

## A little quality time

Vincent Shen,  
*QualityTime Editor*

The goals for all software-engineering research are improvements in productivity and quality.\* Proposals for new tools, methods, and environments all include claims along these lines. But what is productivity? What is quality? What are the factors that influence them, and to what degree? Our field is apparently not yet mature enough to provide answers to these important questions. Software engineering does not yet have the precise and reliable measurements that other traditional engineering disciplines take for granted.

*IEEE Software* has introduced this department to discuss current productivity and quality issues. The purpose is to promote a dialogue between researchers and practitioners. The software-engineering community must be aware of the needs for productivity and quality measurements and the dangers of not knowing the inherent risks.

For readers not familiar with this area, let me provide some historical background. Software metrics were not of much interest in the 1960s when software cost was only a small part of the total cost of computer systems. The size of a computer program was determined simply by the number of punched cards it took to contain the program. This metric (which was almost a weight measure for people who had to carry the programs) is basically equivalent to today's lines-of-code metric. Programmer productivity can be measured by the number of lines written (or debugged) during a month's time. Program quality can be measured by the number of bugs found in a thousand lines of code.

As the proportion of software cost to total computing cost increased drastically in the 1970s, so did the interest in better metrics. Token counts were introduced to increase the precision of size

measures for programs written in the period's higher-level languages that permitted free-format coding.

Programming models were introduced that contained various assumptions about *complexity* — the time to produce, to understand, or to debug computer programs is a mathematical function of the counts of various tokens. Unfortunately, a series of empirical studies conducted in the late 1970s failed to offer concrete evidence that any of these complexity measures was significantly better than the classic lines-of-code measure.

Metrics researchers in the 1980s are generally less optimistic than their colleagues in the 1970s. Even though the pressure to find better metrics is greater because of the greater cost of software, fewer people today are trying to formulate combinations of complex-

ity metrics so that they relate to some definition of productivity and quality.

Instead, they set very narrow goals and show whether these goals are reached using focused metrics. For example, producing higher quality software is a general goal. One corresponding narrow goal would be to test the software thoroughly. An appropriate metric might be some measure of test coverage. The measurement results can be an effective guide for the testing process.

Although metrics research is not a panacea for weaknesses in software development, its evolution has produced many benefits for the industry. The accompanying report gives examples of such benefits. The specific numbers came from the experiences of many practitioners, but they may vary in your organization.

## Industrial software metrics top 10 list

Barry Boehm, *TRW, Inc.*

I am always fascinated by top 10 lists. So, when Vincent Shen asked me to write a piece for this department, I decided to present my candidate top 10 list of software metric relationships, in terms of their value in industrial situations. Here they are, in rough priority order:

1. *Finding and fixing a software problem after delivery is 100 times more expensive than finding and fixing it during the requirements and early design phases.*

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.

2. *You can compress a software development schedule up to 25 percent of nominal, but no more.*

There is a remarkably consistent cube-root relationship for the most effective schedule  $T_{dev}$  for a single-increment, industrial-grade software development project:  $T_{dev} = 2.5 \times MM^{1/3}$  where  $T_{dev}$  is in months and  $MM$  is the required development in man-months.

Equally remarkable is the fact that virtually no industrial-grade projects have been able to compress this schedule more than 25 percent. Thus, if your project is estimated to require  $512MM$ , your best schedule is  $2.5 \times 512^{1/3}$ , or 20, months. If your boss or customer wants the product in 15 months, you will barely make it if you add some extra resources and plan well. If he wants it in 12 months, you should gracefully but firmly suggest reducing the scope or doing an incremental development.

3. *For every dollar you spend on software development you will spend two dollars on software maintenance.*

A lot of industry and government organizations created major maintenance embarrassments before they realized this and instituted thorough software life-cycle planning. This insight has also stimulated a healthy emphasis on developing high-quality software products to reduce maintenance costs.

4. *Software development and maintenance costs are primarily a function of the number of source instructions in the product.*

This was the major stimulus for

\*See, for example, *Encyclopedia of Computer Science and Engineering*, A. Ralston and E.D. Reilly, eds. (Van Nostrand Reinhold, New York, 1983).

migrating from assembly languages to higher order languages. It is now a major stimulus for developing and using very high-level languages and fourth-generation languages to reduce software costs.

