

STRENGTHS AND WEAKNESSES OF SOFTWARE METRICS

Version 5
March 22, 2006

Abstract

The software industry lacks standard metric and measurement practices. Almost every software metric has multiple definitions and ambiguous counting rules. There are also key topics with no metrics at all, such as quantifying the volume or quality levels of data bases, data warehouses, and web sites. The result of metrics problems is a lack of solid empirical data on software costs, effort, schedules, quality, and other tangible matters. This report analyzes some of the key software size metrics and the underlying technical problems associated with software measurement.

Capers Jones, Chief Scientist Emeritus
Software Productivity Research LLC

Email CJones@SPR.com
Web www.SPR.com

Copyright © 1998 - 2006 by Capers Jones. All Rights Reserved.

INTRODUCTION

Measurement, metrics, and statistical analysis of data are the basic tools of science and engineering. Unfortunately, the software industry has existed for more than 50 years with metrics that have never been formally validated, and with statistical techniques that are at best questionable.

One of the fundamental measurement issues for software that should be put on a firm and rational basis is how to measure the size of various software deliverables. A full treatment of the size topic is too large for a short report, so this paper only summarizes the current state of software size metrics research.

What Software Artifacts Require Size Data?

Before turning to the available metrics for dealing with sizes, it is useful to step back and examine the kinds of items associated with software projects where size information is needed. There are 20 software artifacts where size information is important because the cost of creating or dealing with these deliverables is significant:

Table 1: Twenty Software Artifacts Requiring Size Metrics

- 1) The functionality of the application
- 2) The volume of information in data bases and web sites
- 3) The quantity of new source code to be produced
- 4) The quantity of changed source code, if any
- 5) The quantity of deleted source code, if any
- 6) The quantity of base code in any existing application being updated
- 7) The quantity of “dead code” no longer utilized but still present
- 8) The quantity of reusable code from certified or uncertified sources
- 9) The number of paper deliverables (plans, specifications, documents, etc.)
- 10) The sizes of paper deliverables (pages, words)
- 11) The number of national languages supported (English, French, Japanese, etc.)
- 12) The number of on-line screens
- 13) The number of graphs, and illustrations
- 14) The sizes of non-standard deliverables (i.e. music, animation)
- 15) The number of test cases that must be produced
- 16) The number of bugs or errors in requirements
- 17) The number of bugs or errors in specifications
- 18) The number of bugs or errors in source code
- 19) The number of bugs or errors in user manuals
- 20) The number of secondary “bad fix” bugs or errors

Some of the available metrics for quantifying the sizes of software deliverables include the following:

- Natural metrics such as counts of pages and words in paper documents.
- Source code metrics using physical lines of code.
- Source code metrics using logical statements.
- Function point metrics as defined by the International Function Point Users Group (IFPUG) organization.
- Function point variants as defined by other organizations.
- Object-oriented metrics of various kinds.
- Counts of bugs or defects.

A major omission from this list possible of metrics is the absence of any effective size metric for data bases, data warehouses, data quality, or repositories. As of calendar year 2006 the data base domain has no accurate method available to estimate sizes, quality, or costs. This is a major deficiency that needs immediate research.

There are also no convenient metrics for measuring web site “content.” Indeed, there is not even a standard definition of what “content” means in a web context, but it can include images, sounds, text, animation, and software such as Java scripts.

Another weakness of current metrics research is that of *metrics conversion*. As of 2006 there are no standard rules or certified methods for converting size from one metric to another. For example there is no standard method for converting size expressed in terms physical of lines of code into size expressed in terms of logical statements that is valid for all programming languages. There are no standard rules for converting source code size into function points or vice versa. for converting size expressed in terms of physical lines of code into size expressed in terms of logical statements. There are no rules for converting size expressed in terms of IFPUG function points into other “flavors” of function point metrics such as the British Mark II function points or the newer COSMIC function points.

The remainder of this report analyzes the strengths and weakness of the current set of software metrics:

Strengths and Weaknesses of Source Code Metrics

When the software industry began in the early 1950’s the first metric developed for quantifying the output of a software project was the metric termed “source lines of code” or SLOC. The variant “lines of code” or LOC has essentially the same meaning and is also widely used. However, almost at once some ambiguity occurred, because a “line of code” could be defined either:

- A physical line of code.
- A logical statement.

