Defect Management: A Study in Contradictions

Raymond Grossman
L-3 Communication Systems-East

In today's defense business environment, many software and system development contracts require mandatory compliance with standardized software or integrated process models such as ISO 9000, the Capability Maturity Model® (CMM®) or CMM Integration℠. One component that all these models have in common is the use of defects as a measure of process and product quality. Entire key process areas[1] and process areas[2] are devoted to defect prevention and defect causal analysis and resolution. With such a reliance on this metric as a key indicator, it would seem that the definition and interpretation of the term defect would be universally understood and accepted. However, nothing could be further from the truth.

Just as *beauty is in the eye of the beholder*, what constitutes a defect is guided by the personal biases and the organizational position of the person making the evaluation. Is a bug found by a developer during unit testing a reportable defect, or just a *normal*, non-quantifiable part of the development process?

To some (many) developers, the latter is true. Their contention may be that the whole point of unit testing is to uncover problems; if they were forced to document every bug as a defect, the job would never be accomplished on time and on budget. However, if a process engineer were asked the same question, he or she might suggest that the uncovering of a problem during testing is an opportunity to examine the cause of the problem, to determine in what phase the defect was introduced, and to reveal what earlier process failed to find the defect.

Management, in this scenario, is at cross-purposes. On one hand it has schedule obligations and does not want to burden its developers with doing *extra* paperwork. On the other hand, a *process savvy* management is aware of the economy of finding defects as early in the development life cycle as possible and thinks any opportunity for process improvement should be pursued.

A seemingly more straightforward scenario is to document defects at peer review meetings. These types of reviews can take many forms, but their basic task is to have a product (requirements, design, code, document, etc.) reviewed by a group of interested and knowledgeable individuals. The output of peer review meetings is a list of action items or defects that must be addressed or fixed in a timely fashion. Even in organizations that have defined peer review processes, there are often areas of contention when recording defects. Many times the line that divides the definition of action items versus defects is vague, and defects are incorrectly classified as action items. In other cases, several discovered defects, which seem similar in nature, are grouped under one defect identifier. While this seems expedient, valuable information required for defect causal analysis is being lost.

Defect Management Impediments

You might wonder why it is so difficult to get buy-in from most developers and many managers if all the software process models are treating *defects* as useful and even desirable byproducts of development. The problem lies in the definition of the term defect that is found in most reference sources. As a prime example, Webster's Dictionary[3] defines a defect as "a blemish; fault, flaw." It is difficult to see the positive value in such a definition. Can you imagine a teacher of a poorly performing class meeting with the principal and being congratulated for finding all the defects in their student's test papers? Or can you imagine the president of an automobile company who gives bonuses to his engineering staff after they have failed several crash-test categories? We have been programmed to equate defects as failures against the developer, teacher, manufacturer, etc. Is it any wonder that a developer would be reluctant to *admit* finding defects in his or her work product?

"We have been programmed to equate defects as failures against the developer, teacher, manufacturer, etc. Is it any wonder that a developer would be reluctant to admit finding defects in his or her work product?"

Even in the best-case scenario in which developers record defects during peer reviews and testing phases, there are practices employed in the development life cycles that make the use of defects, as a quality measure, questionable. Among these is the correcting of problems and errors prior to the formal peer review processes. Commonly during all life-cycle phases, regular discussions occur between developers and system and test engineers. In many cases, these discussions lead to changes in documentation, design, and/or code. These changes are, in the majority of cases, not formally documented. By the time formal peer reviews for these products are conducted, most of the questionable details have been resolved leading to few if any defects uncovered at the reviews.

From a process quantitative perspective, little can be learned from these reviews. All the pre-review defect information has been lost due to the lack of documentation; defects found at the reviews, which potentially could be used to provide insight into the capability of the review process, are minimal. The statistical process charts tracking these reviews over time may resemble a *flat-line* electrocardiogram with little or no defect variation. Figure 1 shows the results of reviews that have been pre-reviewed without defects being documented. Figure 2 shows a *normal* distribution of defects over time (note: points are almost equally above and below control limit midline).

