In their article “Misleading Metrics and Unsound Analysis” (March/April 2007), Barbara Kitchenham, David Ross Jeffery, and Colin Connaughton accuse the Software Measurement Process Standard ISO/IEC 15939—which, by the way, is misquoted in the opening sentences of the article as ISO/IEC 15393—of misleading practitioners by providing advice on measuring current productivity and estimating the productivity of future projects. This is totally inappropriate, because the standard declares in its scope, “This International Standard does not catalogue software measures, nor does it provide a recommended set of measures to apply on software projects.” Furthermore, in the appendix to which the article refers, the first paragraph states:

The following sub clauses provide examples of instantiations of the measurement information model that address specific information needs. These examples are not designed to recommend best measurement practices, but rather to show the applicability of the measurement information model in a variety of common situations.

I would appreciate it if you could clarify this in the next issue.

Eduardo Miranda
Former Canadian representative to the ISO/IEC SC7/WG13

The article “Misleading Metrics and Unsound Analysis” is itself guilty of misleading readers and offering unsound advice. The authors attack the measurement construct provided in the Annex of ISO/IEC 15939 for productivity. The text of the standard is very clear that the Annex contains examples intended to illustrate the information model, not recommended measures or indicators. I’ve had discussions with many people in industry who have read the ISO/IEC 15939 standard, and none of them ever came away with the idea that the examples in the standard were recommended measures. Anyone working in industry knows that the problem is more complex than the example. The article actually discusses very few of the many issues that must be addressed in developing a productivity measure. Let me refer to my own article on this subject (“The Challenge of Productivity Measurement,” Proc. Pacific Northwest Software Quality Conf., 2006), for a more complete discussion of productivity.

Once the authors have thrown their bricks (a glancing one for the CMMI, as well), most of their discussion has little to do with the content of the standard (or the CMMI). As an example, neither ISO/IEC 15939 nor the CMMI says anything specific about the size of data sets, yet the authors introduce the “misconception” that large data sets are needed for effective analysis. Where do the misconceptions in this article come from? The authors seem to be implying that they come from the ISO/IEC standard and the CMMI.

The last “lesson” is the only one with a specific link. It suggests that ISO/IEC 15939 recommends the use of ratios of measures. Again, the ISO/IEC standard provides exam-
ples of how to define such measures, not recommendations on which ones to use or whether to use them at all. Because many common measures of practical interest (such as productivity and defect rate) are ratios of base measures, the standard’s guidance seems more helpful than the authors’, which appears to be only to avoid such ratios.

Kitchenham and her coauthors seem to have entirely missed the purpose of ISO/IEC 15939—to define basic terms and concepts in measurement. Nevertheless, their article helps to illustrate why such a standard is needed. The captions for figures 2, 3, and 5 describe these graphs as run charts with control limits, but these graphs look a lot more like confidence intervals around a running average than like control limits. Interestingly, if the authors had followed the actual recommendation in ISO/IEC 15939 for defining measures, the reader would know what these lines are intended to represent. Instead, the reader knowledgeable in control charting is left wondering what this discussion is about—it isn’t about control charts in the usual sense! The productivity example in the standard makes a good (if misguided) target exactly because it is well defined, even if it’s only an example.

David N. Card
Former Coeditor of ISO/IEC standard 15939

Barbara Kitchenham, David Ross Jeffery, and Colin Connaughton respond:
We accept that our article criticizes a measurement construct provided in the Annex of ISO/IEC 15939 for productivity; indeed, we note in our article that the construct is in the appendix. However, we don’t think misleading examples being in an appendix means that the standard shouldn’t be criticized.

We also accept that we haven’t covered all aspects of productivity measurement and prediction in our article. For example, a simple ratio measure is unhelpful if the outcome of an activity is multidimensional. Also, the ratio measure of productivity isn’t constant across a set of projects if the projects exhibit economies or diseconomies of scale. Our goal was to identify some of the problems and to use graphical representations that demonstrate clearly the nature of the problems.

Some comments on comments
Although I agree with most of what Hakan Erdogmus said in his March/April 2007 From the Editor column (“What’s Good Software, Anyway?”), I think he missed important functions of comments with respect to data.

First, there’s the obvious function of adding units of measure. Sometimes the unit of measure can be built into the name, but “Latency-InMicroseconds” gets tiresome. Second, there’s the range of valid values, when the range is constrained. This gets worse if the range has discontinuities. For Booleans, it may be necessary to spell out exactly what truth and falsehood represent. The name may not convey meaning without ambiguity. In my experience, it’s next to impossible to answer some questions about data in the absence of annotation.

I don’t think this violates ideas about the use of comments. Use them when needed for communication. The problem is that good communication skills aren’t widely distributed among code writers.

Bill Talbot
Principle member of engineering staff,
Lockheed Martin
william.b.talbot@lmco.com

Yes, software engineering is fun
Thanks once again for a most stimulating Loyal Opposition column (“Is Software Engineering Fun?” Jan./Feb. 2007). I played along with Bruce Blum’s assessments and found my answers quite different.

Selling the concept. I don’t consider myself a people person. I dread the thought of having to convince some ill-defined collection of people that my idea is laudable. If I liked selling, I’d be in used cars. Fun to tedium is split 20/80. (On a side note, I thoroughly enjoy teaching, from three-year-olds in Sunday School to graduate students in college. I’m a real sucker for anything beginning with, “Could you explain …?”)

Requirements. What does the customer really need? What makes this work? How do I express this as needs and not implementations? What shiny tidbit is still hiding in a dusty corner somewhere? I’d rate this as 75 percent fun and 25 percent tedium. Of course, I’ve never written requirements for more than a few hundred lines of code.

