The Role of Object-Oriented Metrics

Bertrand Meyer, EiffelSoft

Perhaps the most common concern of project managers who use or who are about to use object technology is for more measurement tools. I suspect that some of these people would kill for anything that could give them some kind of quantitative grasp on the software development process.

There is, in fact, an extensive literature on software metrics, including much that pertains to object-oriented development. But surprisingly little of it is of direct use to actual projects. The publications that directly address object-oriented metrics often go back quite a while. An example is Barry Boehm's *Software Engineering Economics* (Prentice Hall, 1981), with its Cocomo cost-prediction model. Despite the existence of more recent works on the subject, Boehm's is still among the most practical sources of quantitative information and methodology.

Metrics are not everything, of course. Lord Kelvin's famous observation is exaggerated:

> When you cannot measure, when you cannot express [what you are speaking about] in numbers, your knowledge is of a meager and unsatisfactory kind: You have scarcely, in your thoughts, advanced to the stage of a science.

Editor: Bertrand Meyer, EiffelSoft, ISE Bldg., 2nd Fl., 270 Storke Rd., Goleta, CA 93117; voice (805) 685-6869; ot-column@eiffel.com

Much of math—and most of logic—is not quantitative, but we don't dismiss those elements of science as nonscientific.

Much of math—and most of logic—is not quantitative, but we don't dismiss those elements of science as nonscientific.

These considerations also put in perspective some of the comments published recently in *Computer* by Walter Tichy ("Should Computer Scientists Experiment More?" May 1998, pp. 32-40) on the need for more experimentation; his article was largely a plea for more quantitative data. I agree with Tichy's central argument: We need to submit our hypotheses to the test of experience. But when Tichy writes

> Zelkowitz and Wallace also surveyed journals in physics, psychology, and anthropology and again found much smaller percentages of unvalidated papers [that is, papers not supported by quantitative evaluation] than in computer science…

I cannot help but think, "Physics, OK, but do we really want to take *psychology* as the paragon of how scientific computer science should be?" I don't think so. In an engineering discipline, we cannot tolerate the fuzziness that is almost inevitable in social sciences, in spite of all their numbers. If we are looking for rigor, the tools of mathematical logic and formal reasoning are crucial, even though they are not quantitative.

Still, we need better quantitative tools. In this column I present a classification of software metrics and five basic rules for their application.

TYPES OF METRICS

The first rule of quantitative software evaluation is that if we collect or compute numbers we must have a specific intent related to understanding, controlling, or improving software and its production. This implies that there are two broad kinds of metrics: *product metrics*, which measure properties of the software products, and *process metrics*, which measure properties of the process used to obtain these products.

Product metrics include two categories: External product metrics cover properties visible to the users of a product; internal product metrics cover properties visible only to the development team.

External product metrics include

- *Product nonreliability metrics*, which assess the number of remaining defects.
- *Functionality metrics*, which assess how much useful functionality the product provides.
- *Performance metrics*, which assess a product's use of available resources, including computation speed, space, and occupancy.
- *Usability metrics*, which assess a product's ease of learning and ease of use.
- *Cost metrics*, which assess the cost of purchasing and using a product.

Internal product metrics include

- *Size metrics*, which provide measures of how big a product is internally.
- *Complexity metrics*, which are closely related to size metrics and assess how complex a product is.
- *Style metrics*, which assess adherence to writing guidelines for product components like programs and documents.

Process metrics include

- *Cost metrics*, which measure the cost of a project or of some project activities such as original development, maintenance, and documentation.
- *Effort metrics*, which are a subcategory of cost metrics and estimate the human part of the cost, typically measured in person-days or person-months.
- *Advancement metrics*, which estimate the degree of completion of a product under construction.
- *Process nonreliability metrics*, which assess the number of defects uncovered so far.
- *Reuse metrics*, which assess how much of a development benefits from earlier developments.

INTERNAL AND EXTERNAL METRICS

The second rule is that internal and product metrics should be designed to mirror relevant external metrics as closely as possible. Clearly, the only metrics of interest in the long run are external metrics, which assess the result of our work as perceived by our market. Internal metrics and product metrics help us improve this product and the process of producing it. These metrics should always be designed so as to be eventually relevant to external metrics.

Object technology is particularly useful here because of its seamlessness properties, which reduce the gap between problem structure and program structure—the *direct mapping property*. In an object-oriented context the notion of function point—a widely accepted measure of functionality—can be replaced by a much more objective measure: the number of exported features (or operations) of relevant classes that require no human intervention and can be measured trivially by a simple parsing tool.

