COMMUNICATIONS DESIGN CONFERENCE September 30-October 2, 2003 San Jose Convention Center

Embedded.com

Verification and Validation

By Charles Knutson and Sam Carmichael, Embedded Systems Programming
Jun 1 2001 (11:56 AM)
URL: http://www.embedded.com/showArticle.jhtml?articleID=9900163

Every product has defects. Finding them as early in the development process as possible is definitely something to strive for.

Building quality into software as it's being developed is far more effective than trying to test it in after it's been built. Verification and validation techniques can be applied throughout the product lifecycle to help assure that the correct product is being built and that the product is being built correctly. This article introduces the fundamental theory and techniques of verification and validation and discusses how these techniques have been successfully applied in the creation of high quality embedded software.

The challenge of quality

We typically give the name "quality assurance" to the software development function involved in testing or checking a product after it's been built. But do these folks actually "assure" anything? Granted, if given proper authority, they can assure that a bad product doesn't get out the door. But can they actually assure that a good product gets built in the first place? Very unlikely.

A deeper question might be, can you test quality into a product? Sure. You just have to construct a testing sieve so tight that no bad product can get through it. (Good luck building such a test suite.) Then you begin systematically rejecting everything that doesn't pass. (Good luck surviving the political fallout.) The idea behind this approach is that eventually the development folks will begin to figure out that they have to build good products, and they'll begin to change the way they do things. Not the epitome of software process improvement. And yet, it's an all too common scenario in software companies today.

When the software products being created are destined for embedded systems, the problem is exacerbated. When you burn control software onto a ROM, put it in a device, and ship it to millions of customers, you'd like to have a deeper sense of the innate quality of that software. Or when you put a team of astronauts onto a launch pad and trust their lives to the software controlling the on-board computers, you'd like to think it's going to work the first time.

Yes, you need to test. But the product you're testing must have been built with some level of quality in the first place, particularly when your product is an embedded system of some kind. This idea was well expressed by Boris Beizer, oddly enough (or perhaps appropriately enough) in a book about testing:

"The single most important thing that can be done to achieve quality software is to design the quality in. That's more important than how the quality assurance department is structured, who it reports to, what testing is independent, what kind of reviews are held-more important than the

entire contents of this book, of Software Testing Techniques, and of the next ten books published on software quality assurance."[1]

We concur completely with the notion that quality must be designed into software. So how do you determine whether quality was inherent in the design in the first place? One important idea is to have an eye toward quality throughout the product life cycle, from requirements elicitation all the way to first customer ship.

To incorporate an attention to quality at each phase of software development requires a process to govern its application. Most software process improvement models (such as CMM and ISO 9000) begin with an assumption that effective and appropriate processes naturally and inevitably lead to the production of high quality software.

"An underlying assumption of quality management is that the quality of the development process directly affects the quality of delivered products. This assumption is derived from manufacturing systems where product quality is intimately related to the production process."[2]

While it can be debated whether producing software is similar to producing steel[3], it seems reasonable that a repeatable, common sense process with an eye to quality from the beginning stands a greater chance of producing quality software than a chaotic, ad hoc group of programmers hacking together solutions and then testing them until they appear to work. As Peter Coffee stated, "Quality is not a feature that can be added to a current product: It is a process, one that begins with product design and continues long after the product is sold."[4]

We believe that effective process will indeed contribute to the production of high quality software that works right the first time. In the following sections, we will introduce basic principles of defect management, and then discuss verification and validation of software through the product life cycle. In particular we will discuss reviews, inspections, and testing as mechanisms for performing verification and validation.

Defect management

Software contains faults or defects, which are errors in software introduced by developers. These defects may have been introduced at virtually any point in the development process from requirements to maintenance. These defects may lay dormant if the proper circumstances never arise to force the problematic code into execution. Or they may become evident as failures. Failures span a range of severity. In the worst case, the failures may take the form of system crashes or incorrect system functionality. In milder cases, failures may simply make users unhappy or dissatisfied (such as slow response time or an interface that's difficult to use).

