INDUSTRIAL SOFTWARE METRICS: A TOP-TEN LIST

Barry W. Boehm, TRW Inc.

I am always fascinated by Top-Ten lists. So, when Vincent Shen asked me to write this column, I thought I would present my candidate Top-Ten 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 focussing 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% 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:

$$T_{dev} = 2.5 \sqrt[3]{MM},$$

where T_{dev} is in months and MM is the required development man-months. Equally remarkable is the fact that virtually no industrial-grade projects have been able to compress this schedule more than 25%. Thus, if your project is estimated to require 512MM, your best schedule is $2.5 \sqrt[3]{512} = 20$ 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 in order 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 migrating from assembly languages to higher-order languages. It is now a major stimulus for developing and using Very High Level Languages or 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 4 to 5. 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 toward improving the software process.

7. Only about 15% of software product development effort is devoted to programming. In the early days, there was a "40-20-40" rule: 40% of the development effort for analysis and design; 20% for programming; 40% for integration and test. Nowadays, 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 3 times as much per instruction to fully develop as does an individual software program. Software system products cost 9 times as much. This 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% of the errors. The structured walkthrough or software inspection has been the most cost-effective technique to date for eliminating software errors. In addition, it has significant side benefits in teambuilding 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% of the contribution comes from 20% of the contributors. Knowing this can help a project focus on the 20% subset providing 80% of the leverage for improvement. Some examples:

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

To summarize, 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 column.