Physical lines of code are simply sets of coded instructions terminated by pressing the enter key of a computer keyboard. For some languages physical lines of code and logical

statements are almost identical, but for other languages there can be major differences in apparent size based on whether physical lines or logical statements are used. Unfortunately, the difference between physical lines of code and logical statements is often omitted from the software metrics literature.

Table 2 illustrates some of the code counting ambiguity for a simple COBOL application, using both logical statements and physical lines. This table shows the sizes of the various divisions of typical COBOL applications, and how physical lines of code counts and logical statement counts might vary:

Table 2: Sample COBOL Application Showing Sizes of Code Divisions Using Logical Statements and Physical Lines of Code

Division	Logical Statements	Physical Lines
Identification Division	25	25
Environment Division	75	75
Data Division	300	350
Procedure Division	700	950
Dead code	100	300
Comments	200	700
Blank lines	100	100
Total Lines of Code	1,500	2,500

As can be seen from this simple example, the concept of what actually comprises a “line of code” is surprisingly ambiguous. In the example the size range can run from a low of 700 lines of code if only logical statements in the procedure division are used, up to a high of 2,500 lines of code if a count of total physical lines is used. Almost any intervening size is possible. Most variations are in use for productivity studies, articles, books, and even for producing estimates for contracts.

The author once interviewed six managers in the same company whose offices were all within 150 feet of one another. When the question “how do you define a line of code” was put to these six managers, four distinct counting methods were noted and the range between the most concise method and the most diffuse method was almost 500%.

Bear in mind that table 2 is a simple example using only one programming language for a new application. The SPR catalog of programming languages (1) contains more than 600 programming languages and more are being added on a daily basis. Furthermore, a significant number of software applications utilize two or more programming languages at the same time. For example combinations such as COBOL and SQL or Ada and Jovial are very common. Almost half of large software systems contain more than one

programming language. The author has noted one system that actually contained twelve different programming languages.

Even more troublesome from the standpoint of counting lines of code, there are languages such as Visual Basic where “programming” can be done without using any lines of code at all. This is because Visual Basic supports a variety of pull-down menus, buttons, and other controls which allow programmers to develop features without resorting to conventional line-by-line coding.

There are other complicating factors too, such as the use of macro instructions, inclusion of copybooks, inheritance, class libraries, and other forms of reusable code. There is also ambiguity when dealing with enhancements and maintenance, such as whether or not to count the base code when enhancing existing applications.

Obviously with so many variations in how lines of code might be counted, it would be useful to have a standard for defining what should be included and excluded. Here we encounter another problem. There is no true international standard for defining code counting rules that encompasses all programming languages. Instead, there are a number of published local standards that unfortunately are often in conflict with one another.

Citing just two of the more widely used local standards, the Software Productivity Research (SPR) code counting rules published in 1991 are based on logical statements (2) while the Software Engineering Institute (SEI) code counting standards published in 1992 are based on physical lines of code (3). Both of these de facto standards are widely used and widely cited, but they differ in many key assumptions.

As an experiment, the author carried out an informal survey of code counting practices that had been published in software journals such as American Programmer, Byte, Application Development Trends, Communications of the ACM, Crosstalk, the Cutter Journal, IBM Systems Journal, IEEE Computer, IEEE Software, Software Development, and Software Magazine (4).

About a third of the published articles using LOC data used physical lines, another third used logical statements, while the remaining third did not define which method was used and hence were ambiguous in results by several hundred percent. Only about 10% of the articles reviewed had any definitions of the line counting rules utilized. While there may be justifications for selecting physical lines or logical statements for a particular research study, there is no justification at all for publishing data without stating which method was utilized!

It would be hard to imagine any other form of scientific or engineering literature that utilized metrics such as temperature, pressure, weight, or distance with as much ambiguity as the software literature.

Let us now consider the major strengths and weaknesses of common software metrics as used in the software literature. We will start with the strengths and weaknesses of physical lines of code, since this method is the oldest one in continuous usage in the software literature.