Defects are to be avoided, of course. Two guiding principles govern the management of defects. First, avoid introducing defects in the first place. That can be done by applying proper techniques at each step in the product life cycle. For example, many defects are actually introduced during requirements elicitation. And yet few software engineers have received any formal training in this important function. By performing effective requirements elicitation, it's possible to avoid introducing a significant number of defects. The same can be said for every other phase in the product life cycle.

We know that, despite our best efforts, defects will be introduced into our products. The second governing principle, then, is to detect defects as early in the process as possible. Once a defect has been detected, it needs to be removed at the source. This means that if the defect was introduced during low-level design, it needs to be removed there-ideally before coding begins. If the defect was introduced during requirements elicitation, it needs to be removed there-ideally before high-level design begins.

The longer we wait, the greater will be the cost involved in removing and repairing a defect. Studies have indicated that the cost of defect removal rises dramatically the later they are discovered in the product life cycle.5 To avoid inserting defects requires training in the particular skills involved in each phase of software development.

The focus of verification and validation is to detect defects as early as possible after they are introduced and remove them at the source. Doing so not only makes the removal of defects cheaper, it also provides a much stronger confidence that quality is being built into a product, rather than trying to filter it in just before shipping.

Verification and validation

Verification and validation (commonly referred to as V&V) is concerned with answering two fundamental questions: did we build the right product, and did we build the product right? Speaking broadly, validation is concerned with building the right product, and verification is concerned with building the product right.

The following definitions may shed light on what we mean exactly by building the right product and building the product right. After these three definitions, we'll attempt our own summary of what we mean by V&V.

Definition #1
Validation is the "determination of the correctness of the final program or software produced from a development project with respect to the user needs and requirements. Validation is usually accomplished by verifying each stage of the software development life cycle."

Verification is defined as the "demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development life cycle."6

These definitions point out that validation is primarily concerned with making sure that the requirements have been met by the final product. Verification is then concerned with the translation and traceability of each stage of development to its dependent stage. In other words, design can be shown to correctly derive from requirements. This definition makes the assumption that validation is commonly achieved through verification of each phase.

Definition #2
"Verification involves evaluating software during each life-cycle phase to ensure that it meets the requirements set forth in the previous phase. Validation involves testing software or its specification at the end of the development effort to ensure that it meets its requirements (that it does what it is supposed to). While 'verification' and 'validation' have separate definitions, you can derive the maximum benefit by using them synergistically and treating 'V&V' as an integrated definition."7

This definition takes the view that validation is essentially what we typically refer to as "system test" and involves assessing whether a final product meets its original requirements. It then takes the leap that these two terms can be used "synergistically" as V&V. We would suggest that this approach only works because they both begin with the letter V and thus can be conveniently bunched together. To do so is to lose a tremendous amount of the power inherent in the distinct focus of each.

Definition #3
"Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). Verification refers to the set of activities that ensure that software correctly implements a specific function. Validation refers to a different set of activities that ensure that the

software that has been built is traceable to customer requirements."[8]

This last definition is from a college textbook, and so is a little simpler than the others. The definition of validation is agreeable, but the definition of verification here is a bit too simplistic. There is more to verification than seeing that functions are implemented correctly.

In this article, we will use "validation" to refer to those activities that attempt to determine that customer needs can be met by a product. This may include usability testing or other types of user feedback. It may involve inspection of requirements documents or assessing whether requirements elicitation was performed effectively. It may also include testing of the final system with respect to the original user requirements to see that those requirements were met. Hence, validation helps to see that we are building the right product.

