Software Defect Reduction Top 10 List

Barry Boehm, University of Southern California
Victor R. Basili, University of Maryland

Recently, a National Science Foundation grant enabled us to establish the Center for Empirically Based Software Engineering. CeBASE seeks to transform software engineering as much as possible from a fad-based practice to an engineering-based practice through derivation, organization, and dissemination of empirical data on software development and evolution phenomenology. The phrase "as much as possible" reflects the fact that software development must remain a people-intensive and continually changing field. We have found, however, that researchers have established objective and quantitative data, relationships, and predictive models that help software developers avoid predictable pitfalls and improve their ability to predict and control efficient software projects.

Here we describe developments in this area that have taken place since the publication of "Industrial Metrics Top 10 List" in 1987 (B. Boehm, *IEEE Software*, Sept. 1987, pp. 84-85). Given that CeBASE places a high priority on software defect reduction, we think it is fitting to update that earlier article by providing the following Software Defect Reduction Top 10 List.

ONE

Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.

As Boehm observed in 1987, "This insight has been a major driver in focusing industrial software practice on thorough requirements analysis and design, on early verification and validation, and on up-front prototyping and simulation to avoid costly downstream fixes."

For this updated list, we have added the word "often" to reflect additional insights about this observation. One insight shows the cost-escalation factor for small, noncritical software systems to be more like 5:1 than 100:1. This ratio reveals that we can develop such systems more efficiently in a less formal, continuous prototype mode that still emphasizes getting things right early rather than late.

Another insight reveals that good architectural practices can significantly reduce the cost-escalation factor even for large critical systems. Such practices reduce the cost of most fixes by confining them to small, well-encapsulated modules. A good example is the million-line CCPDS-R system described in Walker Royce's book, *Software Project Management* (Addison-Wesley, 1998).

TWO

Current software projects spend about 40 to 50 percent of their effort on avoidable rework.

Such rework consists of effort spent fixing software difficulties that could have been discovered earlier and fixed less expensively or avoided altogether. By implication, then, some effort must consist of "unavoidable rework," an observation that has gained increasing credibility with the growing realization that better user-interactive systems result from *emergent* processes. In such processes, the requirements emerge from prototyping and other multistakeholder-shared learning activities, a departure from traditional *reductionist* processes that stipulate requirements in advance, then reduce them to practice via design and coding. Emergent processes indicate that changes to a system's definition that make it more cost-effective should not be discouraged by classifying them as avoidable defects.

Software's complexity and accelerated development schedules make avoiding defects difficult. These 10 techniques can help reduce the flaws in your code.

Reducing avoidable rework can provide significant improvements in software productivity. In our behavioral analysis of how software cost drivers affected effort for the Cocomo II model (B. Boehm et al., *Software Cost Estimation with Cocomo II*, Prentice Hall, 2000), we found that most of the effort savings generated by improving software process maturity, software architectures, and software risk management came from reductions in avoidable rework.

THREE

About 80 percent of avoidable rework comes from 20 percent of the defects.

That 80 percent value may be lower for smaller systems and higher for very large ones. Two major sources of avoidable rework involve hastily specified

requirements and nominal-case design and development, in which late accommodation of off-nominal requirements causes major architecture, design, and code breakage. A tracking system for software-problem reports that records the effort to fix each defect lets you analyze the data fairly easily to determine and address additional major sources of rework.

FOUR

About 80 percent of the defects come from 20 percent of the modules, and about half the modules are defect free.

Studies from different environments over many years have shown, with amazing consistency, that between 60 and 90 percent of the defects arise from 20 percent of the modules, with a median of about 80 percent. With equal consistency, nearly all defects cluster in about half the modules produced.

Obviously, then, identifying the characteristics of error-prone modules in a particular environment can prove worthwhile. A variety of context-dependent factors contribute to error-proneness. Some factors usually contribute to error-proneness regardless of context, however, including the level of data coupling and cohesion, size, complexity, and the amount of change to reused code.

FIVE

About 90 percent of the downtime comes from, at most, 10 percent of the defects.

Some defects disproportionately affect a system's downtime and reliability. For example, an analysis of the software failure history of nine large IBM software products revealed that about 0.3 percent of the defects accounted for about 90 percent of the downtime. Thus, risk-based testing—including understanding a system's operational profiles and emphasizing testing of high-risk scenarios—is clearly cost-effective.

SIX

Peer reviews catch 60 percent of the defects.

Given that finding and fixing most defects earlier in the project development cycle is more cost-effective than finding them later, we seek techniques that find defects as early as possible. Numerous studies confirm that peer review provides an effective technique that catches from 31 to 93 percent of the defects, with a median of around 60 percent. Thus, the 60 percent value cited in the 1987 column remains a reasonable estimate.

Peer reviews, analysis tools, and testing catch different classes of defects at different points in the development cycle.

Factors affecting the percentage of defects caught include the number and type of peer reviews performed, the size and complexity of the system, and the frequency of defects better caught by execution, such as concurrency and algorithm defects. Our studies have provided evidence that peer reviews, analysis tools, and testing catch different classes of defects at different points in the development cycle. We need further empirical research to help choose the best mixed strategy for defect-reduction investments.

SEVEN

Perspective-based reviews catch 35 percent more defects than nondirected reviews.

A scenario-based reading technique (V.R. Basili, "Evolving and Packaging Reading Technologies," *J. Systems and Software*, vol. 38, no. 1, 1997, pp. 3-12) offers a set of formal procedures for defect detection based on varying perspectives. The union of several perspectives into a single inspection offers broad yet focused coverage of the document being reviewed. This approach seeks to generate focused techniques aimed at specific defect-detection goals by taking advantage of an organization's existing defect history.

