AUTHOR: Capers Jones
TITLE: Sizing Up Software
SOURCE: Scientific American v279 no6 p104-9 D '98

Unlike oil, steel or paper, software is an intangible commodity. This elusive quality makes computer programs difficult to quantify

Modern society has become increasingly reliant on software—and thankfully so. Computer programs routinely execute operations that would be extraordinarily laborious for an unaided person—handling payrolls, recording bank transactions, shuffling airline reservations. They can also complete tasks that are beyond human abilities—for example, searching through massive amounts of information on the Internet.

Yet for all its importance, software is an intangible quantity that has been devilishly tricky to measure. Exactly how should people determine the size of software?

The question is not just an academic exercise. Without a good gauge of software, the industry has had trouble improving the quality of computer applications and increasing the efficiency of their development. In fact, most companies and government agencies have only a hazy idea of the caliber of their software and the productivity of their programmers. Consequently, predicting the investment of money and time needed to create programs becomes such a difficult task that overruns and delays are the norm rather than the exception.

Indeed, software has proved to be a troublesome technology, as evidenced by the well-publicized computer debacle in Denver International Airport's baggage-handling system that set back the facility's opening by more than a year. In fact, until the so-called Year 2000 computer problem became pressing, most organizations did not know how much software they owned, even though it is a critical and costly asset. In my opinion, this basic problem of measurement is one of the biggest obstacles now facing the software industry.

Of course, one way to size up software would simply be to tally all the bytes that a particular program occupies in computer disk storage. Another approach would be to count the number of lines of "code," the lengthy list of instructions required for each computer application. Such measurements, however, do not always reflect how capable the software truly is. Sometimes a program with three million lines will in fact be richer in functionality and features than an application containing five million.

A better approach, then, is to assess the operations that a particular program performs. One formal method for doing so counts "function points," which are quantitative indicators of what a program can do. Computer scientists pioneered this formalism more than two decades ago, and their efforts may soon result in an international standard. Still, it is far from certain whether function points will eventually provide a universal system for measuring software.

WHAT EXACTLY IS SOFTWARE?

By themselves, digital computers are merely containers for the software that operates them. A player for compact discs acts similarly. With a disc containing a Mozart symphony in the player, the listener hears classical music. With one for Billie Holiday, jazz is heard. Likewise, with different software a computer can serve as a word processor, manipulate spreadsheets or read documents from the World Wide Web.

The software itself is a series of thousands or sometimes millions of individual commands that instruct a computer to take various actions, such as displaying information, performing calculations or storing data. To continue the music analogy, a programmer is like a composer. Just as the latter conceives music and then describes it note by note, the former envisions and then writes software instruction by instruction.

Here, though, the parallelism with music begins to fail. For one, composers use the same symbols and notation regardless of whether the music they are creating is a pop song or an operatic aria. Programmers, on the other hand, have at least 500 different languages at their disposal. Some computer languages, such as COBOL, are aimed at business problems. Others, such as FORTRAN, are geared for the mathematical and scientific fields. In addition, languages such as PL/I can be used for a wide range of applications.

The availability of so many diverse languages, each with its own structure and syntax, is one of several reasons why counting lines of code has proved to be difficult and unreliable. Specifically, there are no general rules or standards for determining exactly what constitutes a line of code in all programming languages. For a newer language, such as Visual Basic, the very concept has little meaning, because programmers typically use so-called graphical controls as well as code to establish how the application being developed will function. Furthermore, sometimes several languages are used within the same software.

And even if code could be counted with absolute precision, it would still not be a perfect solution. The similar approach of assessing software by simply measuring the amount of computer disk storage it occupies is even more flawed. Among various other reasons, the identical application written in different languages will take up very unequal amounts of storage space, even after the programs have been converted into the computer's binary world of 0s and 1s. Thus, the careless use of such bulk measurements—especially when applied to large programs—can produce deceptive results.

PERPLEXING PARADOX

Many applications used today consist of more than one million discrete statements. Some applications have more than 10 million. The expense of writing such large software is determined by considerably more than just the cost of the actual coding process. In practice, companies spend a greater amount of money to produce paper documents (specifications and user manuals) and to test the program and correct the errors that invariably turn up. Furthermore, the cost of managing a large software project can itself be quite steep.