We will use "verification" to refer to the transformational activities that are performed at each step of the product life cycle. In other words, from a user requirements specification, a high-level design can be made. At the point that the design document is complete, it can be "verified" against the requirements document. At this point, defects can be detected and corrected. This high-level design can then be used to verify the low-level design document that stems from it. This process of verification applies at each stage in the development process and can include essentially every document or artifact produced along the way, including (in addition to the documents already mentioned) source code, internal documentation, user documentation, test plans, and test specifications.

The most common way to perform validation of a system is through testing. Few other options are available. Besides, if you have an accurate requirements document and a functioning system, running it through its paces to see if it meets the defined requirements makes a great deal of sense. But how do you assess the quality of a document? That's not as straightforward. The easiest answer is, read it and talk about it. See if it's traceable to the document from which it was derived. This process is largely one of reviews and inspections.

Reviews and inspections

"Technical work needs reviewing for the same reason that pencils need erasers: to err is human."9 It's possible to perform reviews at every stage of development, so long as documentation exists for each stage. For example, if software is being built without defined accurate requirements, it is essentially impossible to verify, via reviews or any other method, whether the design is accurate. Since the design is not based on requirements, it simply stands alone, and any assertion of defects ultimately devolves into a matter of personal opinion. So performing reviews and inspections presumes a certain level of rigor in the process being applied to the creation of the software in the first place.

Reviewing the intermediate development artifacts at each stage of development has two primary values. First, we can detect defects early and remove them when the cost is relatively low. Second, and possibly more significant, we can influence the process within our company, forcing a greater amount of rigor in the creation of the software in the first place. If management buys off on the value of reviews, it will become immediately obvious that without good requirements (or good design) reviews won't bring much value.

Reviews typically involve a small group of people all looking at the same work product or development artifact. Why involve other people? For the same reason an author can skip past a typo in an article a dozen times, while a copy editor will see it more readily: we all have blind spots. "We need technical reviews because although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else."[9]

Despite the benefits, there are challenges to reviews, some of them social. For some, the thought of bringing one's heretofore private work product under the scrutiny of a room full of people is disconcerting at best. It's not a pleasant experience to have one's baby dubbed ugly in a public setting. For that reason, the scope of these meetings should be limited to a handful of people, and all participants should be trained, so that negative repercussions can be avoided.

Reviews function as a form of quality filter that is applied at various points during software development. The motivation behind reviews is to uncover errors and purify work products as early as possible. Specifically, reviews attempt to achieve the following outcomes:

- Point out needed improvements in the product.
- Confirm the parts that are good.
- Bring some consistency to the product in terms of coding style, document style, design approach, and so on, which makes the technical work more manageable.
- Improve the software development process.

Reviews can span the spectrum of formalism, from extremely formal to very informal. In the most formal settings, many people may participate (although there are clearly points at which more participants will reduce effectiveness), tremendous corporate resources may be expended, and lots of photocopies are needed to keep everyone on the paper trail. On the other extreme, there are very informal gatherings that are reviews nonetheless. These may involve just one or two people gathering in a cube, or in the hall. It may be as simple as a request for help from one engineer to another.[10]

We typically use walkthrough to refer to less formal meetings in which a group reviews some work product. In these informal meetings, few rules govern the meeting, diversions are common, and the group often jumps into problem solving mode. These kinds of gatherings can be very valuable under the right circumstances.

We typically use inspection to refer to more formal meetings in which specific roles are played by participants, specific rules govern the meeting, and greater rigor is applied, particularly when involving the traceability of one work product to its predecessor.

Focus for V&V activities

A great number of things can be checked for during V&V activities, but four are particularly significant: completeness, consistency, feasibility, and testability.

Completeness means that a work product is complete with respect to its predecessor. This means that there are no items marked "TBD" (to be determined), there are no non-existent references, no missing specification items (particularly unconsidered special cases), no missing functions, and no missing products. If a previous work product identifies a function or feature, then the work product being reviewed must have its analogous treatment of the same function or feature.

