ride my bicycle to work most days. On the way
in, immersed in the beauty of the bike trails, I
plan my day, thinking about the most pressing
issues and how to address them. On the way
home, I assess my performance, plan to im-
prove what I've done so far, and address those
things yet undone. I don't think of my 30-mile com-
mute as an “extra,” to be bolted on to my day only
if I have time. Instead, it's integral
to who I am, what I do, and how
I do it.

I have the same attitude about
software measurement. When we
were planning our book, Solid
Software (Prentice Hall, 2002),
Chuck Howell, Les Hatton, and
I discussed what activities de-
velopers should include in the
development process to ensure
that software is reliable and useful. Chuck and
Les were surprised that I didn't put software met-
rics on the list. But I think metrics should be inte-
gral to many of the other development activities;
I wanted our book to discuss it in the context of
other activities, not as a separate— and therefore
separable—activity.

Have we reached that goal? I don't think so. But
we have come a significant way in 25 years.

From the Beginning
When IEEE Software was born, researchers were
trying to develop approaches to software measure-
ment and software cost estimation using “software science”1
and function points2 to capture the func-
tionality and size of what we build. This work was
consistent with a general faith in engineering, as
typified by books like The Limits to Growth3; we
thought that we could describe our world with a se-
ries of equations, plug in the right values for the pa-
rameters, and control the future.

This work was followed by a rush to process in
the 1980s. Manufacturing experts such as W. Ed-
wards Deming convinced us that, armed with pro-
cess definition and good measurement (in this case,
statistical process control), we could build qual-
ity into our products.4 Inspired by that approach,
software developers defined processes and applied
“maturity models” to improve our output: quality
software. When we associated measures with the
process models, we began to see that measurement
begins with baby steps; good metrics grow more
sophisticated as we understand more about what
we're trying to measure. Whether we gain that
understanding from asking about goals and ques-
tions5 or associating attributes with process inputs,
outputs, and flows,6 the metrics provide visibility
and insight about what we're doing and how well
we're doing it.

In the 1990s, capability maturity was all the
rage. Level 4 explicitly involved measurement, and
organizations rushed to implement metrics pro-
grams in the race to be rated a “4” or “5” before
their competitors. Dan Paulish and Anita Carleton
discussed measurement success in the context of
process improvement,7 and eventually the CMMI
included a separate Measurement and Analysis
process area. IEEE Software published a special
issue on software measurement, including a sta-
tus report saying that we were headed in the right
direction, but it was still a long journey to our
destination.8

Barbara Kitchenham and her colleagues be-
gan cautioning us to be rigorous and thoughtful
in our measurement and to make sure that we
were really measuring what we thought we were.

We began to look at the human context in which we were measuring—for example, Tracy Hall and Norman Fenton examined what makes a metrics program effective and welcome in an organization. Slowly, the research community (but only a few practitioners) began using metrics not as ends but as means. For instance, we can use measurement as a tool to investigate other issues, such as whether formal methods improve software quality and whether mature organizations produce better software. By the end of the 1990s, software practitioners saw measurement as useful and necessary. But, like the acknowledgment that healthy diet and exercise are important, there was more talk than action.

The Path Forward

It’s a new millennium. More than ever, metrics are like my bicycle commute. We hear less about “the right metrics” and more about the right kind of evaluation. This change’s genesis has its basis in the safety and dependability community. Forward-thinking researchers such as Bev Littlewood have been counseling the UK Nuclear Safety Advisory Committee for many years about the need for safety cases. That is, it’s not enough to make claims about your software; you must support your claims with measurable evidence. At the same time, companies such as Adelard have built tools to support safety-case creation and management. In 2004, Chuck Howell and his colleagues convened a workshop on using safety cases for all kinds of dependable systems, based on the notion that metrics provide evidence to support “dependability cases” about software quality and functionality. Inspired by David Schum’s seminal work, I explained how to apply concepts about legal evidence to software measurement, so that a variety of metrics can support arguments about how a given software product will likely work in a particular environment. And in a report for the US National Academy of Sciences, Daniel Jackson and Martyn Thomas recently suggested requiring dependability arguments for safety-critical software systems, much as they’re already required for proposed nuclear power plants.