5. *Variations between people account for the biggest differences in software productivity.*

Studies of large projects have shown that 90th-percentile teams of software people typically outproduce 15th-percentile teams by factors of four to five. Studies of individual programmers have shown productivity ranges of up to 26:1. The moral: Do everything you can to get the best people working on *your* project.

6. *The overall ratio of computer software to hardware costs has gone from 15:85 in 1955 to 85:15 in 1985, and it is still growing.*

This relationship has done more than anything else to focus management attention and resources on improving the software process.

7. *Only about 15 percent of software product-development effort is devoted to programming.*

In the early days, there was a 40-20-40 rule: 40 percent of the development effort for analysis and design, 20 percent for programming, and 40 percent for integration and test. Now, the best project practices achieve a 60-15-25 distribution. Overall, this relationship has been very effective in getting industrial practice to treat software product development as more than just programming.

8. *Software systems and software products each typically cost three times as much per instruction to fully develop as does an individual software program. Software-system products cost nine times as much.*

A software system contains many software modules written by different people. A software-system product is

such a system that is released for external use. The discovery of this cost-tripling relationship has saved many people from unrealistically extrapolating their personal programming productivity experience into unachievable budgets and schedules for software-system products.

9. *Walkthroughs catch 60 percent of the errors.*

The structured walkthrough (software inspection) has been the most cost-effective technique to date for eliminating software errors. It also has significant side benefits in team building and in ensuring backup knowledge if a designer or programmer leaves the project.

I had a hard time picking number 10. I ended up with a composite choice:

10. *Many software phenomena follow a Pareto distribution: 80 percent of the contribution comes from 20 percent of the contributors.*

Knowing this can help a project focus on the 20 percent of the subset that provides 80 percent of the leverage for improvement. Some examples:

- 20 percent of the modules contribute 80 percent of the cost,
- 20 percent of the modules contribute 80 percent of the errors (not necessarily the same ones),
- 20 percent of the errors consume 80 percent of the cost to fix,
- 20 percent of the modules consume 80 percent of the execution time, and
- 20 percent of the tools experience 80 percent of the tool usage.

I think it has been a strong credit to the software metrics field that it has been able to determine and corroborate these and many other useful software metric relationships. And there are many useful new ones coming along. I look forward to reading about them in this department.

*Continued from p. 5*

Science and Engineering), and research in our field is severely underfunded.

Although a liberal citation policy cannot solve such broad problems by itself, encouraging authors to cite related, historically significant research will help break down the narrow perspectives to which many of us are bound. In addition, it will encourage scholarship and intellectual honesty among authors. Also, having comprehensive bibliographies will greatly help those who want to learn more about the subject in an article.

It seems to me that increasing *IEEE Software's* citation limit could yield significant benefits to authors, readers, and our field as a whole.

Joseph A. Giguen  
Program Manager  
Computer Science Laboratory  
SRI International  
Menlo Park, Calif.

## Comments misread

*To the editor:*

Douglas Schuler ("Real-World Coverage," Letters, July, p. 5) has misread my comment on SDI coverage ("Wasted Space?" Letters, May, p. 3). I merely proposed that the space normally devoted to the umpteenth rehash of arguments, whether pro or con, be severely reduced. Never did I ask that "topics be excluded" or suggest that anyone "disregard real-world implications."

My point was that these important issues could indeed be covered, minus the customary hot air, in about a third the space normally used. A perusal of the letter from Frederick Seitz ("SDI Advocate's View," Letters, July, p. 5) confirms this figure.

If I exist "in a vacuum," that analogy should be extended to say that many others have their heads in pressure cookers — they are always steamed and can't seem to get out anything intelligible.

As soon as *new* information on SDI is reported, I will be the first to acknowledge its newsworthiness.

John D. Wolf  
Programmer  
SCS, Inc.  
Fort Wayne, Ind.

## Share your experience

Whether your experience tends to confirm or refute the opinions expressed here, we encourage you to share that experience with us. We welcome short submissions (less than 750 words) that cover practical experiences, factors influencing productivity and quality, attempts to overcome some of the weaknesses in certain metrics, risks in using imperfect measures, and the like. We welcome both specific and general discussions of productivity and quality.

We welcome your letters. Send them to Letters Editor, *IEEE Software*, 10662 Los Vaqueros Cir., Los Alamitos, CA 90720. All submissions are subject to editing for style, length, and clarity.