tivity and flexibility in output, which could result in information customization systems.

Data interchange. Data interchange involves transforming information from one representation to another. Often, the same initial and target formats occur many times, as in converting documents from one word processing format to another. Sometimes, though, data must be converted to a specialized format for a one-time analysis. Typically, a program will be written to do this custom conversion. To our knowledge, no one has yet developed a tool for interactively helping a user specify initial and/or target formats. Such an artifact would be a useful information customizing tool.

Better tools are coming. Access to the "common world brain" envisioned by Wells will remain insufficient until information can be presented in a form customized to each consumer's evolving needs. Part 2, in the next issue of *Computer*, will provide some examples of information customization and describe our prototypes of information customizing systems.

References

1. H. Berghel, "Cyberspace Navigation," *PC AI*, Vol. 8, No. 5, Sept./Oct. 1994, pp. 38-41.
2. *Comm. ACM*, special section on information filtering, Vol. 35, No. 12, Dec. 1992, pp. 26-81.
3. N.J. Belkin and W.B. Croft, "Information Filtering and Information Retrieval: Two Sides of the Same Coin?" in Ref. 2, pp. 26-38.
4. B. Sundheim, ed., *Proc. Third Message Understanding Conf.*, Morgan Kaufman, San Mateo, Calif., 1991.
5. N. Chinchor, L. Hirschman, and D.D. Lewis, "Evaluating Message Understanding Systems: An Analysis of the Third Message Understanding Conf. (MUC-3)," *Computational Linguistics*, Vol. 19, Sept. 1993, pp. 409-449.
6. G. Piatetsky-Shapiro and W.J. Frawley, eds., *Knowledge Discovery in Databases*, MIT Press, Cambridge, Mass., 1991.
7. C.-S. Ai, P.E. Blower Jr., and R.H. Ledwith, "Extracting Reaction Information from Chemical Databases," in Ref. 6, pp. 367-381.
8. A.J. Gonzalez et al., "Automated Knowledge Generation from a CAD Database," in Ref. 6, pp. 383-396.

Dan Berleant is an assistant professor in the Computer Systems Engineering Department at the University of Arkansas.

Hal Berghel is a professor in the Computer Science Department at the University of Arkansas.

Correspondence can be addressed to either author. Berleant is at the Department of Computer Systems Engineering, 313 Engineering Hall, University of Arkansas, Fayetteville, AR 72701. Berleant's e-mail address is djb@engr.uark.edu; Berghel's e-mail address is hlb@acm.org

SOFTWARE

Software metrics:

Capers Jones,
Software Productivity Research

The software industry is an embarrassment when it comes to measurement and metrics. Many software managers and practitioners, including tenured academics in software engineering and computer science, seem to know little or nothing about these topics. Many of the measurements found in the software literature are not used with enough precision to replicate the author's findings — a canon of scientific writing in other fields. Several of the most widely used software metrics have been proved unworkable, yet they continue to show up in books, encyclopedias, and refereed journals. So long as these invalid metrics are used carelessly, there can be no true "software engineering," only a kind of amateurish craft that uses rough approximations instead of precise measurement.

Several widely used software metrics do not work, yet they continue to show up in books, encyclopedias, and refereed journals.

Software metrics that don't work. Three significant and widely used software metrics are invalid under various conditions: lines of code or LOC metrics, software science or Halstead metrics, and the cost-per-defect metric. The first two metrics are not invalid under all conditions, but they are when used to compare productivity or quality data across different programming languages. The third metric requires a

Capers Jones, Software Productivity Research Inc., 1 New England Executive Park, Burlington, MA 01803-5005;
(617) 273-0140; fax (617) 273-5176; Internet capers@spr.com; Compuserve 75430,231

Good, bad, and missing

careful separation of fixed and variable costs to produce valid information.

Lines of code. The widely used LOC metric is one of the most imprecise metrics ever used in scientific or engineering writing. After more than 50 years of use, it lacks a standard definition for any major programming language, and there are more than 400 programming languages in use.