Testing processes present their own

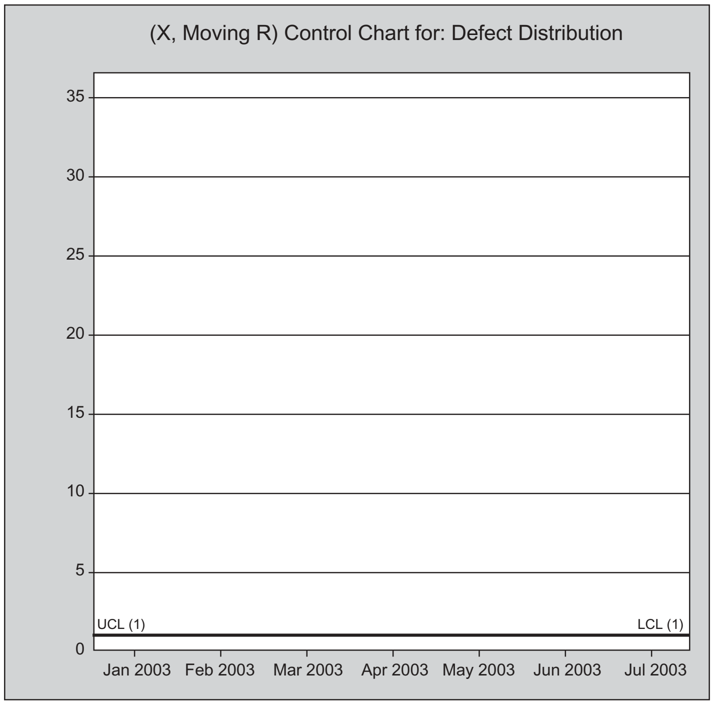

Figure 1: *Distribution of Review Defects with Pre-Review Defects not Documented*

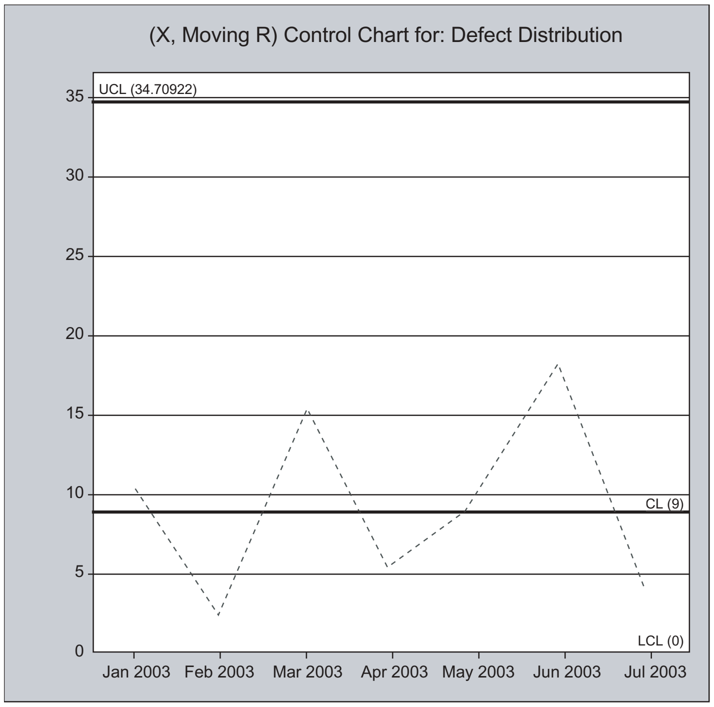

Figure 2: *Normal Distribution of Review Defects*

unique impediments to accurate defect reporting. In the traditional *Waterfall Model* for software development, *unit testing* is the process immediately succeeding *coding*. It has the potential of removing a great deal of defects and uncovering weaknesses in earlier life-cycle processes. Yet, in many organizations, few or no defect measurements are recorded and analyzed. Usually, the only quantitative measures that can be determined are the number of passed and failed unit tests.

Even in organizations that do capture unit test defects, the documentation is vague and usually does not include casual analysis parameters that would pinpoint the life-cycle event that failed to detect the defect that was found during the unit test. Once again the prime culprit for the lack of defect reporting is fear of retribution by the developer or team leader. Unlike other tests such as system-level testing, the individual who created the code normally performs unit testing. To report defects that are found during unit testing is, in their minds at least, an admission of incompetence.

A second but no less common explanation for the lack of defect reporting is *time*. If done diligently, it takes some effort to analyze each defect found, to determine why it was introduced, and to specify which prior process failed to find it.

While both of these explanations are understandable, it still makes the unit test process far less useful as a quality tool. The same problems may be found in the software integration phase in those companies where the developer of the code also writes the integration test procedures and performs the tests. In some organizations, however, independent test groups perform this phase so defect reporting may be more accurate and complete.

While the usefulness of defect management analysis is compromised by the sins of omission, as in the non-reporting of defects illustrated in the previous cases, equally damaging is the reporting of inaccurate defect information. Bad defect reporting leads to the phenomenon sometimes known in the computing world as *garbage in, garbage out*. Simply put, if the data reported is bad, the analysis and resulting conclusions will be wrong as well.

The reasons behind the incorrect reporting of defects may range from expedient reporting of defects on mandatory forms to the misunderstanding of defect definitions (as discussed earlier in this article). In the process-oriented environment found in most defense organizations today, developers are required by their organization's standard defined processes to transcribe defects at various life-cycle milestones. Pressured by schedules and wary of possible management criticism, some developers may choose to fill in the required defect fields with information that does not necessarily reflect the true nature of the problem and/or cause. The resultant analysis may lead to incorrect conclusions about the causes of the defects and thus dilute or eradicate any benefit derived from the process.

Utilizing Defects for Quality and Process Improvement

Up until this point, this article has painted a rather bleak picture of the utility of defects as a quality measure. Typically, metrics such as *peer review defect density* will yield little benefit as a measurement of quality if only small numbers of defects are being reported during the process. However, there are still process improvement opportunities to be gained by determining the root causes[4] of those defects that are reported during the software development life cycles.

