Mythoids don’t represent absolute falsities. They occupy that uncomfortable grey area in our perception of the world. Our beliefs about software development transform into myths as deeper understanding reveals essential underlying difficulties. Categorical dismissal follows frustration and displaces efforts to tackle those difficulties. The problem with dispelling mythoids is the twice-sunk intellectual investment. Our brains don’t like to be rewired back and forth, each time wasting our appreciation for the clever arguments and counterarguments made.

Which brings us to this editorial’s subject. I’d like to address this mythoid: “You can measure software development productivity.” Commonly held, it belongs to a larger family I call measurement acquiescence. It’s a mythoid because you can measure productivity to a useful goal. And many indeed do. For example, extreme programmers can track “velocity” to improve their estimation capability, and what’s velocity but a productivity measure? In particular, effort estimation hinges on the ability to gauge productivity.

Yes you can, but …

Measuring software development productivity is at best difficult. Like all software measurement, it requires careful thought and comes with ample caveats. But we can still do it, and do it meaningfully, provided we understand why we’re doing it, and provided we’re aware of limitations. We may not be able to compare the productivity of a large team developing a commercial API for masses to that of a small in-house team developing a custom web portal. Productivity measurement still can help to answer a whole host of relevant questions.

I’ve been able to measure my own productivity to get a better idea of why I’m going faster sometimes and slower other times, and to test my intuition about the effectiveness of a new technique I’m trying out. We can reliably gauge individuals’ and teams’ productivity in controlled experimental settings. And we can devise and collect sound, context-aware measures to compare and benchmark projects of similar characteristics. If you can’t gauge and leverage productivity meaningfully across organizational boundaries, you might still be able to do it within a single organization or a single project.

Purposefulness and context

Productivity in economics is the ratio of a production process’s output to its input. This standard definition, which is independent of any underlying scale economies or diseconomies, is the one I’ve adopted.

The difficulty with measuring productivity is that of measuring development output. Software development doesn’t have a universal, perfect output measure, but some proxies do make sense in specific contexts and for specific purposes. If you know why a certain measure is unfit for a goal or in a given context, you can often adjust it in view of that knowledge or find a better-fitting proxy.

Even the simplest output measure, LOC, can be meaningful in certain circumstances. And the most complicated measures, adjusted for $n$ factors, are meaningless in some circumstances in which simpler and more intuitive alternatives will do.

The central trade-off is between the measure’s granularity and its correlation with external functionality. On one hand, fine-grained measures such as LOC tend to be more uniform and suffer less from...
instability due to low values. The more unequally sized or scoped an output measure’s unit is, the lower the measure’s uniformity. Low uniformity causes counting inconsistencies that result in instability.

On the other hand, fine-grained measures’ correlation to external functionality tends to be weaker. Conversely, coarse-grained measures—such as those based on function points, user stories, story points, use cases, scenarios, and features—tend to be less uniform and more prone to low-value instability. But they correlate better with external functionality. Instability arises when you can’t be sure about the significance of the ratio between two low-valued measurements. For example, it might not be possible to confidently state that two user stories, scenarios, and features—tend to be less uniform and more prone to low-value instability. But they correlate better with external functionality. Instability arises when you can’t be sure about the significance of the ratio between two low-valued measurements.

Once you determine the level of granularity, adjusting or normalizing a measure can make it more uniform, stable, and meaningful for the purpose at hand. Suppose two teams within an organization are developing accounting applications for different functions. Both applications are written in Java using the same platform, framework, and libraries. The requirements aren’t expressed at a level sufficiently fine-grained to track weekly progress (the team should think about fixing that later). The organization has a standard way of counting new source lines of code and settles on LOC as the base output measure. One concern might be that this measure is easily distorted by code cloning, a discouraged practice that leads to poor design and difficult-to-change code. If a team is cloning code extensively, its productivity will be unfairly inflated. It would then make sense to use a clone analysis tool to gauge the extent of duplication in the code and adjust the output measurements accordingly, thereby penalizing copy-and-paste programming and rewarding reuse. This adjustment turns LOC into a normalized measure—clone-free LOC.

Or perhaps the organization doesn’t value untested code. A coverage tool could allow the counting of only the code covered by tests. The resulting output measure could then be branch-covered LOC. Or the organization might want to encourage the use of a standard framework’s built-in libraries, because it considers reinventing the wheel a waste of resources. In that case, a custom counting tool can adjust the LOC measure upward by in-lining library calls.