Scenario-based reading techniques have been applied in requirements, object-oriented design, and user interface inspections. Improvements in fault detection rates vary from 15 to 50 percent. Further, focused reading techniques facilitate training of inexperienced personnel, improve communication about the process, and foster continuous improvement.

EIGHT

Disciplined personal practices can reduce defect introduction rates by up to 75 percent.

Several disciplined personal processes have been introduced into practice. These include Harlan Mills's Cleanroom software development process and Watts Humphrey's Personal Software Process (PSP).

Data from the use of Cleanroom at NASA have shown 25 to 75 percent reductions in failure rates during testing. Use of Cleanroom also showed a reduction in rework effort so that only 5 percent of the fixes took more than an hour, whereas the standard process caused more than 60 percent of the fixes to take that long.

PSP's strong focus on root-cause analysis of an individual's software defects and overruns, and on developing personal checklists and practices to avoid future recurrence, has significantly reduced personal defect rates. Developers frequently enjoy defect reductions of 10:1 between exercises 1 and 10 in the PSP training course.

Effects at the project level are more scattered. They depend on factors such as the organization's existing software maturity level and the staff's and organization's willingness to operate within a highly structured software culture. When you couple PSP with the strongly compatible Team Software Process (TSP), defect reduction rates can soar to factors of 10 or higher for an organization that operates at a modest maturity level. Results tend to be less spectacular if the organization already employs highly mature processes.

The June 2000 special issue of *CrossTalk*, "Keeping Time with PSP and TSP," offers a good set of relevant discussions, including experience showing that adding PSP and TSP to a CMM Level 5 organization reduced acceptance test defects by about 50 percent overall, and reduced high-priority defects by about 75 percent.

NINE

All other things being equal, it costs 50 percent more per source instruction to develop high-dependability software products than to develop low-dependability software products. However, the investment is more than worth it if the project involves significant operations and maintenance costs.

The analysis of 161 project data points for the Cocomo II model resulted in an added cost of 53 percent for its "required reliability" factor, while normalizing for the effects of 22 other factors. Does this mean that Philip Crosby's landmark book, *Quality Is Free* (Mentor, 1980), had it all wrong? Maybe for some low-criticality, short-lifetime software, but not for the most important cases.

First, in the Cocomo II maintenance model, low-dependability software costs about 50 percent per instruction more to maintain than to develop, whereas high-dependability software costs about 15 percent less to maintain than to develop. For a typical life-cycle cost distribution of 30 percent development and 70 percent maintenance, low-dependability software becomes about the same in cost per instruction as high-dependability software—again, assuming all other factors are equal.

Second, in the Cocomo II-related quality model, high-dependability software removes about four times as many defects as average-dependability software, which in turn removes about four times as many defects as low-dependability software. For example, consider an average-dependability system such as a commercial billing system, in which the operational cost of software defects—due to lost worker time, lost sales, added customer service costs, litigation costs, loss of repeat business, and so on—roughly equals life-cycle software development and maintenance costs. For such a system, the increased defect rate of using low-dependability software would make its ownership costs roughly three times higher than the ownership costs of high-dependability software.

TEN

About 40 to 50 percent of user programs contain nontrivial defects.

A 1987 study in this area (P.S. Brown and J.D. Gould, "An Experimental Study of People Creating Spreadsheets," *ACM Trans. Office Info. Sys.*, July 1987, pp. 258-272) found that 44 percent of 27 spreadsheet programs produced by experienced spreadsheet developers contained nontrivial defects—mostly errors in spreadsheet formulas. Yet the developers felt confident that they had produced accurate spreadsheets.

The creators of Web-programming facilities face the challenge of providing their tools with the equivalent of seat belts and air bags, along with safe-driving aids and rules of the road.

Subsequent laboratory experiments have reported defective spreadsheet rates between 35 and 90 percent. The analysis of operational spreadsheets reveals defect rates between 21 and 26 percent; the lower rates probably stem from corrections already made during operation.

Now, and increasingly in the future, user programs will escalate from spreadsheets to Web-scripting languages capable of sending agents into cyberspace to make deals for you. The ranks of "sorcerer's apprentice" user-programmers will also swell rapidly, giving many who have little training or expertise in how to avoid or detect high-risk defects tremendous power to create high-risk defects. One study for the Cocomo II book estimated that the US will have 55 million user-programmers by 2005. If we classify active Web-page developers as user-programmers, this prediction appears to be on track.

Thus, the creators of Web-programming facilities face the challenge of providing their tools with the equivalent of seat belts and air bags, along with safe-driving aids and rules of the road. This software engineering research challenge is one of several identified by a National Science Foundation study, "Gaining Intellectual Control of Software Development," which we recently summarized in *Computer* (May 2000, pp. 27-33).

Surely, our list can benefit from refinement and further empirical research on defect reduction. Much of the data we have reported, for example, fails to account for the interaction between many of the variables that, if known, could provide answers to questions like:

- If I invest in peer reviewing, Cleanroom, and PSP, am I paying for the same defects to be removed three times?
- How much testing would this investment enable me to avoid?

We hope to involve the software community in expanding the Software Defect Reduction Top 10 List and other currently available data into a continually evolving, open source, Web-accessible handbook of empirical results on software-defect reduction strategies. We also plan to initiate counterpart handbooks for commercial off-the-shelf systems and other emerging software areas. We welcome your participation in this effort and urge you to visit the CeBASE Web site at http://www.cebase.org for further information. You can also find an expanded version of this column at http://www.cebase.org/defectreduction/ top10. ✸

Barry Boehm *is director of the University of Southern California Center for Software Engineering. Contact him at boehm@sunset.usc.edu.*

Victor R. Basili *is a professor in the Institute for Advanced Computer Studies and the Computer Science Department at the University of Maryland. Contact him at basili@cs.umd.edu.*