If a particular phase seems more prone to producing defects, the related process may have to be reviewed and subsequently updated or changed. This idea can be applied to the examples presented earlier in order to show the possible benefits of finding and analyzing defects. If the school teacher with the poorly performing class uses those negative test results as a basis for changing his or her style or teaching methods to be more in compliance with the students' special needs, the performance of the students may improve. Similarly, if the poor crash-test results in the automobile company cause the improvement of the related quality assurance methods, future test results will most likely be positively affected.

Thus far, only defects found during the software development life-cycle processes have been considered. Once a product has been put under configuration management and delivered to the customer (internal or external company), defects are uncovered during field-testing, system testing, and normal customer usage. These defects are then reviewed, verified, and categorized and submitted back to the development team for resolution. These defects differ in several ways from those discovered during the development life cycle.

First, the reporting reluctance the developer wrestled with during the development life cycle is eliminated. The defect has been found and now must be dealt with. In the same vein, the *time-to-document-the-defect* dilemma has been removed from the developer's plate. Time and budget have been allotted to analyze and fix the defect. Finally, the fear of retribution is, if not eliminated, made less significant. The defect exists and, no matter how it was caused or whose *fault* it was it now must be corrected.

The key benefits of post-development cycle defects are that they can be used as the basis of product and process quality measurements as well as for process improvement identifiers. Quality metrics, such as product defect density, can be calculated from the *post-development cycle* defects and used as a basis of comparison against other projects in order to set product goals and to predict future budgets and schedules. Performing causal analysis on post-development cycle defects will enable the process team to determine the development cycle processes' defect insertion, removal, and leakage rates and make the appropriate process adjustments.

The Team Approach to Improving Defect Management

In an effort to assist developers in distinguishing true defects from action items and to expedite the proper use of techniques such as root-cause analysis and quantitative process management, some organizations have established defect analysis teams (DATs). These teams, usually consisting of members of the organization's Software Engineering Process Group, are tasked with reviewing the defect documentation generated by developers during the various development processes.

DATs examine questionable items such as missing fields, inconsistent or contradictory defect descriptions, or flag other anomalies. Interviews with the appropriate developers are held where the team's findings are discussed. At these meetings, the developer and the team review defect definitions and related causal analysis concepts, and the defect documentation is completed to all participants' satisfaction. At times, even presumably correctly completed documents are discussed with developers in an effort to reinforce the defect management concepts.

In most organizations employing this teaming technique, it is the goal of the DATs to review selected documentation from every appropriate software project in the organization. The employment of DATs helps foster more consistent and complete defect reporting and may act as a catalyst to the institutionalization of defect management techniques throughout the organization.

"The reasons behind the incorrect reporting of defects may range from expedient reporting of defects on mandatory forms to the misunderstanding of defect definitions."

Conclusion

In the best of all possible worlds where no schedules, egos, or calendars exist, defect management would be an extremely useful and informative process in all phases of software development and maintenance. However, due to the reasons mentioned in this article, the utility of defects is diminished during the software development processes. Is this an insurmountable problem with no corrective actions possible? The answer largely depends on factors outside the control of the software developer.

Training must be given and constantly reinforced to change the perception of defect reporting being purely negative and self-indicting. The value of uncovering defects and concepts of causal analysis must be part of this training. Management must also be made aware of the cost and maintenance savings opportunities in defect removal in the earliest possible life-cycle phase. The employment of DATs is an effective way of mentoring and reinforcing defect reporting concepts and processes. Finally, but most importantly, management must assure developers that it supports and encourages defect identification and documentation.

While these suggestions are few, they may look daunting to organizations who have long histories of treating defects purely as problems and to developers who believe that their performance reviews are negatively affected by the number of defects found in their products.◆

Notes

1. This is Capability Maturity Model® nomenclature.
2. This is Capability Maturity Model Integration nomenclature.
3. PSI & Associates Inc. New Webster's Dictionary & Thesaurus. 1991.
4. The root cause is the identified reason for the presence of a defect or problem. The most basic reason, which if eliminated, would prevent recurrence. <www.isixsigma.com/dictionary/Root_Cause>.

About the Author

Raymond Grossman is a senior software process engineer with L-3 Communications-East in Camden, N.J. He is responsible for creating and implementing a metrics program and in helping the organization transition to the Capability Maturity Model® Integration model. Grossman has more than 30 years experience in all phases of software development, including project management. For the past seven years, he has been involved in process improvement and software measurements and metrics, and has been a key player in several Capability Maturity Model (CMM) evaluations. Grossman was part of a team that helped his former organization achieve a CMM Level 5 rating. He has a bachelor's degree in engineering from the City College of New York and has been awarded Certified Professional Management Credentials from James Madison University.

L-3 Communication Systems–East
1 Federal St.
Camden, NJ 08103
Phone: (856) 338-6220
Fax: (856) 338-2425
E-mail: raymond.grossman@L-3com.com