Consistency means that the work product does not create conflicting requirements. Consistency can be internal (meaning within the work product itself) and external (with respect to another work product). For example, a function may have a definition of output values that conflicts with the input values of another function that it calls. Each function may be internally correct, but the defect in the specification will become evident when the two functions are integrated later in the product life cycle.

Feasibility relates primarily to the ability of delivering a work product that depends upon the one being reviewed. For example, a requirement may be internally consistent and correct from the

user's perspective, but not feasible to construct given the level of human resources allocated to the product. Similarly, a specification may require a particular level of performance by a module which is almost certainly not achievable given the available technology. Feasibility should also explore issues of risk, whether such risk relates to technical issues, cost or schedule, environment, or interactions between the system and other systems or users.

Testability relates to the ability to perform specific tests on appropriate work products. In particular, a work product must be specific, unambiguous, and quantitative or it cannot be tested. For example, imagine a specification that states, "The sorting algorithm should be as fast as possible." Such a requirement cannot be tested in a work product, because no objective measure exists to determine whether a test passed or failed. Paying attention to testability not only makes the product more testable (obviously) but it leads to a higher quality work product (whether that work product is tested or not), because designers and coders will have had to clarify these issues before the final product is constructed, instead of discovering the problem later. In addition, testability includes identifying "exit criteria," which help test engineers to know when the product has been tested to a sufficient level before shipping.

Review meetings

Review meetings are typically limited in attendance, involving from three to five people. In these meetings, members play specific roles for which they should already be trained. For these meetings to be successful, each participant needs to prepare in advance, typically investing less than two hours. In addition, the meeting must be kept to a reasonably short length, typically under two hours. Such a meeting allows reviewers to focus on only a small portion of the work product. The following are important principles that should guide review meetings.

Review the product, not the producer. The producer of the work product is not on trial. It is difficult enough to objectively discuss a work product without getting personal about it. In some extreme formats, a producer may either be asked to serve as a reader only, or as a bystander, or may be not invited to the meeting at all. In addition, the results of the review should not be held against an engineer, although defects that make it into shipping products may be.

Set an agenda and keep it. There is a large cost involved in taking up to four hours from each of five engineers (collectively about half an engineering week) for a review of some work product. If such reviews are to succeed (and/or survive as a part of the process), all participants must have the sense that their time has been well spent.

Limit debate and rebuttal. The moderator must keep control of the meeting. If disagreements arise, they should be noted by the scribe, assigned as action items to appropriate individuals, and the meeting should move on. Any debate and rebuttal beyond the most minimal level will violate the previous principle of setting an agenda and keeping it.

Enunciate problem areas, but don't try to fix anything. The focus of the meeting is on discovery, not repair. The purpose of the group in a formal review is not to collectively design. As tempting as it is, the urge must be suppressed, and the meeting must move forward in its path of defect discovery. The responsible engineer will have ample opportunity afterward to make changes.

Take written notes. A scribe should be designated. Ideally this person has no responsibility other than writing, so that a written record can be created for the benefit of the individual responsible to make changes after the meeting.

Limit the number of participants. Avoid giving special permission to bystanders who just want to be "flies on the wall." There tends to be an optimal number of bodies in the room and exceeding that number will almost certainly limit the effectiveness of the meetings.

Insist upon advance preparation. In order for the meeting to run on schedule, and for the time of all participants to be well-spent, each attendee must prepare in advance. Without advance preparation, everyone's time will be wasted while the unprepared make up their learning curve in front of everyone else. Typically, material should be distributed to participants two to three days in advance of the scheduled meeting.

Develop a checklist for each work item to be reviewed. Checklists are absolutely essential for reviews and inspections. Specific checklists should be developed for various kinds of work items. These checklists can be used to walk through and perform a thorough check of relevant issues. In addition, as meetings progress and other questions are raised, the checklists themselves should evolve and become more thorough and complete over time.