The fact that much of the work of building large applications is not directly related to coding leads to a surprising paradox {see illustration below}. Suppose that two companies decide to create programs that do exactly the same things. One firm uses assembly language, which requires many instructions to handle basic tasks, such as adding one number to another. With such a low-level language, the application requires, say, one million lines of code. The second company uses COBOL, a business-oriented language that takes fewer statements to perform the same functions. With this high-level language, the program contains perhaps just 400,000 statements. These line counts would superficially indicate that the former software is more than twice as "large" as the latter even though both programs are, in effect, identical.

The comparison is complicated further if the programmers at the two hypothetical companies write code at different rates (as would be expected even if they have the same fundamental abilities). Because of the different languages used, each staff member at the first firm may be able to deliver 500 lines of code each month, whereas the comparable number for the second company might be 360. Even so, the second company would be able to develop the application faster—1,100 staff-months versus

2,000—because fewer lines need to be written and because programs in high-level languages typically require less debugging, among other reasons. As a result, the total cost of the program for the first company would be $20 million, and the corresponding expense for the second firm would be $11 million. Yet each line of code at the first company costs $20, as opposed to $27.50 for the second firm. So it might at first glance appear that the programmers in the first company are more productive even though the second company is able to develop the same application faster and more cheaply. Thus, a low cost per line of code does not necessarily indicate economic efficiency.

Clearly, blindly counting lines of code can be misleading. Computer scientists realized as much years ago, and consequently several research teams at IBM, one of the largest software development organizations in the world, began to explore other approaches to measuring software. One group, which included me, noted the productivity paradox in 1973, when we compared software written in assembly language with programs developed in PL/I. On average, the former required about four times as many statements as the latter. But both languages could be coded at roughly the same rate.

So our immediate solution to the problem of comparing apples with oranges was simply to multiply the PL/I line count by 4 to convert it into an equivalent number of assembly statements. Although this method provided a rough equivalency, it was far from ideal. Essentially we were doing nothing more than performing a monetary currency exchange, as if converting dollars into yen.

THE POINT OF FUNCTION POINTS

Other investigators at IBM in White Plains, N.Y., and elsewhere began a more fundamental research program. Allan Albrecht and his colleagues decided to formulate units of software complexity, which they dubbed function points. They designed this novel accounting method to be independent of programming languages.

To calculate function points, people examine five attributes of a program: its inputs, outputs, interactive inquiries (when the user issues a query and the computer returns a response), external logical files (interfaces) and internal logical files. Consider a simple spell checker that has one input (the name of the file that needs to be examined), three outputs (the total number of words reviewed, the number of errors found and a listing of the misspelled words), one inquiry (a user can interactively obtain the number of words processed thus far), one external file (the document that needs to be inspected) and one internal file (the dictionary). For this simple program, the number of elements is 1 + 3 + 1 + 1 + 1 = 7.

In practice, function-point analysis proves much more complicated than the mere counting of the five types of attributes. Practitioners of this method use various weights to account for the complexity (low, average or high) of each element, according to detailed guidelines maintained by the International Function Point Users' Group (IFPUG), an organization based in Westerville, Ohio. The actual calculations are quite involved, but the weighted sum of function points for the example of a simple spell checker would be 40 if each of the seven elements were of average complexity.

Last, the function-point total is either increased or decreased with a multiplier to match the perceived intricacy of the overall system. This adjustment is based on 14 factors, including, for instance, whether the processing of the application will be split across different computers and whether parts of the software need to be designed for reusability by future programs {see box on next page}.

The calculation of function points for complex software is fairly complicated; it typically requires specialists who have passed a certification examination. Few software managers need to know the actual mechanics of function-point counting, but all of them should understand the assessments of productivity and quality now being expressed with such measurements.

Because they are independent of the programming language used, function points provide the most accurate way to compare different software. For the earlier paradoxical example, if both the assembly language and COBOL applications contain 4,000 function points (the two programs, having the same functionality, will score an equal number of points), then the total cost for the first company is $5,000 per function point, compared with $2,750 for the second {see illustration on page 106}. This result gives a truer picture of the actual economies of the two projects.

Using function points, corporate managers can now conduct more reliable analyses of productivity and quality. Currently this methodology is the most widely used approach for such studies in the U.S. and in much of Europe. The total number of software projects worldwide measured using function points surpasses 100,000. My colleagues and I, among others, have used information from these studies to examine variations in productivity and quality by country, industry and programming language.

