

Sorting Out Software Complexity

Underlying complexity escalates exponentially: some little-known research findings.

There has been, over the years, a raging controversy about how complex a task software construction is. Some people say it is easy, that it can be automated, that error-free software is possible, that the solution to whatever “software crisis” exists is right around the corner. Others say it is extremely difficult, perhaps even the most complex task ever undertaken by humanity, and that any solution to the “software crisis” is as elusive as a silver bullet.

On the former side is a curious mixture of academic researchers, vendors, and a few practitioners, all of whom usually have something to gain if you believe in their point of view. You buy their products, or their training, or you support their research.

On the latter side are most practitioners, a very few vendors, and some leading academics who have done real software work while pursuing their software theory development. People such

as David Parnas, emerging from his work on naval and nuclear systems, and Fred Brooks, from his work on the earliest, most massive (at the time) system

development task, the OS/360 system.

Elsewhere, I have talked about my own research attempts intended to move toward resolution of this raging controversy [1]. In those studies, I examined the degree to which software work was clerical and the degree to which it was intellectual,

obtaining findings that came down solidly on the intellectual—and to some extent, even creative—side. Those findings, I would assert, support the view of those who find software work to be extremely difficult.

That’s not what I want to write about here, however. In this column, I want to talk a little more specifically about how the complexity of software manifests itself. To do that, I want to introduce a couple of little-known, but enormously relevant, research findings.

The first of those findings is this: *For every 25% increase in problem complexity, there is a 100% increase in complexity of the software*

solution. This is one of my favorite research findings because it is so little known, so compellingly important, and so clear in its explanation. It says, oh-so-succinctly, that the difficulty of solving a problem in software grows exponentially. The more difficult the problem, the more difficult its solution. No wonder

Complexity, I would assert, is the biggest factor involved in anything having to do with the software field. It is explosive, far reaching, and massive in its scope.

so much of the software literature is devoted to “scaling up” issues, and “programming in the large.”

The second of those findings is this: *Explicit requirements explode by a factor of 50 or more into implicit (design) requirements as a software solution proceeds.*

This is another of my favorite research findings. It is also little known, and it speaks of the speed with which complexity consumes the software solution process. As we move from requirements specification to design, this finding tells us, the creation of a design solution compels us to add design requirements to the original problem requirements (that is, to solve this problem we must not only code things to meet those original requirements, but these evolved design requirements as well). And there may be 50 times as many of those implied, derived, requirements as we started out with only a life-cycle phase ago.

I’ll briefly discuss the sources of these research findings later; for now, let’s pause quickly to cogitate on what these findings tell us. To me, they virtually annihilate the thought that software development is easy, automatable, and something that can be performed error-free. Very early in the life cycle, that exponential increase in solution com-

plexity gets under way. Very early in the life cycle, what may have seemed to be a fairly simple problem—or may not have—has suddenly exploded into something else entirely. And that evolving complexity never looks back. By the time we get to the part of the life cycle where error-removal becomes the main focus, we have reached the point where virtually no combination of error-removal approaches can guarantee the software product will never fail.

This highly explosive complexity goes a long way toward reinforcing some other important thoughts on software. Why do top-quality people matter? Because it takes considerable intelligence and skill to overcome complexity. Why does process matter? Because it’s one way of coping with complexity.

Why is estimation so difficult? Because our solutions are so much more complicated than our problems appear to be. Why is reuse-in-the-large so elusive? Because complexity magnifies the solution diversity that limits the value of large-scale reuse.

Why are there so many different correct approaches to designing the solution to a problem? Because the solution space is so complex. Why do the best designers use iterative, heuristic approaches? Because there are sel-

dom any simple and obvious design solutions, and even more seldom any optimum ones.

Why is 100% path or branch test coverage rarely possible and, in any case, insufficient? Because of the enormous number of paths in those complex solutions to most problems, and because software complexity leads to errors that complete coverage cannot expose. Why are inspections the most effective and efficient error-removal approach? Because it takes a human to filter through all that complexity in order to spot errors.

Why is software maintenance such a time-consumer? Because it is seldom possible to determine, at the outset, all the ramifications of a problem solution. Why is “understanding the existing product” the most dominant and difficult task of software maintenance? Because there are so many possible correct solution approaches to solving any one problem.

Why does software have so many errors? Because it is so difficult to get it right the first—or even the Nth—time.

There. Complexity, I would assert, is the biggest factor involved in anything having to do with the software field. It is explosive, far reaching, and massive in its scope. We work within that complexity, managing to

build software products that for the most part are amazingly reliable and magnificently useful. But it's far from easy.

Now, let's talk about the (admittedly obscure) sources for these findings. That 25% growing to 100% finding comes from a paper published in *IEEE Transactions on Software Engineering*. But hardly anyone seems to remember either the paper or its finding. In the course of preparing a book on the fundamental facts of software engineering recently, I sought to find the origin of this particular finding (I had forgotten its origin myself over the years!) Almost no one I talked to, including some of the top names in the software field, remembered it. It was only after I took a guess at the journal it was published in, and the approximate year of its publication, that someone found it for me [3].

That factor of 50 requirements explosion was even more difficult to track down. I distinctly remember a speaker at a conference I organized stating this was a by-product finding of some work he was doing, and as a result of that: I documented it in a newsletter article I wrote at the time; and, I included it in a couple of books on software quality that I wrote (for example, [2]). But when it came time to track down that original source for this new book, I contacted the person whom I remembered making the statement, and he

could not provide me with any citation for the finding.

I often tell people that "my head is in the theory of software engineering, but my heart is in its practice." My own experience as a software developer prepared me to accept these explosive findings on complexity wholeheartedly; I realize some readers may consider these findings difficult to accept. I think the exponential finding is pretty ironclad, even if no one seems to remember it. But I don't blame you if you're skeptical about that factor of 50 finding, since I can't track it back to an original source. Still, if you believe either of these findings, the other seems to follow almost automatically. In a very nice way, these are research findings that support one another. Even if, as I suspect, neither researcher was aware of the work of the other! **■**

REFERENCES AND COMMENTS

1. Glass, R.L. *Software Creativity*. Prentice-Hall, 1995. See section 2.6, "Intellectual vs. Clerical Tasks," for research findings about the difficulty of solving problems in software.
2. Glass, R.L. *Building Quality Software*. Prentice-Hall, 1992. A reference to the "50 times" finding appears on page 55.
3. Woodfield, S.N. An experiment on unit increase in problem complexity. *IEEE Transactions on Software Engineering* (Mar. 1979). The source of the 25% →100% finding.

ROBERT L. GLASS (rlglass@acm.org) is the publisher of the *Software Practitioner* newsletter and editor emeritus of *Elsevier's Journal of Systems and Software*.

© 2002 ACM 0002-0782/02/1100 \$5.00

Coming Next Year in Communications

JANUARY

Digital Government

FEBRUARY

Peer-to-Peer Computing

MARCH

Attentive User Interfaces

APRIL

Digital Rights Management

MAY

Wireless Networking
Security

JUNE

Marketing Digital Products

JULY

A Game in Every Application

AUGUST

Program Compaction

SEPTEMBER

Mathematical Underpinnings
of CS

OCTOBER

Service-Oriented Computing

NOVEMBER

The Future of Infostructure
Networking

DECEMBER

Mobile E-Commerce