Allocate resources and time schedules. If reviews are to be successful, management must understand the time investment that they represent, and schedule accordingly. If reviews are scheduled without giving the participating engineers credit for the time spent, they will not succeed. Doing so may represent a leap of faith for management, but without it, the process is essentially doomed.

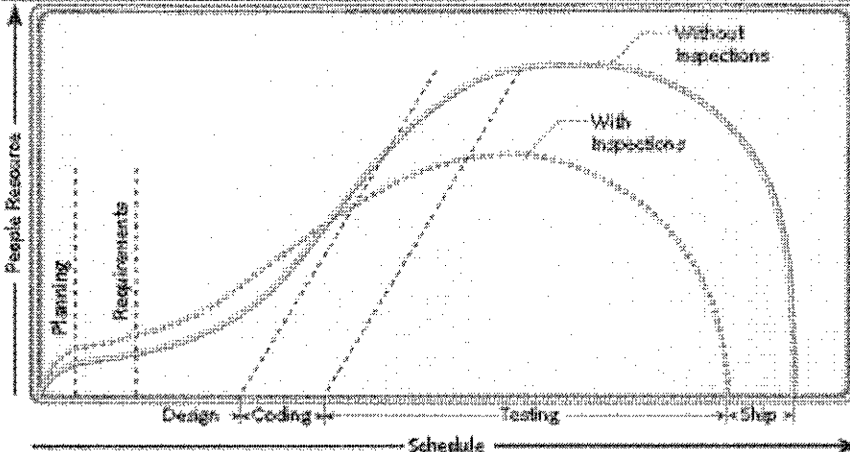
Conduct meaningful training for all reviewers. While running an effective review meeting may be intuitive to some, it's important that all participants be trained in the appropriate roles. Such training may have less to do with imbuing the participants with the requisite skill sets, as much as identifying for all the appropriate rules of engagement. Even a talented and sincere group of people can waste tremendous time and conduct ineffective reviews if they don't all understand the ground rules.

Review your early reviews. No single static process will help every company in every circumstance. Reviews must be customized and changed over time to meet the needs of the development team. Therefore, the reviews themselves should be reviewed in order to recommend appropriate changes in the process.

Evidence suggests that inspections are cost effective in many situations. The primary reason for this is that they reveal defects early in the cycle when they are relatively inexpensive to fix. In addition, they can prevent the ripple effect that occurs when making a change to requirements necessitates changes to multiple levels of design documents, product code, and product documentation. Another key to the success of reviews is that they are designed to detect defects, not failures. Finally, reviews naturally enforce improvement of the process, since each work product must be traceable to a preceding work product. Immature organizations that do a poor job of requirements and design will have few appropriate artifacts that can be used for reviews.

While it's clear that reviews can contribute greatly to the quality of software, it may not be intuitive that they would necessarily be less expensive. But some of the evidence suggests that they may actually be. The first assertions were made by Fagan in his early work on inspections. **11** Figure 1 shows a "snail curve" that attempts to explain how spending more in inspections up front may pay off in less testing time (primarily bug fix and regression testing cycles) at the end and hence yield a collective cost benefit.

Figure 1: Fagan's "snail curve," representing the cost-effectiveness of inspections



So where does that leave testing? Right where it has always been. For all the confidence we can place in things we design and build, even when we do it carefully, there's nothing quite like putting it in water and seeing if it floats. It's still appropriate to run the software through its paces, because there are still bound to be defects. However, if inspections are applied well, the number of defects left in the product when testing begins will be dramatically smaller.

Testing

While formal reviews and inspections have occupied the bulk of our attention, testing is still a significant and important part of V&V. When we test, we run software under controlled conditions to detect defects. Just as we perform reviews and inspections at every appropriate level during the requirements, design, and coding phases, we test wherever appropriate as software is being created. The following sections briefly describe unit, integration, system, and regression tests.