Such investigations have spawned many useful rules of thumb that can aid companies developing software, particularly before they undertake large projects. (If the software is completely specified at the outset, the total number of function points of a project can be calculated before even a single line of code has been written.) For instance, raising the function-point total of a system to the 1.25 power gives a ballpark estimate of the number of errors that will have to be removed. Dividing the number of function points by 150 approximates the number of programmers, software analysts and technicians who will be needed to develop that application, and raising the function-point total to the 0.4 power gives a rough estimate of the time in months that staff will need to complete the project. Obviously, these rules of thumb are no substitute for formal, detailed estimates, but they do provide preliminary numbers that can temper unrealistic and excessively optimistic schedules, which are a common problem in software development.

Despite such helpful information, the regrettable fact is that most organizations do not perform any kind of useful software measurements at all. Although more than 100,000 projects have been sized with function points to date, that number is below 1 percent of the total number of software applications in use. Among smaller firms, especially those with fewer than 100 software professionals, neither function points nor any other measurement techniques are yet widespread. But function-point usage has become quite common among large companies with more than 10,000 people involved in developing, testing and maintaining computer programs. For such corporations, software is a major cost, one that demands accurate measurement and estimation.

It is also interesting to look at the size of personal software applications. The word processor with which I wrote this article contains about 435,000 lines of code and has roughly 3,500 function points. In the course of daily business, an office worker might use more than 25,000 function points' worth of software in the form of spreadsheets, word processors, database applications, statistical tools and custom programs. All these applications run under the control of operating systems, which can top 100,000 function points in size.

NOT A PERFECT METRIC

The use of function points has expanded throughout the world, but not without some debate. A number of researchers have faulted the approach as too prone to human misinterpretation, for example, in assigning the various weights according to software complexity. Such subjectivity does exist, but the use of certification programs and exams has narrowed the range of counting variance to within about 10 percent. This margin of error is roughly in the same range as that associated with the counting of lines of code for widely used languages such as COBOL. (In fact, companies that lack standards for counting lines of code have experienced huge variances of more than 100 percent when different managers or programmers measured the same program.)

Many researchers have complained that the calculation of function points is too labor-intensive. This criticism is valid, but automated counting tools might soon ease the burden. Others have stated that because function points were developed for business information systems, they are not adequate for assessing other types of software. But function points have helped measure systems software, applications permanently programmed into various consumer devices and high-tech weapons. And computer professionals continually revise the rules for function points to include newer types of software, such as applications for the World Wide Web.

Standardization may be the greatest challenge. Many people in the software field have questioned some of the counting procedures in the original IBM approach as well as in the current method established by IFPUG. As a result, there are now about 20 variations of the technique.

In the early 1980s Charles R. Symons, a researcher with Nolan, Norton & Company in London, developed a scheme for counting function points called Mark II. Among other modifications, Mark II expands the number of adjustment factors from 14 to 19. Another adaptation, called feature points, is aimed at engineering and scientific applications. Because such software can be quite complex but sparse in the number of inputs and outputs, the use of traditional function points can sometimes lead to underestimation. Thus, the feature-point approach considers the number of algorithms in a program and reduces the number of adjustment factors from 14 to only three. Other offshoots include so-called 3-D function points, engineering function points, object points and full function points.

The International Standards Organization and the major associations of function-point users around the world are now working to draft a consolidated set of rules. Because representatives from various countries are involved, it appears likely that the final standards may embody concepts from IFPUG and from the major variations, if indeed standardization is possible.

In spite of the shortcomings, function points currently provide the best way to measure software. And the growing use of this formalism is indeed welcome, for without some kind of objective criteria, software development will have great difficulty in taking its place as a true engineering discipline rather than an artistic activity, as it has been for much of its history.

Added material

SOFTWARE ECONOMIC ANALYSIS requires a suitable method for measuring the size of computer programs; otherwise the results can be misleading. Consider the same application that is developed in two different computer languages, assembly and COBOL, with the former requiring more work, resulting in a higher cost (a). If the software is measured by counting the number of lines of programming code, the assembly version will appear to be more than twice as large as its COBOL counterpart (b), even though both do identical things. This distortion will make it seem that the assembly project has higher productivity (c) and is more cost-effective (d). But if the software is measured in function points, which indicate a program's capability {see top illustration on opposite page}, the two programs will be equal in size (e), leading to a truer picture of productivity (f) and cost (g).(BRYAN CHRISTIE)

FUNCTION POINTS provide a means to assess the size of a program in terms of its capability. The measurement requires the examination of five attributes of an application: its inputs, outputs, interactive inquiries, external files and internal files. For a simple spell checker, the number of such elements is seven, and each item needs to be weighted according to its individual complexity. The weighted sum of function points is then either increased or decreased to match the perceived intricacy of the overall program, as judged with 14 criteria {see box on next page}. The final total will thus indicate the functionality and complexity of the application.