Multiple measures

Using multiple measures lets you triangulate your insights. You might combine a fine-grained measure with a coarse-grained measure, such as LOC with number of use cases. You can adjust use cases according to their complexity by using a weighting scheme to improve uniformity. As you

Software Boards

Three associate editors in chief recently joined our editorial board. John Favaro, our new management editor, runs an IT consulting practice in Italy. He is a leading member of the software economics and reuse communities and a pioneer of value-oriented software management. Ann Hickey is associate professor of information systems and associate dean in the University of Colorado’s College of Business at Colorado Springs. Ann is extremely active in the requirements engineering arena. Complementing her knowledge in requirements elicitation, Ann brings extensive industrial experience as our new Requirements editor. John Grundy is a software engineering professor, deputy department head, and co-director of the University of Auckland’s Centre for Software Innovation. He holds a joint appointment in the computer science and computer engineering departments. We’ll leverage his broad, practical expertise to grow the magazine’s newly established Distributed and Enterprise Software coverage area.

Former associate editors in chief Roel Weirenga and Stan Rifkin deserve many thanks for their outstanding service. Together they oversaw three core areas, provided invaluable guidance to authors, and spearheaded many special projects. Other retirees include long-serving advisory board members Rob Thomsett and John Vu. I thank all the departing members and wish them success in their future endeavors.
aggregate your measurements over several individuals, teams, and projects, averaging-out effects might kick in and they start behaving increasingly similarly and in a more stable manner. For example, the contorting effects of skill will matter less when you’re measuring an entire organization’s productivity than it will when measuring a small project’s productivity.

In research settings, deliberate controls and randomization will also lessen ill effects. In a study designed to evaluate two development strategies, you can select subjects to achieve a balanced mix of skills. Randomization minimizes the influence of factors that apply equally to the strategies under evaluation. Such precautions won’t completely eliminate biases or guarantee portability, but they’ll let you assess productivity appropriately to satisfy the study’s goals.

Thinking outside the box

You need not limit yourself to obvious output measures. If acceptance tests are standard, the number of passing acceptance tests (perhaps adjusted using a weighting scheme) could make a good output proxy. Acceptance tests embody requirements, and passing tests represents chunks of functionality realized. This might ultimately be what you care about.

If the output measure is functionality— or requirements-oriented, say based on user stories delivered, but you want it to also account for end value delivered, you could have customers or users assess the released stories and rate them. You can then use these ratings as weights to adjust the measure. A user story that turns out to have no value to the customer will then be considered waste. Be careful about interpreting such measures: they’ll capture productivity from the perspective of software’s consumer: external factors’ contributions such as requirements volatility might end up playing a more significant role than the development team’s internal effectiveness.

Real productivity

It’s important not to take end-value reasoning to the extreme point of confusing productivity with profit, return on investment, or other operational benefits. A broom factory might produce on average 100 brooms a day but might not be able to sell all of them. The factory’s profitability at a given time will depend on several other factors outside the production process. Examples are selling price, inventory costs, demand for brooms, and the marketing department’s effectiveness. Thus, while profitability may fluctuate, the speed of production remains constant at 100 brooms a day.

What if you want to account for quality, or lack thereof, in your productivity measure? This question relates to short-term or nominal productivity versus long-term or real productivity. The latter is when we factor into the measure the effects of latent implications, such as extra future rework. A process that’s seemingly highly productive in the short term might result in considerable downstream costs in the long term. The net effect is a lower real productivity. This is often the case with poor quality. If you track or estimate the cost of poor quality, you can adjust nominal productivity downward over time to reflect real productivity.

It’s also important to distinguish between different types of productivity and how they interact. For example, we measure rework output (bugs fixed) in different terms than nominal production output (functionality delivered). We can then combine these measures to estimate real productivity (bug-free functionality delivered).

Y es, measuring software development productivity is difficult, but it’s not futile. It’s prone to misuse and misinterpretation, and highly portable, precise measures are elusive. But context-dependent and proximate measures can still be very valuable. Let’s not succumb to measurement acquiescence that easily.

FROM THE EDITOR

Thanks to Our 2007 Reviewers

Authors often tell us how much they value our reviewing process. We sincerely thank all of you who reviewed submissions for IEEE Software during the past year. Your expertise, care, and attention add immeasurably to the magazine’s quality and standing among professionals. For a full list of the 2007 reviewers, see www.computer.org/software/07reviewers.html.