The main strengths of physical lines of code (LOC) are:

1. The physical LOC metric is easy to count.
2. The physical LOC metric has been extensively automated for counting.
3. The physical LOC metric is used in a number of software estimating tools.

The main weaknesses of physical lines of code are:

1. The physical LOC metric may include substantial “dead code.”
2. The physical LOC metric may include blanks and comments.
3. The physical LOC metric is misleading for mixed-language projects.
4. The physical LOC metric is ambiguous for software reuse.
5. The physical LOC metric is a poor choice for full life-cycle studies.
6. The physical LOC metric does not work for some “visual” languages.
7. The physical LOC metric is erratic for direct conversion to function points.
8. The physical LOC metric is erratic for direct conversion to logical statements.

When we turn our attention the metric consisting of logical statements, we find a slightly different pattern of strengths and weaknesses:

The main strengths of the logical statements are:

1. Logical statements exclude dead code, blanks, and comments.
2. Logical statements can be mathematically converted into function point metrics.
3. Logical statements are used in a number of software estimating tools.

The main weaknesses of logical statements are:

1. Logical statements can be difficult to count.
2. Logical statements are not extensively automated.
3. Logical statements are a poor choice for full life-cycle studies.
4. Logical statements are ambiguous for some “visual” languages.
5. Logical statements may be ambiguous for software reuse.
6. Logical statements may be erratic for direct conversion to the physical LOC metric.

On the whole, logical statements are a somewhat more rational choice for productivity and quality data based on coding, but neither physical lines of code nor logical statements are appropriate for exploration of non-coding work such as creation of specifications, user manuals, or project management tasks.

Indeed, when lines of code metrics are utilized for productivity studies that include non-coding activities such as the creation of requirements, specifications, and user manuals they paradoxically move backwards and give invalid results. This problem was first published by the author in the IBM Systems Journal in 1978 (Vol. 17, No. 1), and has been discussed in more than a dozen books and 100 articles since the initial publication. The LOC results are so misleading that starting in 1995 the author has stated that “using lines of code for productivity studies involving multiple languages and full life-cycle activities should be viewed as professional malpractice.”

An example can illustrate the paradoxical reversal of productivity when lines of code metrics are used to perform productivity comparisons between two different languages. Assume that two versions of the same application are created, with one being programmed in assembly language and the other version being programmed in the C++ language.

Assume that the programming staffs are equally skilled in both languages and have exactly the same salary levels. Both versions of the application are functionally identical, and differ primarily in the use of two different programming languages. Thus the version done in assembly language required 1,000,000 logical source code statements, while the version done in the higher level C++ language required only 500,000 source code statements to code the same feature set. Table 3 gives a side by side comparison of the two versions:

Table 3: Productivity Paradox With "Lines of Code Metrics"

	Assembly Source Code	C++ Source Code	Difference
Lines of source code	1,000,000	500,000	500,000
Activities	Staff Months	Staff Months	
Testing/Debugging	650	250	400
Paper Documents	600	550	50
Coding	550	250	300
Management	200	100	100
Total	2,000	1,150	850
Application cost	\$20,000,000	\$11,500,000	\$8,500,000
Lines of code per staff month	500	435	65
Cost per line of code	\$20.00	\$23.00	-\$3.00

In terms of real economic productivity, the C++ version is visibly superior since the C++ project cost only \$11,500,000 while the version done in assembly language cost \$20,000,00 or almost twice as much.

But when the two projects are compared using “cost per source line” or “lines of code per staff month” the assembler version looks best! This reversal of real economic productivity is the basis for the assertion that using “lines of code” metrics for cross-language comparisons constitutes professional malpractice.

The reason for this economic distortion has actually been understood for hundreds of years and is common knowledge among managers in every industry except software. When a manufacturing process is heavily impacted by fixed costs and there is a reduction in the number of units produced, the cost per unit must go up.

When “lines of code” are considered to be an economic unit and there is a switch from a low-level language programming language to a high-level programming language, it is obvious that the number of “units” will be reduced. But the costs of producing paper documents such as specifications stay constant and act as though they were fixed costs. Thus when two programming languages are compared using lines of code metrics, the lower-level language will falsely appear to be superior to the higher-level language.