Unit testing

The role of unit testing is to exercise a specific module in a controlled environment. This typically involves some form of scaffolding, typically stubs and drivers. Stubs are modules of test software that sit below a unit under test and mimic the behavior of dependent modules. Drivers are modules of test software that sit above a unit and drive it in the same fashion that its calling modules might do. Stubs and drivers typically work in harmony to create conditions or states in which the module under test must respond.

Unit testing typically verifies a module's functional attributes, but may also test other things such as performance, usability, and so on. It may be performed using either "black box" or "clear box" (sometimes called "white box") methods. In black box testing, a module is treated as a box whose internal behavior is not known. This box can be viewed as performing the function of mapping input values to output values. Since these mappings should be specified, you can test possible input values or conditions against the resulting outputs. Creating such tests is typically done with no reference to the internal code, and the test cases either pass or fail.

In contrast, clear box testing requires an understanding of the code (although it should be equally dependent on the appropriate specifications). For example, a test might focus on control flow by seeking to exercise every line of code, or assuring that every decision point is traversed in every possible way. However, a test might also focus on data flow by seeing that all data manipulations are considered.

Unit testing is typically viewed as a form of verification, since the behavior of functioning modules is compared to functional specifications.

Integration testing

Once modules have been individually unit tested, we begin bolting them together to see how they function when integrated with other modules. Modules can be integrated in a number of ways, including top-down and bottom-up. Any approach can be taken so long as the tester understands the appropriate behavior represented by a specific combination of modules, starting with the first two, and ending with the adding of the last module to form a complete system. Ideally, integration testing involves testing as each module is added.

Integration testing is typically viewed as a form of verification, since the behavior of modules in combination is a product of functional specifications.

System testing

System testing involves the execution of an entire system or product to see that it conforms to the overall system requirements. The system tester should be the customer's advocate, since the guiding document should be either the user requirement document or a specification directly based upon it.

System tests can be performed in different ways, including both simulated and real execution. In some systems, such as the launch of a new space shuttle, it is impossible to test the actual software in operation. So elaborate scaffolding is created to simulate all of the external conditions that the system must be able to respond to. In such situations, there is an important dependency on the quality of the test simulator, which causes an interesting quality dilemma. While the same can be said for any automated system test, simulators are particularly challenging because of their complexity, and because of the lack of other alternatives to validation.

System tests can pursue any number of quality factors including (but certainly not limited to) functionality, performance, reliability, and usability.

Regression testing

Regression testing deals with testing software after bug fixes have been made to assure that the software has not "regressed" or gotten worse because of the fix. This typically involves the re-running of the original test suite that found the problem in the first place. There are two points of focus for regression tests. The most important is to see that no new bugs were introduced. The second is to see that the bug fix actually caused the failure to go away. For effective regression testing, automated testing is almost an absolute must. There are two broad classes of tests: slow ineffective ones, and fast effective ones. If you use fast effective ones, there is relatively little pain associated with regression testing.

Who should test the software?

Testing is a profession with a specific set of skills that are employed to perform these critical V&V functions. Good testers are involved in the development of software from the beginning, bringing a quality perspective to the product life cycle. They apply engineering skills to create effective and efficient (and automated) test suites. They design these tests from requirements, and apply them in performing verification and validation functions. They understand that attempting to break the product is in the best interests of the company and the customer. They are engineers whose product is a test, and their skills are not strictly the same as those employed by development

engineers. Despite that, broadly speaking, two types of people often get roped into testing: users and developers. There are problems with both.

A developer's job is constructive. He builds things. A tester's job is destructive. He breaks things. There's a specific personality associated with each job, and it's not trivial to casually jump between them. More importantly, developers should never be counted on to test their own code. For one thing, they unconsciously flinch when they approach the tender areas of their code. Without even realizing it they may skirt the problem areas in their software. They just can't bring themselves to swing the hammer with sufficient force. But even if you can manage to brainwash a development engineer into beating on his own code, he will still be less effective than a fresh pair of eyes. The same blind spot in this engineer that led to a particular defect will undoubtedly lead to a similar blind spot in the testing of the same software.