The software literature, and even draft LOC counting standards, are about equally divided between those using physical lines and those using logical statements as the basis for the LOC metric. This difference alone can cause variations of 500 percent for some programming languages that allow multiple statements per physical line. Even worse, when used as a normalizing metric, LOC has been proven to penalize high-level languages, object-oriented languages, program generators, and every useful modern programming language.

If a manufacturing process includes substantial fixed costs, reducing the number of units manufactured increases the cost per unit. For software, more than half the effort is usually devoted to noncoding work such as requirements, design, user documentation, and management. If you define a line of code as a manufacturing unit, and then move from a low-level language to a high-level language, the paperwork costs tend to act like fixed costs and hence drive up the cost per line of code.

Here is an example. Suppose you're doing a project in assembly language that takes 10,000 lines of code. The requirements, design, documentation, and paperwork activities take five months, and the coding and testing take 10 months. So total project time is 15 months. The productivity of the entire project is 10,000 LOC divided by 15 months, or 666 lines of code per month. At $5,000 per staff month, the project costs $75,000 or $7.50 per LOC.

Now suppose you're doing the same project in Ada83. Since Ada is a much more powerful language, you'll need only 2,000 lines of code. The paperwork activities still take five months, but coding requires only two months. Now the total project amounts to seven months of effort, which is more than a 50 percent reduction compared to assembly language. Yet the apparent productivity of the project using the LOC metric is reduced: 2,000 LOC divided by seven months is only 285 lines of code per month. At $5,000 per staff month, the project costs only $35,000, but the cost per LOC has ballooned to $17.50.

It is hazardous to use a metric that gets worse as real economic productivity gets better.

Obviously, if used carelessly, LOC metrics penalize high-level languages, and the magnitude of the penalty is directly proportional to the level or power of the language. Only if all data is converted into "equivalent lines" in the same language can the LOC metric produce valid results across different languages. It is hazardous to use a metric that gets worse as real economic productivity gets better.

Halstead and cost per defect. The software science or Halstead metrics (invented by the late Maurice Halstead of Purdue University) are essentially a more convoluted way of dealing with code, by dividing it into operator and operand portions. These metrics share the same mathematical anomaly of lines of code and are equally unreliable.

The cost-per-defect metric is also misleading because of the impact of fixed costs. It actually penalizes quality and gets worse as quality gets better.

Suppose you are unit testing the assembly language example and find 100 bugs. Creating the test cases takes a week, running them takes another week, and fixing the 100 bugs takes one month. At a labor cost of $5,000 per month, unit test costs $7,500 or $75.00 for each bug found.

Now suppose the Ada version of the same project has only 10 bugs, an order-of-magnitude improvement in quality. Creating the test cases still takes a week, and running them still takes a week. Assume the bugs are fixed on the fly, with essentially no extra effort. Now unit testing takes only two weeks and costs only $2,500, for a savings of $5,000 compared to assembly language. But the cost per defect for the Ada version has jumped to $250. A 66 percent reduction in unit testing costs is a significant improvement, but it becomes invisible when using the cost-per-defect metric. Also, if only one bug were found instead of 10, then the cost per defect would explode to $2,500 because the entire cost of creating and running the test suite would be charged to that single bug.

It can easily be seen that careless use of the cost-per-defect metric penalizes quality and approaches infinity as quality approaches zero defects.

Metrics that do work. Fortunately, two metrics that actually generate useful information — complexity metrics and function-point metrics — are growing in use and importance.

Complexity metrics. We've known since the 1960s that excessive complexity in software usually raises defect potentials and reduces productivity of both development and maintenance.

This general statement can be quantified by cyclomatic complexity and essential complexity metrics, although there are other complexity metrics as well.

Of course, there are some exceptions to the "excessive complexity" rule. However, my own and my company's observations of several thousand projects over the past 15 years indicate that excessive complexity is often associated with lower than average quality and higher than average maintenance expenses. Surprisingly, much of the observed complexity appears to be technically unnecessary. In interviews with the programmers themselves, we find that excessive schedule pressure and hasty design tend to be a common root cause.