Some researchers attempt to avoid this paradox by including only coding or core development activities in their measures, and excluding the activities that behave as fixed costs such as requirements, specifications, user manuals, and the like. Unfortunately, this approach excludes more than 50% of the total cost elements for large software projects and is not a valid solution to the distortions of using lines of code.

Since the paradoxical failure of LOC metrics to measure economic productivity has been known since the 1970’s, the author has long claimed that using LOC metrics for economic studies involving multiple programming languages should be viewed as professional malpractice. This is because software management practitioners should know enough about standard economics and measurements to realize the LOC metrics do not measure economic productivity.

The form of metric which allows non-coding work to be measured as effectively as coding work consists of functional metrics which are not related to source code volumes at all, but rather to the external features of the application as defined in user requirements.

Strengths and Weaknesses of Function Point Metrics

In the early 1970’s the IBM Corporation, which at the time was the world’s largest developer of software applications, commissioned a research team to develop a software metric that could be used for software economic studies regardless of what programming language, or combination of languages, were utilized for the code itself.

The IBM metrics research team was headed by Allan Albrecht, and the result of their research was termed the “function point” metric. Function points were used internally by

IBM in the mid 1970's, and then placed into the public domain at a presentation by Albrecht in October of 1979 at a conference in Monterey, California (5).

The function point total for a software application is enumerated from analysis of the requirements and specifications of the application, and consists of the weighted and adjusted totals of five key elements:

1. Inputs (screens, signals, etc.)
2. Outputs (screens, reports, checks, etc.)
3. Inquiries
4. Logical files
5. Interfaces

The actual rules for counting and adjusting function points are quite complex so training is needed to count function points accurately. The counting rules have passed from IBM, and are now controlled in the United States by the non-profit International Function Point Users Group (IFPUG). IFPUG publishes function point counting rules (6) and administers the certification exams. Passing the certification examination is a prerequisite for those who wish to become function point analysts.

Function points are now the preferred choice for software economic studies involving multiple programming languages and full life-cycle costs. For example function points actually match the real economic situation of the case study shown previously in table 3. If we assume that both the assembly language version and the C++ version perform the same functions and both are 4000 function points in size then we can perform a valid economic comparison. The assembly language version cost \$5000 per function point while the C++ version only cost \$2875 per function point, for a savings \$2125 per function point or an economic gain of 42.5%.

Many software researchers recognized the advantages of function points for software economic studies, but were not completely satisfied with the form and structure of the function point metric as originally defined by the IBM team, and more recently defined by the IFPUG counting practices committee.

In 1983, Charles Symons gave a presentation in London on an alternative function point variant which he termed the Mark II Function Point (7), which is now widely used in the United Kingdom and to a lesser degree in Hong Kong and Canada.

The Mark II function point alternative was only the first in a growing set of function point variants, which now include at least these 25 alternative functional metrics including the COSMIC function point. It is fairly obvious that 25 variations in how the function point metric might be counted is too many. The International Standards Organization (ISO) has developed a standard for function point sizing, but it is a fairly generic standard and does not specify which of the many function point variants might or might not be valid.

A common reason for developing alternative function point metrics is based on a misunderstanding of function point principles. Function points were developed as a unit of measure for expressing the size of software applications.

Other factors in addition to size determine the effort required to build applications. When effort and cost data for real-time and embedded software was compared to ordinary information systems applications, it was noted that information systems always seemed to have much higher productivity levels. This is due, of course, to the fact that information system applications do not usually deal with some of the complex timing and reliability issues associated with systems and real-time software.

Rather than accepting the fact that real-time and systems software is intrinsically difficult, several researchers developed alternative variants of function point metrics that returned larger values for software that was high in complexity. Thus some systems or real-time applications that might be sized at 1000 function points using IFPUG function points might balloon to 1500 or even 2000 function points when using some of the alternative function point metrics.

One of the variant metrics to function points is the “feature point” metric developed by the author in conjunction with Allan Albrecht in 1986. The feature point metric was created to deal with the psychological problem that members of the real-time and systems software world viewed function point metrics as being suitable only for management information systems.

