Slow Down, You Read Too Fast

Steve Wartik
Software Productivity Consortium
Reston, VA

wartik@Software.ORG

Reading two papers in recent issues of IEEE Software and IEEE Computer, I was once again saddened to observe that few software engineering researchers who write comparative analyses really know the research against which they compare their own. Each paper presented work with which I am both partially familiar -- meaning I've read papers on the topic and perhaps tried it on toy examples – and intimately familiar, meaning I've applied it over a period of years in developing real software. In both papers, I reacted to the partially familiar research with tacit agreement, thinking the authors made good points.

But I reacted differently to the intimately familiar research as I saw how the authors had misinterpreted it. Some errors were simple and forgivable. Most, unfortunately, were more severe, as if the authors were only superficially familiar with the work. I had to question if the authors had done more than read the papers they cited. Certainly they could not have interpreted the work as they did if they had tried it in real software development, or had even dug deeply into the references.

Now this isn't an indictment of the authors, who are justly respected members of the software engineering community. Furthermore, my tacit agreement with the topics partially familiar to me strongly suggests I'm no more qualified than them or the next person to compare and contrast others' work with my own.

This natural occurrence in a field like software engineering, where so much is subjective and opinions are more prevalent than data, leads to nothing if not entertainment, in the form of countless vituperative conference panels, journal articles, and Internet flame wars. Thus we have reached a state where, as a colleague of mine so aptly put it, "Every software engineering researcher thinks everyone else is an idiot, and 99% of them are right." Pick any area of software engineering, any life cycle phase, any technique, any tool, and you'll find hordes of people loudly voicing their opinions on it at the expense of somebody else. Empirical studies, that cornerstone of advancement in the hard science and engineering disciplines, fare little better in software engineering. Their data and assumptions are hotly debated, and (with a few exceptions) they aren't the foundation of much future research. Beizer goes so far as to doubt the possibility of controlled experiments comparing software methods [1]. Right or wrong, his paper shows how little agreement there is as to how to make progress in the field.

On the one hand, I sometimes think it doesn't matter. My observations and experiences make me believe that any method or tool, conscientiously and consistently applied, will yield impressive productivity gains when compared with using no method or tool, which unfortunately still characterizes many companies today. On the other hand, I worry that our misunderstandings of others' work is causing us to miss their best insights. We make mistakes in software development when we miss the details, and missing the details is just what we all seem to have done.

I see an obvious solution to this problem. Everyone must abide by the Commandment of Comparative Publishing, which is:

> You cannot publish a comparison of your work with others' work unless you are intimately familiar (as defined above) with their work.

In other words, every researcher should learn a method by applying it on one or more – preferably more – realistic examples. I'd guess that adopting this solution would cause most people to see that what they thought were benefits of their research are either debatably so or of minuscule concern. It's a simple principle, and its advantages are clear.

But so are its disadvantages. I cannot imagine this solution ever becoming popular. Software engineering is still enough of a craft that learning to apply a software development method – or even a tool – can take years. If every researcher took the time to learn several methods, industrial research labs would cease to produce results. Professors would have to suspend research activity. They also couldn't conduct studies using Master's degree candidates, who aren't around long enough to be useful. Ph.D.'s are, but learning additional methods would require a commitment that would have them reaching for hammers. Even when someone is intimately familiar with a method, they can't claim to have developed realistic software until that software has been used for a while, giving people a chance to discover and fix bugs (i.e., measuring the research's utility in realistic settings). That in turn implies the need to develop user manuals, training materials, and all those other products that separate toys from reality. If everyone adopted the Commandment of Comparative Publishing, software engineering research would cease for the better part of a decade. I do not claim such a hiatus would be productive. Craft-oriented disciplines grow by innovation.

I can see a more plausible but equally unpalatable solution. Editors could require that any paper claiming an advantage over X be refereed by the person responsible for X. The benefits to the quality and value of published papers are clear. On the down side, no sane person would want to be an editor. Tenure standards would need revision – one journal paper would have to be sufficient. The Internet would quickly jam up with "Does" – "Does not" – "Does so" messages. And finally, anyone unfortunate enough to do excellent research would find themselves deluged with papers to review.

