ESTIMATING SOFTWARE SIZE: IMPACT AND METHODOLOGIES

Timothy J. Hohmann
Galorath Associates, Inc.
El Segundo, CA

ABSTRACT

This paper discusses the impact of size on software development in relation to other major cost and effort drivers, and describes current methodologies for estimating size. Automation, through computer hardware and software, has enabled great improvements in training, by making training more efficient and more effective. Computer-based training packages have shown great potential for cost savings as well as improvements in the quality and consistency of training delivery. And embedded training routines in weapons and tactical information systems allows trainees to conduct realistic exercises on the same hardware that they will use in combat. Estimating the cost of training software development is critical to developing specifications and making informed decisions about program development. Although many factors influence software development, software program size is one of the key determinants of cost and effort. This paper discusses personnel capabilities, development resources and specification characteristics, and the relative impact of size on cost & effort is demonstrated. The paper also describes commonly used software size measures. Size metrics discussed include source lines of code (SLOC) as well as functionality measures (function point analysis). It also discusses methods and demonstrates tools for developing accurate estimates of software size. Various size estimation methods are briefly described, and the SEER-SSM software sizing tool is demonstrated and discussed in detail. SEER-SSM uses a relative comparison methodology with reference programs of known size to develop accurate, probabilistic estimates of software size.

AUTHOR BIOGRAPHY

Mr. Timothy Hohmann is a consulting manager and director of training for G A Consulting Division of Galorath Associates, Inc. in El Segundo, CA. Galorath Associates produces the SEER family of parametric cost estimation tools and provides cost estimation support to military, government and commercial clients. In addition to developing and conducting training for the SEER line of parametric estimation tools, Mr. Hohmann is involved in cost analysis for both software and hardware development projects.

Mr. Hohmann spent ten years in the U.S. Navy as a Surface Warfare Officer, and was actively involved in training in all of his assignments. He is currently a Lieutenant Commander in the Naval Reserve. He also teaches information systems management courses for the University of Phoenix in the undergraduate Business and MBA/Technology Management curricula. He holds a BS degree in History from Iowa State University, and a MS in Information Systems Management from the University of Southern California.

ESTIMATING SOFTWARE SIZE: IMPACT AND METHODOLOGIES

Timothy J. Hohmann
Galorath Associates, Inc.
El Segundo, CA

THE IMPORTANCE OF SOFTWARE COST ESTIMATION IN TRAINING DEVELOPMENT

Military, government and commercial organizations all face a similar dilemma in terms of training. The importance training has long been recognized. However, in today's tight budgetary climate, training dollars are often near the top of the list to be cut. Consequently, organizations must figure out how to provide good, effective, realistic training in a cost-effective way.

Increasingly, the solution to this problem is automation—computer-based training, distance learning and online instruction, simulation systems and embedded operator training modules in operational systems. While automated training can be very cost effective once established, it requires a considerable up-front investment, much of which is attributable to software development. The effective management of software development cost and risk is thus of interest to anyone involved in training development.

FACTORS INFLUENCING SOFTWARE DEVELOPMENT COST

Many factors affect software development, and these factors have different impacts on the cost of development. The most critical step in estimating the cost of a software development project is determining the overall scope of the project. Important factors influencing scope include: the capabilities and experience of the development team; the tools, methods and practices used; the inherent complexity of the application under development; and the overall size of the application.

The Development Team

The capabilities and experience of the systems engineers and programmers responsible for developing software can have a major impact on productivity and cost. Better people will tend to produce more effective software, with fewer bugs, in less time and with less effort. Although it is important to use the most capable individuals available, it is equally important that they work as a team. Software development is an intensely collaborative undertaking, and a poorly functioning or poorly managed team will not be productive, no matter how brilliant the individual members.

Development Tools and Environment

Similarly, the tools, practices and methods available to the development team can affect productivity. Structured programming techniques and CASE tools, coupled with firm requirements and stable target hardware will improve productivity and lower costs. Implementing better tools and methods can, however, be a double-edged sword. Upgrading equipment and tools in the development environment and instituting modern programming practices will normally pay off in the long run, but may result in lower productivity as the development team climbs the learning curve.

Complexity

As may be expected, some types of applications are inherently harder to implement than others. A simple hierarchical database may be relatively easy to program, while a larger, more complex , relational database program will be more difficult. A real-time simulation program with a 360 degree display, control of six-degree motion and intense user interactivity will be even more complex. The language chosen for development will also impact its complexity. Coding in most 4^{th} generation languages, for instance, may be relatively simple and easy to understand, enhancing productivity. Implementing the same functionality in a language like Ada, or directly in mainframe assembly language, would be more difficult.