The feature point metric was so named because telecommunications software engineers used the term “feature” for major functions being developed for switching systems. (The author, Capers Jones, had worked on telecommunications systems at ITT and had first-hand experience with the difficulty of getting systems engineers to adopt function points.) The feature point metric added one new parameter, algorithms, to Albrecht’s set:

1. Algorithms
2. Inputs
3. Outputs
4. Inquiries
5. Logical files
6. Interfaces

The feature point metric reduced the default weight for “logical files” from 10 to 4, and thus assigned a value of 6 to the new algorithm parameter. Thus the sum of algorithms and logical files totaled to the same default weight as standard function points.

Since telecommunications and systems software personnel with familiar with algorithms, the feature point method essentially gave them two things: 1) A functional size method not automatically associated with information systems; 2) The ability to deal with algorithms in an explicit fashion.

The purpose of feature points was psychological rather than technical. In fact, feature points and IFPUG function points were specifically designed to give the same final totals! Feature point metrics just arrived at that total by a slightly different route.

From 1986 to about 1990, the feature point metric did introduce the utility of functional metrics to many telecommunications and software engineers. However when the IFPUG organization began to provide counting rules for systems software, and included examples of counting systems software with standard function points, the psychological need for feature points began to fade away. Therefore both the author and Allan Albrecht began to recommend that standard IFPUG function points rather than feature points should be used for systems and telecommunications software from about 1995 forward.

The presence of so many variants all called “function points” means that serious benchmark and baseline studies must be careful not to mix up function point data based on the IFPUG standard with function point data from variants such as Mark II, COSMIC, full function points, or object points. This is not usually a problem because very little benchmark data exists in any of the variants. To date, the volume of data measured using IFPUG function points appears to be approaching 50,000 projects world wide. The volume of data using all of the variant function point metrics may be less than 500 projects world wide.

Instead of exploring the reasons why real-time software productivity is lower than information systems, it is unfortunate that some researchers preferred to develop alternative function point metrics which give real-time software larger apparent sizes than standard IFPUG function points and hence artificially elevate the productivity of real-time development. (The exception to this rule is the feature point metric, which yields the same totals as IFPUG function point metrics.)

Overall, when we examine the patterns of strengths and weaknesses of function point metrics, we see that for economic studies and for studies that include non-coding work such as specifications, function points are clearly superior to lines of code metrics.

The main strengths of function point metrics are:

1. Function points stay constant regardless of programming languages used.
2. Function points are a good choice for full-life cycle analysis.
3. Function points can measure non-coding activities such as documentation.
4. Function points can measure non-coding defects in requirements and design.
5. Function points are a good choice for software reuse analysis.
6. Function points are a good choice for object-oriented economic studies.
7. Function points are supported by many software cost estimating tools.
8. Function points can be mathematically converted into logical code statements for many languages.

The main weaknesses of function point metrics are:

1. Accurate counting requires certified function point specialists.
2. Function point counting can be time-consuming and expensive.
3. Function point counting automation is of unknown accuracy.
4. Function point counts are erratic for projects below 15 function points in size.
5. Function point variations have no conversion rules to IFPUG function points.
6. Many function point variations have no “backfiring” conversion rules.

Unfortunately when data is published using function point metrics, it is imperative that the authors identify which variety of function point is actually being used. Some of the major variants with published data include:

1. Function points as defined by IBM in 1979
2. Function points as defined by IBM in the 1984 revision
3. Function points defined by the International Function Point Users Group (IFPUG)
4. Mark II function points, widely used in the United Kingdom since 1983
5. 3D function points, which originated in the Boeing Corporation circa 1995
6. Full function points, which originated in Canada circa 1997
7. SPR function points, which originated in the U.S. in 1985
8. SPR feature points which originated in the U.S. in 1986
9. Netherlands function points, which originated in the Netherlands circa 1996
10. Engineering function points, which originated in the U.S. circa 1996
11. DeMarco function points of “Bang points” which originated circa 1982
12. Object points, which originated in the U.S. circa 1995
13. COSMIC function points originating in Canada and the U.K. circa 1998.

There are many metrics that have variants in use, but function point metrics have more variants than most. For example it is necessary to distinguish between nautical and statute miles; between temperature measured in Fahrenheit or Celsius, and between three separate ways of calculating gasoline octane ratings. However the competition among the function point variants has reached a point that there is some danger that the fragmentation of the metric into competing variants will damage its usefulness. (The author’s book Applied Software Measurement McGraw Hill 1996, contains a comparison of counting the same application using 12 different function point variants.)