Various complexity metrics have been discussed and popularized by many researchers. Among those widely known are cyclomatic and essential complexity. Cyclomatic complexity is essentially the measure of the branches in the control flow of a program. Perfectly structured code that has no branching has a cyclomatic complexity of 1. If the control flow is graphed, cyclomatic complexity captures the basic branching structure. As the branching becomes more convoluted, this metric can indicate areas that may need simplification or restructuring, such as modules with a cyclomatic complexity above 20.

Essential complexity is derived from cyclomatic complexity and is a similar concept, except that it uses special graph-theoretic techniques to eliminate redundancy.

While it's possible to calculate cyclomatic and essential complexity manually, there's also no shortage of commercial tools to automate the work. Tools that directly scan source code are now available for most programming languages, and they operate on various platforms such as DOS, Windows, and Unix.

Function points. The function point metric, developed within IBM by A.J. Albrecht, has been in the public domain since 1979. It's not perfect, but it is free of the economic distortions of the LOC metric.

Function points are the weighted sum of five external attributes of software projects — inputs, outputs, inquires, logical files, and interfaces — that have been adjusted for complexity. The actual counting rules are quite complex, but they are logical and consistent enough to have been encoded in scores of automatic function-point counting tools.

It's easy to see why function points are now so widely used. The function point total for an application does not change with the programming language. For example, both the assembly example and the Ada example would contain the same function point total, since both versions actually perform the same functions.

Assuming that both versions contain 50 function points, productivity is 3.3 function points per month for the assembly language version and 7.14 for the Ada83 version. Economic costs can also be calculated with function points. The assembly language cost per function point is $1,500, while the Ada version is only $700 per function point. Now it's possible to see the economic advantages of the Ada version. For once, the industry has a metric that moves in the right direction when used with high-level languages.

Use of function points and associated tools has been exploding throughout the software industry. At least 30 commercial software-estimating tools now support function points. Indeed, 1994 will probably mark the first time in software history that every major commercial software-cost-estimating tool in the United States supports the function point metric. Some of the software estimating tools that already support functional metrics include (in alphabetic order) AEM, Asset-R, Bridge, BYL, Checkpoint, Cocomo II, Gecomo, Estimacs, Microman, Price-S, SEER, SLIM, SoftCost, and SPQR/20. Also, many upper CASE tools can now create function point totals automatically during the design phase. There are now probably more commercial tools supporting function point automation than for all other metrics combined.

For once, the industry has a metric that moves in the right direction when used with high-level languages.

Many of the commercial software-estimating tools also support a technique called "backfiring," a mathematical conversion of data between lines of code and function points. About 400 programming languages, and many combinations of mixed languages, can now be normalized using both LOC metrics and functional metrics by means of the backfiring approach. Using both metrics can clarify the problems with LOC metrics.

Missing metrics. One very important software domain — the data or information associated with software — lacks any metrics whatsoever. As of 1994, there are no known metrics for quantifying or normalizing the volume of data contained in a database or repository or used by a corporation. There are no metrics for dealing with the costs of creating, using, or removing data. There is no way to explore the volume of active data versus archival data, nor the costs of moving data back and forth between active and inactive status. There are no metrics for measuring data quality, although everyone who has ever worked with a database realizes that poor data quality is a critical problem.

Database consultants suspect that the costs of data creation and use in large companies are roughly equivalent to the costs of creating software. The value of the data may be even higher than the value of the software, but no one can explore these hypotheses until useful data metrics are invented.

Another domain with a severe shortage of useful metrics is that of hybrid applications that include hardware, microcode, and software. There are no metrics that can cross the hardware/software boundary and allow unified economic analysis at the complete system level.

Metrics and measurement are the basic underpinnings of science and engineering. Software has suffered through most of its 50-year history with inaccurate metrics and inadequate measurements. But as the 20th century draws to a close, improvements are rapidly occurring.

Recent developments in functional and complexity metrics and measurement are steps in the right direction, but the software community still has far to go before the phrase "software engineering" can be taken seriously.