DESIGNING METRICS

The third rule is that any metric applied to a product or project should be justified by a clear theory of what property the metric is intended to help estimate. The set of things we can measure is infinite, but most of them are not interesting. For example, I can write a tool to compute the sum of all ASCII character codes in a program, but this is unlikely to yield anything of interest to product developers, product users, or project managers.

> The set of things we can measure is infinite, but most of them are not interesting.

A simple example is a set of measurements that we performed some time ago on the public-domain EiffelBase library of fundamental data structures and algorithms, reported in my book *Reusable Software* (Prentice Hall, 1994). One of the things we counted was the number of arguments to a feature (attribute or routine) over 150 classes and 1,850 features; we found an average of 0.4 and a maximum of three, with 97 percent of the features having two or less.

We were not measuring this particular property blindly: It was connected to a very precise hypothesis that the simplicity of such interfaces is a key component of the ease of use (and hence the potential success) of a reusable component library. These figures show a huge decrease in arguments compared to the average number for typical non-OO-subroutine libraries, which often contain five or more and sometimes as much as 10. (Note that a C or Fortran subroutine has one more argument than the corresponding OO feature.)

Sometimes people are skeptical of the reuse claims of object technology; after all, their argument goes, the idea of reuse has been around for a long time, so what's so special with objects? But quantitative arguments like those derived from EiffelBase measurements provide some concrete evidence to back the OO claims.

A THEORY OF METRICS

The third rule requires a theory and implies that the measurements will only be as good as the theory. Indeed, the correlation between a small number of feature arguments and how easy a library is to use is only a hypothesis. Authors such as Tichy might argue that the hypothesis must be validated through experimental correlation that measures ease of use.

They would have a point, but the first step is to have a theory and make it explicit. Experimental validation is seldom easy anyway, given the setup of many experiments that often use students under the sometimes dubious assumption that their reactions can be used to predict the behavior of professional programmers.

In addition, as Tichy suggests, it is very hard to control all the variables. For example, I recently found out, by going back to the source, that a 1970s study often used to support the use of semicolons as terminators rather than separators in programming languages seemed to rely on an unrealistic assumption that casts doubt on the results.

Two PhD dissertations at Monash University—by Jon Avotins and Glenn Maughan under the supervision of Christine Mingins—have applied these ideas further by producing a guide called a "Quality Suite for Reusable Software" (http://www.sd.monash.edu.au). Starting from several hundred informal methodological rules in the book *Reusable Software* and others, they identified the elements of the rules that could be subject to quantitative evaluation, defined the corresponding metrics, and produced tools that evaluate these metrics on submitted software. Project managers or developers using these tools can assess the values of these measurements on their products.

In particular, you can compare the resulting values to industry-wide standards or to averages measured over your previous projects. This brings us to the fourth rule, which states that measurements are usually most useful in relative terms.

CALIBRATING METRICS

The fourth rule is that most measurements are only meaningful after calibration and comparison to earlier results. This is particularly true of cost and reliability metrics. A sophisticated cost model such as Cocomo will become more and more useful as you apply it to successive projects and use the results to calibrate the model's parameters to your own context. As you move on to new projects, you can use the model with more and more confidence because of comparisons with other projects.

Similarly, many internal product metrics are particularly useful when taken relatively. Presented with an average argument count measure of four for your newest library, you will not necessarily know what it means. Is this good, bad, or irrelevant? Assessed against published measures of goodness, or against measures for previous projects in your team, the figures will become more meaningful.

Particularly significant are outlying points: If the average value for a certain property is five with a standard deviation of two, and you measure ten for a new development, it's probably worth checking further, assuming of course—see rule two—that there is some theory to support the assumption that the measure is relevant. This is where tools such as the Monash suite can be particularly useful.

METRICS AND THE MEASURING PROCESS

The fifth rule is that the benefits of a metrics program lie in the measuring process as well as in its results. The software metrics literature often describes complex models purporting to help predict various properties of software products and processes by measuring other properties. It also contains lots of controversy about the value of the models and their predictions.

But even if we remain theoretically skeptical about some of the models, we shouldn't throw away the corresponding measurements. The very process of collecting these measurements leads—as long as we confine ourselves to measurements that are meaningful, at least by some informal criteria—to a better organization of the software process and a better understanding of what we are doing.

This idea explains the attraction and usefulness of process guidelines such as the Software Engineering Institute's Capability Maturity Model that encourage organizations to monitor their processes and make them repeatable, in part through measurement.

To quote Emmanuel Girard, a software metrics expert, in his advice for software managers: "Before you take any measures, take measurements." ❖