Today, we no longer value metrics in and of themselves—we value and use them as a means to an end. That is, they’re becoming part of the software development fabric, an integral thread that’s essential to understanding whether what we’re building will be all that we hope it will be. We still have significant ground to cover, however, before metrics are as commonplace in software engineering as they are in other engineering disciplines. Researchers now acknowledge the need for them, but many practitioners are still skeptical, especially for small projects. We must overcome several key obstacles.

Dealing with Uncertainty

First, we must learn how to deal with considerable uncertainty in software development. As engineers, we often handle uncertainty by making assumptions about underlying probability distributions. These assumptions typically involve choosing a familiar distribution, such as Poisson or beta, that simplifies modeling or computation. But as researchers, we rarely question our assumptions’ appropriateness or study data distribution in its own right to propose new and more apt ways to describe the actual uncertainty we see. For example, researchers who study software quality improvement often assume that quality is an increasing (and usually convex) function over time. But data on fault and failure rates usually disprove this assumption; indeed, fixing one problem often introduces or enables another, so quality can actually degrade with a fix. As practitioners, when we use models developed in the research community, we rarely see the assumptions expressed explicitly, let alone evaluate whether they apply to our particular situation. This disconnect is especially true when we apply economic models for cost estimation or trade-off analysis.

Measuring “Soft” Characteristics

Second, we must recognize software development’s multidisciplinary nature and the concomitant need for measuring “softer” aspects of relevant variables, such as developer experience or expertise. We lived with ballpark estimates of factor-of-ten variability in developer skill for many years as we studied other issues. Now, it’s time to involve social scientists in our research and to develop useful measures that can help us understand the degree to which context, experience, and expertise affect our product and process outcomes. Some investigations are already showing promise: Nachiappan Nagappan, Brendan Murphy, and Victor Basili suggest that organizational structure is a good predictor of software quality, and my colleagues Martin Libicki and Michael Webber and I found that “market discipline” might predict attitudes toward the amount and kind of cybersecurity built into today’s software products. In the end, we should be able to make better decisions about software quality—as well as hiring, training, and team building—by measuring the past and changing the future.

Anticipating Change

Third, we must find a way to understand and anticipate some of the inevitable change we see during software development and maintenance. We expect changes in requirements and design, but we forget to think about changes to the environment in which people will use the software, or changes to the team doing development or maintenance. These changes can be critical to a system's proper performance. Manny Lehman laid out broad areas of change in his software evolution analysis, where he focused on whether the change derives from the problem, the solution, the degree of abstraction, or the degree to which the solution affects the problem itself (as when computerized stock trading changes the very market it’s trying to model). As practitioners and researchers, we should...
think more carefully about the genesis of change and try to design our systems to embrace it gracefully and appropriately.

Developing Heuristics

Fourth, we should make more room for heuristics, based on careful studies of their effectiveness. Gerd Gigerenzer and Richard Selten point out that heuristics can often be more effective than strict, rational rules because humans aren’t always capable of optimizing their behavior. Heuristics can help us understand when some metrics and processes are good enough, even when they can never be perfect. For example, we don’t always need to measure everything with high levels of precision; it’s often enough to know a rating on a five-point scale, where 1 “meets none of the requirements” and 5 “meets all the requirements.” Even when using such a highly subjective method, we know that the system isn’t ready to be coded if the design review yields mostly 1s and 2s.

Today, both practitioners and researchers encounter institutional barriers to addressing these needs—including the narrow focus of most researchers’ agendas, oppressive market pressures on software development teams, and the lack of reward for cross-disciplinary work. Nevertheless, a time will come when software measurement will be an integral part of software development and maintenance.

References