Correlating Lines of Code and Function Points Metrics Via “Backfiring”

In the 1970’s Allan Albrecht and his colleagues at IBM measured a number of projects using both logical source code statements and function point metrics. These pioneering studies found some interesting but not perfect correlations between source code size and function points for many programming languages.

Table 4 illustrates some typical ratios of logical source code statements and equivalent volumes of function points. This is a small excerpt from the main SPR table of languages with more than 600 entries (1):

Table 4: Ratios of Logical Source Code Statements to Function Points for Selected Programming Languages Using Version 4.1 of the IFPUG Rules

Language	Nominal Level	Source Statements Per Function Point		
		Low	Mean	High
Basic assembly	1.00	200	320	450
Macro assembly	1.50	130	213	300
C	2.50	60	128	170
FORTRAN	3.00	75	107	160
COBOL	3.00	65	107	150
PASCAL	3.50	50	91	125
PL/I	4.00	65	80	95
ADA 83	4.50	60	71	80
C++	6.00	30	53	125
Ada 95	6.50	28	49	110
Visual Basic	10.00	20	32	37
SMALLTALK	15.00	15	21	40
SQL	27.00	7	12	15

Backfiring has become the most popular method for ascertaining function point sizes of aging legacy applications. In fact, for many legacy applications backfiring is the only convenient method for developing function point totals because the specifications are often missing and the original developers have departed.

Backfiring was quickly adopted by commercial software estimating vendors, and is now a common feature among most of the well-known software estimating products such as CHECKPOINT®, COCOMO II, GECOMO, KnowledgePlan™, and SLIM®.

Backfiring was also adopted for benchmark studies by a number of well-known consulting companies such as Compass Group, Gartner Group’s Real Decision subsidiary, Meta Group, Quantitative Software Management (QSM), Rubin Systems, Inc.; The David Consulting Group, and Software Productivity Research LLC.

Because backfiring is so often used in commercial software estimating tools and also by management consultants and commercial benchmark companies, it is arguably the most widely used software metric in the world.

The accuracy of backfiring is not claimed to be as high as normal function point counts. The accuracy of backfiring from logical statements is about plus or minus 20%, while normal counting by certified function point counters has been measured to range by about plus or minus 10% in a study commissioned by IFPUG and performed by Dr. Chris Kemerer (8) when he was at MIT.

It is a curious situation that neither the IFPUG organization nor any of the other major function point associations have evaluated backfiring, and the published literature on this method comes primarily from estimating tool vendors and software management consultants rather than the function point user community itself.

A number of software consulting companies publish backfiring data, but they do not always agree on the values for specific languages. Since these companies compete, they are not in a position to cooperate on sharing data or reaching agreements as to the values of specific languages. It is unfortunate that neither IFPUG nor any university has stepped up to becoming a neutral clearing house for coordinating the published values of backfired data.

The main strengths of backfiring function points are:

1. Backfiring is extremely quick and easy to perform.
2. Backfiring automation is commercially available.
3. Backfiring is supported by many software cost estimating tools.
4. Backfiring is used in many software benchmark studies.

The main weaknesses of backfiring function point metrics are:

1. Backfiring is of lower accuracy than normal function point counting.
2. Backfiring is ambiguous if the starting point is physical lines of code (LOC).
3. Backfiring may be ambiguous for mixed-language applications.
4. Backfiring results may vary based on individual programming styles.
5. Backfiring is not endorsed by any of the major function point associations.

Backfiring is perhaps the most common technique for determining function point values for aging legacy applications. The accuracy of this approach is below that of counts by certified personnel, but the speed and ease of generating results continue to make this approach popular.

Strengths and Weaknesses of Object-Oriented Metrics

As the object-oriented (OO) paradigm began to spread throughout the software community, it was quickly apparent that OO projects needed to be estimated and measured just as had procedural projects. Both lines of code metrics and function point metrics have been utilized with OO projects with varying degrees of success (9).

