechanism from the developers.

Activity 3 — Install the evaluation and testing infrastructure.

1. Acquire and install the testing tools needed for this project.

2. Acquire and install the test hardware and software configuration required to create and execute the tests.

3. Train management and staff on the evaluation and testing methods and tools to used.

Activity 4 — Perform the evaluation/testing for each deliverable, for each characteristic, at the designated test stages.

1. Design the evaluation/test cases using the identified methods and tools.

2. Physically implement the cases in their final “executable” form.

3. Perform the evaluation / Execute the test cases.

4. Verify the evaluation/test results against the expected results.

5. Verify that the evaluation/tests fully covered their target objectives.

6. Provide periodic reports as to the status of the evaluation/testing effort against the test plan.

Activity 5 Defects detected are reported, tracked till closure, and analyzed for trends according to the project's defined software process.

Examples of the kinds of data to be collected include:

- defect description
- defect category
- severity of defect
- units causing/containing the defect
- units affected by the defect
- activity where the defect was introduced (i.e., root cause)
- evaluation/test that identified the defect
- description of the scenario being run that identified the defects
- expected results and actual results that identified the defect

Activity 6 Perform regression testing as needed.

1. Create regression test procedures and test libraries for use in revalidating changes to deliverables.

2. Execute the regression test procedures and test libraries anytime modifications are made to already tested deliverables.

Activity 7 Revise the evaluation and test plan as needed.

1. Review the effectiveness and efficiency of the evaluations and testing to date and the defects reported to refine the evaluation and test plan as needed.

4.5 Measurement and Analysis

Measurement 1 Measurements are made to determine the effectiveness of the evaluations and testing.

An example of the measurements includes:

- defect removal rate by phase (i.e., the portion of defects removed in an evaluation/testing phase that were introduced in the corresponding development phase)

Measurement 2 Measurements are made to determine the completeness of the software evaluations and testing.

Examples of the measurements include:

- using a functional coverage analyzer to determine what percentage of the requirements have been validated
- using code coverage monitors to determine what percentage of the software statements and branches were executed by the test cases

Measurement 3 Measurements are made to determine the quality of the software products.

Examples of the measurements include:

- analysis of the mean time to failure and the mean time to fix by severity of defect
- analysis of the distribution of defects by unit
- analysis of the number and severity of the unresolved defects
- analysis of the closure rate for defects versus the rate new ones are being reported

4.6 Verifying Implementation

Verification 1 The activities for software testing are reviewed with senior management on a periodic basis.

Refer to Verification 1 of the Software Project Tracking and Oversight key process area for practices covering the typical content of senior management oversight reviews.

Verification 2 The activities for software testing are reviewed with the project manager on both a periodic and event-driven basis.

Refer to Verification 2 of the Software Project Tracking and Oversight key process area for practices covering the typical content of project management oversight reviews.

Verification 3

The software quality assurance group reviews and/or audits the activities and work products for software evaluation and testing and reports the results.

Refer to the Software Quality Assurance key process area.

At minimum, the reviews and/or audits verify that:

1. All parties are involved in the definition of the software evaluation and test approach and are committed to implementing it.

2. The test criteria and test methods are appropriate in light of the defect impact risk assessment.

3. The software project deliverables are testable as defined by the project's standards.

4. The entry and exit criteria for each stage of evaluation and test is being adhered to.

5. The evaluation/testing of all of the software project deliverables is performed according to documented plans and procedures.

6. Evaluations and tests are satisfactorily completed and recorded.

7. Problems and defects detected are documented, tracked, and addressed.

8. The test cases are traceable to the software products they test.

5. Reconciling With the Existing CMM KPAs

The CMM has been in use for a number of years now in a growing number of organizations. This makes modifying it problematic. If it changes too drastically, what does that do to all of the organizations which have achieved certain certification levels based on the prior version? How do modifications to the CMM affect process improvement efforts already underway? In this section we will deal with two topics. The first is leveling the Software Evaluation and Test KPA into the overall CMM. The second is some repackaging suggestions to ease adding the additional KPA without passing a pain threshold of having too many KPAs.

Leveling the Evaluation and Testing KPA Within the CMM

Currently, testing is part of the Software Product Engineering KPA which is at Level 3. However, many of the Level 2 KPAs are dependent on having a disciplined approach to evaluation and test in place. As stated in the justification section it is difficult to solve problems until those problems are well understood. Evaluation and test helps provide this insight. It is, in fact, one of the key drivers of cultural change that positions an organization to aggressively address many of the other KPAs.

The CMM recognizes the criticality of good requirements to the whole process. The Requirements Management KPA is appropriately KPA number 1. However, experience over the last two decades has shown it is difficult to get really good requirements without concurrently installing requirements based evaluation and testing. This provides the necessary tight feedback loop on the quality of the requirements as they are being written.

The Software Project Tracking and Oversight KPA, another Level 2 item, also requires the Evaluation and Testing KPA. Tracking involves determining what tasks are actually completed versus what was planned to be completed. However, without verifying that the tasks have met their completion criteria you really do not know that the tasks are truly completed.

The Software Subcontract Management KPA, a Level 2 KPA, also requires the Software Evaluation and Testing KPA to unambiguously define the success criteria contractually and to verify that that criteria has been met. All of the legal disputes that I have testified in as an expert witness were the result of not having formal evaluation and test defined and executed.

Given the above, the recommendation is made that the Software Evaluation and Test KPA be made a Level 2 KPA.

Repackaging Suggestions for the Existing KPAs

The most obvious re-packaging is splitting the Software Product Engineering KPA into two KPAs: Software Evaluation and Test and Software Product Engineering with a reduced scope. The name of the latter should probably stay the same unless the new scope causes confusion.

The Peer Reviews KPA should be subsumed into the Evaluation and Testing KPA. As discussed, peer reviews are just one means of performing an evaluation. Separating out a single evaluation technique and making it a full KPA is a bit disproportionate. However, as an admitted testing bigot, I would not argue very hard against keeping it. It adds emphasis to the overall importance of evaluation and test.

Some have suggested that the Software Evaluation and Test KPA itself could be split into an Evaluation KPA and a Testing KPA. My own feeling is the process loses some continuity if that is done. However, it is not something I would argue too vehemently about.

In order to keep the number of KPAs down, I would suggest that the Software Project Planning and Software Project Tracking and Oversight KPAs be merged into one KPA. These are very tightly coupled activities. Xerox, for example, is treating them as essentially one item to install in their CMM activities. I cannot believe they are alone in this view. While this does not have anything directly to do with testing, it does help make room for a Software Testing KPA.

