Editor in Chief: Steve McConnell ■ Construx Software ■ software@construx.com

Common Sense

Steve McConnell

"I like your columns, but they're really all just common sense," a client told me. He didn't have a software background, and my columns were his first introduction to systematic ways of understanding software projects. To my chagrin, the net effect of a well-written column appeared to be that he thought software engineering was trivial!

The idea that good software engineering "is all just common sense" is one of my pet peeves. If good software engineering were really all just common sense, we should expect to see projects routinely meet their schedule and budget targets and delight their end users. I shouldn't receive emails every week that describe projects that seem sensible initially but end up failing dramatically.

A catalog of proverbs

How much does common sense tell us about software engineering? Let's look at several familiar proverbs and see how well they apply to software.

Let sleeping dogs lie

This bit of common sense implies that issues that aren't currently causing problems for a software project should be left alone. Wait until the issues become problems, and then worry about them. Don't waste time worrying about them prematurely.

How true is this common sense when applied to software projects? A key success factor in software project management is active risk management, in which we proactively search out sleeping dogs, poke and prod them until they wake up, and then deliberately try to move them out of the project's way. Failure to actively manage risks is one of the biggest contributors to software project failure.[1] Conversely, many authorities cite active risk management as one of the most significant contributors to software project success.[2,3] For example, the Software Project Manager's Network puts it first on its list of 16 critical software practices.[4] So "let sleeping dogs lie" turns out to be 180 degrees wrong. The American sage Benjamin Franklin assessed the software situation better 200 years ago when he said, "A little neglect may breed great mischief."

Two heads are better than one

This is an interesting adage if only because it competes with another aphorism, "Too many cooks spoil the broth." Which is right in our world? In the software engineering classic *The Mythical Man-Month*, Fred Brooks argues that one of the most difficult challenges on large software projects is maintaining a software design's conceptual integrity. Brooks proposes a Surgical Team–Chief Programmer Team programming model, one of the main benefits of which is that most design decisions are made by one mind—the mind of the chief programmer. In this context, two heads might not be better than one; too many cooks might indeed spoil the broth.

On the other hand, one of the most effective software engineering techniques is formal inspection, which is based on the idea that a reviewer can detect problems that the original author of a work product will overlook. Inspections are one of the techniques that have indisputably proven their worth in practice,[5] so these two commonsense sayings would seem to be both right, and both wrong—depending on the context.

IEEE Software

An ounce of prevention is worth a pound of cure

The average project spends 40 to 80 percent of its total effort fixing defects. Software industry research has generally found that upstream defect-prevention work pays big dividends in avoiding downstream rework. As I described in my column in the previous issue of *IEEE Software*, however, there are limits to how much prevention any project can stand. One contribution of the agile-programming movement is to cast a critical eye toward prevention and ensure that projects do not spend more on prevention than they would need to spend on a cure. On balance, I think this commonsense maxim does apply to software, but we need to be careful not to overextend it—that is, "moderation in all things."

He who hesitates is lost

The point of this maxim is to instill a bias toward action. In software development, a project's success or failure is often determined within the first 10 to 20 percent of its duration.[6] Occasionally, software projects become mired in analysis paralysis—debating endless design alternatives, gold-plating project plans, and so on. Regrettably, though, more often they tend not toward analysis paralysis but toward underplanning.[1] So, counter to this commonsense adage, hesitating (or planning) is a sign of a healthy, well-run project. In this case, what is actually common sense is unclear, because we again have dueling maxims. For software, I prefer "look before you leap" or, drawing on Benjamin Franklin again, "failure to prepare is preparing to fail."

Spare the rod and spoil the child

This saying about disciplining children is based on the belief that strictly enforcing rules will produce well-behaved children; leniency will produce spoiled, disobedient children. In software, we commonly see managers, customers, and marketing departments who seem to take this proverb to heart. They believe that coercing their development teams into working longer hours will produce the software in less time. The highbrow management version of this old chestnut is Parkinson's law, "work expands to fill available time," which I see used as justification for numerous coercive and manipulative management practices.

Here again, the commonsense guideline is diametrically opposed to the software reality. Software design and construction are highly contemplative, internal activities. A developer must be highly motivated to be able to do software development work at all. One of the most basic insights of motivation research is that when a person tries to apply external motivation to someone who is already highly internally motivated, internal motivation decreases. So, the net effect of "using the rod" is a reduction in internal motivation, and the effect on productivity is a net loss.[7]

A final pearl

On the basis of these examples, common sense appears to not be a good foundation for practicing software engineering. If you think otherwise, you will likely run afoul of another pearl from Benjamin Franklin: "Life's tragedy is that we get old too soon and wise too late."

Making it look easy

With all these counterexamples, why do intelligent people continue to think good software engineering practices are just common sense?

After pondering this question for the past 10 years, I have concluded that the phenomenon is related to the notion of conditional probability. Conditional probability is the branch of probability that deals with how the likelihood of future events is related to the occurrence of past events. If you flip a coin, what are the odds of it coming up heads 10 times in a row? The odds are 1 in 2 raised to the 10th power, or 1 in 1,024. But what if the coin has already been flipped nine times and come up heads each time? The only remaining random event is the last coin toss. The coin will either come up heads, making 10 heads in a row, or tails, making nine heads in a row followed by one tail. Thus, the conditional probability of a coin coming up heads 10 times in a row, if it has

already come up heads nine times in a row, is 1 in 2, or 50 percent.

The claim that software engineering is all common sense is related to this idea. Once someone has explained the concepts clearly, they all seem like common sense because the newcomers don't have to explore all the answers that sounded plausible but that turned out to be wrong. They didn't see the many series of coin tosses in which we got tails on the first, second, or third toss. They're only seeing the series from the vantage point of already having nine heads with one to go; they haven't banged their heads against the wrong answers enough times to know just how many wrong answers there are.

We can interpret the fact that people think any part of software engineering is commonsensical as a sign that the field is maturing. A few effective practices are now being described so clearly that they appear to be obvious. When we get to the point that all of software engineering appears to be trivial, we will know the profession has really come of age.

References

1. A. Cole, "Runaway Projects—Cause and Effects," *Software World*, vol. 26, no. 3, 1995, pp. 3–5.
2. B.W. Boehm, "Software Risk Management: Principles and Practices," *IEEE Software*, vol. 8, no. 1, Jan. 1991, pp. 32–41.
3. C. Jones, *Assessment and Control of Software Risks*, Yourdon Press, Englewood Cliffs, N.J., 1994.
4. *SPMN Software Evaluation Model, Vol. II: Project Execution*, Software Project Manager's Network, 2000; www.spmn.com/products_guidebooks.html (current 13 June 2001).
5. T. Gilb and D. Graham, *Software Inspection*, Addison-Wesley, Wokingham, UK, 1993.
6. F. O'Connell, *How to Run Successful Projects II: The Silver Bullet*, Prentice Hall Int'l, London, 1996.
7. S. McConnell, *Rapid Development*, Microsoft Press, Redmond, Wash., 1996.

Upcoming Topics

September/October '01:
Benchmarking Software Organizations

November/December '01:
Avoiding Defects Faster

January/February '02:
Building Systems Securely from the Ground Up

March/April '02:
Building Internet Software

May/June '02:
Knowledge Management in Software Engineering