Top-level design. Oh, boy, we’re getting closer to the programming. The challenge is to figure out ingenious ways to satisfy the requirements those Twilight Zone folk came up with. I have faith there’s always some “junk-ridden garage” that lowers the fun-edium ratio to 60/40.
Detail design. That’s why they pay me the big bucks, isn’t it? Still, I can take pride in a job well done. By the end of the umpteenth page, I’m at 50/50.

Programming. Now my output is executable. The ratio jumps back up to 75/25. Here, at last, I agree with Bruce.

Testing. What? You don’t have that devious twist that says, “I’ll bet the developer forgot this,” or, “there’s a bug in there somewhere, and I’m just the one to expose it”? The growing coverage numbers and the satisfaction of knowing that not much is going to get by me pushes the level as high as 80/20.

Maintenance. Yeah, it takes a bit of charity to adopt an orphan piece of software, figure out what makes it tick, and breathe new life into it. But it brings those opportunities to rip out a whole old, crusty subsystem and reengineer it shiny and new and better than ever. Fun to tedium ratio is no lower than 50/50 here.

So, Bruce, maybe you’re in the wrong line of work. Maybe I have more joie de vivre. Or maybe I’ve just had cushy jobs, while you’ve taken on the really hard tasks that make our world go around. Dunno.

Paul E. Black
Computer scientist, National Institute of Standards and Technology
p.black@acm.org; paul.black@nist.gov

Dealing with aging systems

“What can developers do when faced with an aged software system?” Diomidis Spinellis’ column “Silver Bullets and Other Mysteries” (May/June 2007) presented this interesting question. Should the answer be, “They could simply come clean”? Some people, particularly in a corporate environment, might feel that it’s quicker to safely fix the current system to reap the rewards of increased productivity. No doubt, a brand new system or application comes with a price—users and system administrators must spend time learning and then maintaining a system. And the more complex the system, the more costly it is to deploy. However, I believe that the gain would outweigh the loss once the users have become familiar with the new system. Many aged systems aren’t scalable, and further maintenance is back to square one in terms of fulfilling ever-increasing sophisticated computing requirements.

This question can be extended to every computer user. Many of us have experience with operating system decay. We install software, upgrade it, and then uninstall it; we add and remove peripherals and device drivers; we run security updates and service patches. Over time, the operating system will be running on a different machine than it was when it was first installed. This can take an agonizing toll on the user in the form of frequent system lockups or crashes, unusually slow performance, and strange error messages. We attempt to remove unnecessary files and then run Scandisk and Disk Defragmenter. Nevertheless, rebuilding the system from scratch is the only solution if the system continues down this problematic path.

In other cases, a significant security incident, for example, might simply mean that an intruder has breached and manipulated our machine. Certainly, we can patch the system and clean up the changes. According to my experience, you can’t just fix the system and forget about it—it’s possible to spend hours rebuilding a system, only to have it cracked shortly after putting it back up. The troubleshooting and removal processes are challenging yet necessary, and sometimes it’s difficult to discover all the changes made to a cracked system (such as backdoors and hide files). You should repair a compromised system only when it’s possible to guarantee a clean system. Otherwise, it would be wise to rebuild from scratch, either by installing clean new versions of the software or recovering from a clean image and then performing the necessary upgrade.

To improve is to change, and the only way to stop computer problems is to keep up with technology. It’s also important that clients understand systems’ requirements. You must involve them in the project and give them a clear picture of how the new system will work for them—and, most importantly, how it will work better.

Hong-Lok Li
Information technology manager
University of British Columbia
lihl@ams.ubc.ca

The sad fact is that as long as upper management people don’t understand what software development is about, they will bully and intimidate their subordinates into meeting their executive goals. The lower-level managers, in turn, will fail to report the truth to the higher levels for fear of retaliation or, at very least, censure. The overall result of this diabolical lack of honest communication is usually that the manure flows downhill and ends up
on the heads of the lowest of the low (the developers), who have to do more and more with less and less, faster and faster, until they are doing everything with nothing, instantly.

Edwin Fine
emofine@finecomputerconsultants.com

In my experience, rebuilding a system from scratch is usually a bad idea, and we have numerous examples of why this is. Just getting to the point where you can offer most of the old system’s functionality usually takes a lot more time than initially anticipated. It’s much better to come up with an architecture that lets you replace smaller or larger components at times and reuse parts of whole subsystems. However, this is not always possible, and in some cases you just need to switch to a better underlying infrastructure. But even then reusability is possible. IMHO there are no silver bullets other than sound programming principles, experience, and talent.

Dimitris Andreadis
Project lead, JBoss AS
dandread@redhat.com

Diomidis Spinellis responds:
I agree that rebuilding a system from scratch can be problematic, and that an architecture that allows gradual evolution is the way to go. However, the architecture that serves us in this way today is unlikely to serve us similarly in 20 years or after 20 years of accumulated cruft. After all, when we design an evolvable architecture, we allow for evolution in some specific foreseen axes. Unfortunately, the changes we are asked to make in our system often violate the initial architectural assumptions on how our system would evolve.

As an example, consider Microsoft’s MS-DOS. Its architecture allowed the addition of new commands and new INT 21-based system calls (there was even a call to check whether a given system call was supported). However, MS-DOS was clearly unsuitable as a base for running Windows, with its C-based API and support of the i386 memory management features. Consequently, Microsoft ditched it in favor of the NT platform. It was painful for Microsoft and its customers, but it had to happen.