Another common approach is to get users to perform testing. While users can provide tremendous feedback on usability issues, they are typically very poor for functional testing if the software has any reasonable level of quality. Users tend to be repetitious, non-systematic, and unimaginative in their testing. They also tend to be slow, due to a lack of confidence around the new software. Having said that, users are excellent sources to understand potential use models, and hence may permit testers to focus test cases around potential problem areas based upon actual usage patterns.

Summary

Many companies have become comfortable with independent testing or validation functions. The need for a safety net (or quality sieve, if you will) is obvious, particularly when building embedded systems that don't enjoy quite the same ease of upgrading as other platforms. But in such companies, testing is typically an activity that occurs at the end of the process, and provides one of the few objective mechanisms to make sure a product possesses the requisite level of quality.

Our focus in this article has been on reviews and inspections because they are much less common in software companies, despite the fact that they provide a more dependable mechanism for building quality into a product as the development cycle unfolds. Visually inspecting a work product with a small group of people is painstaking in many ways, but the benefits are potentially enormous.

Every product has defects. That's not going to change. We will find defects in products sooner or later. The worst case is to find defects in the field. The best case is to find defects as early as absolutely possible in the software development process. That gives us a much more reasonable chance of producing software with fewer defects, and for the severity of those defects to be more acceptable to customers. When you're building a software product that will be embedded in some device, it's imperative that it be right before the product releases. By the time the next version of the device comes out, it might be too late.

Charles D. Knutson is an assistant professor of computer science at Brigham Young University in Provo, UT. He holds a PhD in computer science from Oregon State University. You can contact him at knutson@cs.byu.edu.

Sam Carmichael is a validation engineer at Micro Systems Engineering. Contact him at carmicha@biotronik.com.

References and Notes

1. Beizer, Boris. *Software System Testing and Quality Assurance.* New York: Van Nostrand Reinhold, 1984, p. 309.

Back

2. Sommerville, Ian. *Software Engineering, 5th Edition.* Reading, MA: Addison-Wesley, 1995, p. 615.
Back

3. Humphrey, Watts. "Characterizing the Software Process: A Maturity Framework," *IEEE Software,* March 1988, pp. 73-79.
Back

4. Coffee, Peter. "Attacking the Quality Monster," *PC Week,* December 14, 1998.
Back

5. Boehm, Barry W. and C. Papaccio. "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering,* October 1998, p. 1466.
Back

6. Adrion, W. Richards, Martha A. Branstad, and John C. Cherniavsky. "Validation, Verification, and Testing of Computer Software," *Computing Surveys,* June 1982, pp. 159-192.
Back

7. Wallace, Dolores R. and Roger U. Fugii. "Software Verification and Validation: An Overview," IEEE Software, May 1989.
Back

8. Pressman, Roger S. *Software Engineering: A Practitioner's Approach,* 4th Edition. New York: McGraw-Hill, 1997, p. 488.
Back

9. Weinberg, Gerald M. and Daniel P. Freedman. "Reviews, Walkthroughs, and Inspections," *IEEE Transactions on Software Engineering,* January 1984, pp. 68-72.
Back

10. These informal reviews can be tremendously valuable in discovering the sources of errors. A common pattern is one that we've heard called, "Tell it to the Furby." In this kind of review one engineer shares his problem with another engineer, who typically asks the first to begin by explaining what he's doing. While walking through his code or design, he will often see it from a different perspective, just because he had to state it out loud to the other engineer. The second engineer added no real value other than being the "Furby" for the first engineer to talk to. Still, a tremendous value was added.
Back

11. Fagan, Michael E. "Advances in Software Inspections," *IEEE Transactions on Software Engineering,* July 1986, pp. 744-751.
Back

Return to Table of Contents

Copyright 2003 © CMP Media LLC