SOFTWARE SIZE of typical programs, such as word processors and operating systems, can be compared with function points. Major defense systems are the largest kind of application, containing about 300,000 function points (roughly 27 million lines of code).(BRYAN CHRISTIE)

FILLING IN THE SCORECARD

The function-point methodology allows analysts to judge the complexity of a computer program using 14 criteria. The actual assessment, which requires people with special training who have passed a certification exam, is a detailed process with guidelines specified by the International Function Point Users' Group, based in Westerville, Ohio. The 14 factors, with corresponding examples of high-scoring programs for each, are listed below.

1. Complex data communications: A program for a multinational bank that must handle electronic monetary transfers from financial institutions around the world.
2. Distributed processing: A Web search engine in which the processing is performed by more than a dozen server computers working in tandem.
3. Stringent performance objectives: An air-traffic-control system that must continuously provide accurate, timely positions of aircraft from radar data.
4. Heavily used configuration: A university system in which hundreds of students register for classes simultaneously.
5. Fast transaction rates: A banking program that must perform millions of transactions overnight to balance all books before the next business day.
6. On-line data entry: Mortgage approval program for which clerical workers enter data interactively into a computer system from paper applications filled out by prospective home owners.
7. User-friendly design: Software for computer kiosks with touch screens in which consumers at a subway station can purchase tickets using their credit cards.
8. On-line updating of data: Airline system in which travel agents can book flights and obtain seat assignments. The software must be able to lock and then modify certain records in the database to ensure that the same seat is not sold twice.
9. Complex processing: Medical software that takes a patient's various symptoms and performs extensive logical decisions to arrive at a preliminary diagnosis.
10. Reusability: A word processor that must be designed so that its menu tool bars can be incorporated into other applications, such as a spreadsheet or report generator.
11. Installation ease: An equipment-control application that nonspecialists will install on an offshore oil rig.
12. Operational ease: A program for analyzing huge numbers of historical financial records that must process the information in a way that would minimize the number of times computer operators have to unload and reload different tapes containing the data.
13. Multiple sites: Payroll software for a multinational corporation that must take into account the distinct characteristics of various countries, including different currencies and income tax rules.
14. Flexibility: A financial forecasting program that can issue monthly, quarterly or yearly projections tailored to a particular business manager, who might require that the

information be broken down by specific geographic regions and product lines. (BRYAN CHRISTIE)

COST OF SOFTWARE can be investigated using function points. This chart compares the expense of building different types of software: military applications; scientific and engineering programs; commercial packages, such as Microsoft Word; information systems used by companies to run their business operations; and personal applications developed by nonspecialists. Cost ranges are given with average values indicated.

The Author

CAPERS JONES is chief scientist and an executive vice president of Artemis Management Systems in Boulder, Colo., which recently acquired Software Productivity Research in Burlington, Mass., a consultancy that he founded. He is the designer of several tools for estimating the cost and quality of software and the author of 11 books on programmer productivity and software quality. Jones was formerly an assistant director at the ITT Programming Technology Center in Stratford, Conn. Before joining ITT, he was with IBM for 12 years, where he received an outstanding contribution award for his work in software quality and programmer productivity. He is a member of the IEEE Computer Society, IFPUG and ISPA.

FURTHER READING

AD/M Productivity Measurement and Estimate Validation. Allan Albrecht. IBM Corporation, Purchase, N.Y., May 1984.

Software Sizing and Estimating: Mk II FPA (Function Point Analysis). Charles R. Symons. John Wiley & Sons, 1991.

IFPUG Counting Practices Manual. Release 4. International Function Point Users' Group, Westerville, Ohio, April 1995.

Metrics and Models in Software Quality Engineering. Stephen H. Kan. Addison-Wesley, 1995.

Applied Software Measurement: Assuring Productivity and Quality. Second edition. Capers Jones. McGraw-Hill, 1996.

Function Point Analysis: An Empirical Study of Its Measurement Processes. A. Abran and P. N. Robillard in IEEE Transactions on Software Engineering, Vol. 22, No. 12, pages 895-909; December 1996.