The fundamental differences in the way OO projects are constructed compared to procedural projects quickly led to new kinds of software metrics aimed exclusively at OO projects. Some of the specialized OO metrics include those of Kemerer and Chidamber (10) in the United States originally termed “metrics for object oriented systems environments” or MOOSE; and Abrieu and colleagues in Portugal with their “metrics for object oriented design” or MOOD metrics (11).

Some of the specialized OO metrics constructs include weighted methods per class, depth of inheritance tree, number of children, coupling between object classes, and a number of others.

In the past, both lines of code metrics and function point metrics have splintered into a number of competing and semi-incompatible metric variants. There is some reason to believe that the OO metrics community will also splinter into competing variants, possibly following national boundaries.

The main strengths of OO metrics are:

1. The OO metrics are psychologically attractive within the OO community.
2. The OO metrics appear to be able to distinguish simple from complex OO projects.

The main weaknesses of OO metrics are:

1. The OO metrics do not support studies outside of the OO paradigm.
2. The OO metrics do not deal with full life-cycle issues.
3. The OO metrics have not yet been applied to testing.
4. The OO metrics have not yet been applied to maintenance.
5. The OO metrics have no conversion rules to lines of code metrics.
6. The OO metrics have no conversion rules to function point metrics.
7. The OO metrics lack automation.
8. The OO metrics are difficult to enumerate.
9. The OO metrics are not supported by software estimating tools.

Unfortunately the OO metrics are totally unrelated to all other known software metrics. So far as can be determined, there are no conversion rules between the OO metrics and any other, so it is difficult or impossible to perform side by side comparisons between OO projects and conventional projects using the current crop of available OO metrics.

Summary and Conclusions

Software metrics research is an important topic, but not yet a well-formed or mature topic. Each of the major software metrics candidates has splintered into a number of competing alternatives, often following national boundaries. There is no true international standard for any of the more widely used software metrics. Further, the

adherents of each metric variant claim remarkable virtues for their choice, and often criticize rival metrics.

From a distance, the software metrics domain is fragmented, incomplete, and gives the appearance of being more influenced by “metrics politics” than by technical considerations.

References

1. Jones, Capers; Table of Programming Languages and Levels; Version 8.2; Software Productivity Research, Burlington, MA; URL <http://www.SPR.com>.
2. Jones, Capers; Applied Software Measurement; McGraw-Hill, New York, NY, 1996; 457 pages.
3. Park, Robert E.; SEI-92-TR-20; Software Size Measurement: A Framework for Counting Software Source Statements; Software Engineering Institute, Pittsburgh, PA; 1992; 220 pages.
4. Jones, Capers; Critical Problems in Software Measurement; IS Management Group, Carlsbad, CA; 1993.
5. Albrecht, A.J.; *Measuring Application Development Productivity*; Proceedings of the Joint IBM/SHARE/GUIDE Application Development Conference; October 1979; reprinted in Jones, Capers; Programming Productivity - Issues for the Eighties; IEEE Computer Society Press; 1986.
6. Symons, Charles; Software Sizing and Estimating - Mk II FPA; John Wiley & Sons; Chichester, UK; 1991.
7. Garmus, David (Editor); IFPUG Counting Practices Manual, Release 4.0; International Function Point Users Group (IFPUG); Westerville, OH; April 1994.
8. Kemerer, Chris F.; Reliability of Function Point Measurement: A Field Experiment; MIT Sloan School Working Paper 3192-90-MSA; January 1991.
9. Jones, Capers; *Economics of Object-Oriented Software*; Software Productivity Research; Burlington, MA; April 1997.
10. Chidamber S.R. and Kemerer, Chris F.: *Toward a Metrics Suite for Object Oriented Design*; OOPSLA 1991; pp 197-211.
11. Abreu, Fernando Brito e; *An email information on MOOD*; Metrics News, Otto-von-Guericke-Univeersitaat; Magdeburg; Vol. 7, No. 2; p.11; June 1997.

12. Jones, Capers; Conflict and Litigation Between Software Clients and Developers; Software Productivity Research, Burlington, MA; 2003; 54 pages.
13. Kan, Stephen H.; Metrics and Models in Software Quality Engineering, 2nd edition; Addison Wesley Longman, Boston, MA; ISBN 0-201-72915-6; 2003; 528 pages.