The simplest solution is perhaps the best one. Let's skip subjective comparisons altogether. Papers should reference related research but should omit statements like, "Our work satisfies the following need that X does not address." When

these statements aren't just plain inaccurate, they often express only a personal software development philosophy. The differences between related research projects are seldom as striking as their proponents would have us believe.

Since my most-cited paper compares a domain analysis approach with which I am intimately familiar to several with which I am partially familiar [2], you may detect a certain irony. Nolo contendere. I admitted above that I'm probably as guilty of misinterpretation and oversimplification as everyone else. In writing [1], I took the precaution of consulting the creators of the other approaches prior to publishing, but after several years of reflection I can't help but wonder if I didn't overstate my conclusions.

And I still feel the problem acutely. Those of us with graduate degrees in Computer Science – that is, most researchers – were trained to learn by reading a paper or a book. That works nicely for theoretical material. Software development methods, CASE tools, and topics of that ilk cannot be easily encapsulated in a journal paper. To be learned, they must be tried. To be appreciated, they must be used realistically. Until that happens, we must view our comparisons of others' work with a regrettable skepticism.

REFERENCES

[1] Beizer, Boris. The Cleanroom Process Model – Caution Advised. Journal of the Society for Software Quality (March/April 1995), pp. 1-23.

[2] Wartik, S. and R. Prieto-Diaz. Criteria for Comparing Reuse-Oriented Domain Analysis Approaches. International Journal of Software Engineering and Knowledge Engineering 2:3 (September 1992), pp. 403-431.

Is "Software Quality" Intrinsic, Subjective, or Relational?

Dick Carey
20 Riverside St. #22
Watertown, MA 02172-2670
(617) 926-2556

carey@dma.isg.mot.com

The software community almost unanimously measures Software Quality (SQ) as the ratio of Errors per Kilo Lines Of Code (E/KLOC). In spite of this, the underlying assumptions for this concept and its application have not been critically examined enough. In this paper, it is asserted that E/KLOC is not a reasonable measure of SQ, and a different model is introduced.

An important requirement for objectivity is that judgements be quantifiable. How we define SQ has consequences in determining a software organization's goals. I classify three different interpretations of SQ in terms of what Chris Sciabarra [Sci] calls Ayn Rand's dialectic of the Intrinsic, the Subjective, and the Relational.

INTRINSIC-SQ

Intrinsic-SQ says quality is only a measurement of the current code, independent of past changes. A large program is going to have more errors than a small program, so in order to judge the density of errors, Intrinsic-SQ requires us to know the number of Lines of Code (LOC). The E/KLOC formula makes errors and LOC dependent on each other. A big fraction is bad, a small fraction is good. Program changes that decrease the ratio are seen as improvements.

This satisfies our need to have an objective measure of SQ but is it correct? The E/KLOC formula says that if we don't fix any errors but we add code my SQ improves (because I've increased the denominator). If we take 1000 lines of unstructured code and modularize it so it shrinks to 1/10th its original size with the same functionality, I've improved its SQ 10 times. But E/KLOC says I've worsened its Intrinsic-SQ 10 times from 10/1000 to 10/100 (because I've decreased the denominator).

E/KLOC says there's a trade-off between error count and code size. There's a point where if we remove enough code in order to fix a error it won't improve its Intrinsic-SQ at all:

```
                PREVIOUS VERSION:     CURRENT VERSION:

                10 errors             9 errors
E/KLOC    =     ----------      =     ---------
                100 LOC               x LOC

                10 x            =     900

                   x            =     90
```

therefore

```
                10 errors             9 errors
E/KLOC    =     ----------      =     ---------
                100 LOC               90 LOC
```

If we fix 1 error by removing 10 lines of code E/KLOC says the Intrinsic-SQ hasn't changed. We could almost make the opposite case that a large E/KLOC is good.

If we fix a error by adding code has SQ improved? Yes. If we fix a error by removing code has SQ improved? Yes. If we fix a error without adding or removing code has SQ improved? Yes. The use of E/KLOC is at odds with the original objective of improving SQ. These weaknesses of E/KLOC make it a very inaccurate model of true SQ. It's wrong to accuse people of "fooling" the E/KLOC metric. The problem is that the E/KLOC metric "mismeasures" true SQ. It is the underlying false assumption that SQ is intrinsic that causes us to make LOC and errors dependent variables.

SUBJECTIVE-SQ

Most software organizations apply externally-generated meth-