ESTIMATING SOFTWARE SIZE: IMPACT AND METHODOLOGIES

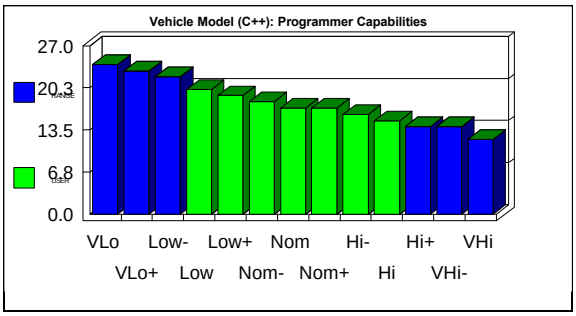

Figure 1. Effort Sensitivity to Programmer Experience

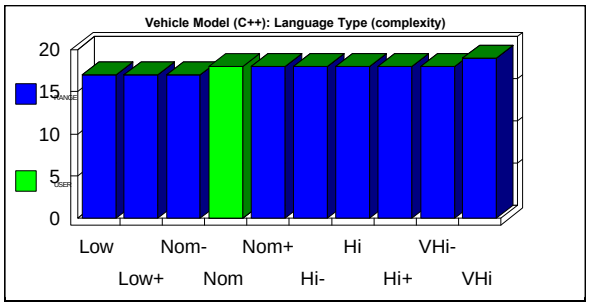

Figure 3. Effort Sensitivity to Language Complexity

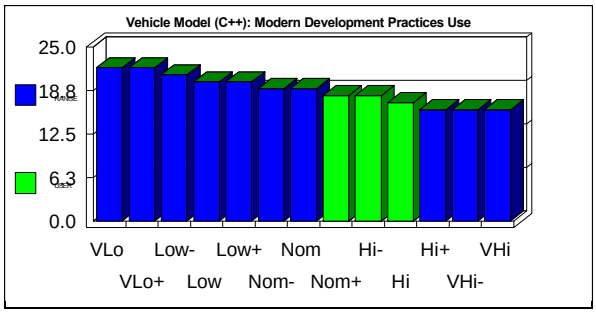

Figure 2. Effort Sensitivity to Modern Development Practices

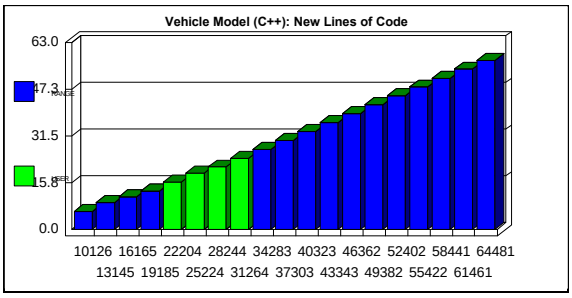

Figure 4. Effort Sensitivity to Size

Software Size

Software size is a measure of how "big" a program is. While this may seem self-evident, the actual concept of software size is very slippery. Many measures exist, and two of the most common are described below. Each measure attempts to capture and quantify the work or functionality performed by a piece of software. Software that does more things, or does more complex things, or does things more quickly or more reliably will normally be "larger." Size will also vary depending on a number of other factors (Humphrey, 1989), such as:

Code source:
- New
- Modified
- Reused

Language type:
- Assembly
- High-level
- Object code

Ancillary code:
- Comments
- Patches
- Test and debug code
- Support Code

Relative Impact on Cost

Figures 1 through 4 show the impact of several of the factors listed on effort, as estimated by a software cost model. As can be seen, changes in size have significant impact on effort, and therefor on cost and schedule. Furthermore, much of the impact of other factors varies proportionally with size, and so can be described as a factor of size. For this reason, software estimating models like SEER-SEM, COCOMO, PRICE-S and SLIM use size or volume of the end product as a primary input. It is obvious, then, that an accurate estimate of program size is one of the most critical factors in developing an accurate estimate of software project cost, effort and schedule.

MEASURES OF SOFTWARE SIZE

In order to estimate software size, some unit of measurement must be chosen. Measuring or determining software size is, however, more difficult than it might sound. Even a concept as simple as counting lines of code becomes more complicated when factors such as language, programming style, and non-source code must be considered. Various methods have been developed to adequately describe the size or volume of a program, and different methods may be appropriate for different estimating situations. The two most common size metrics currently in use are lines of code and function points.

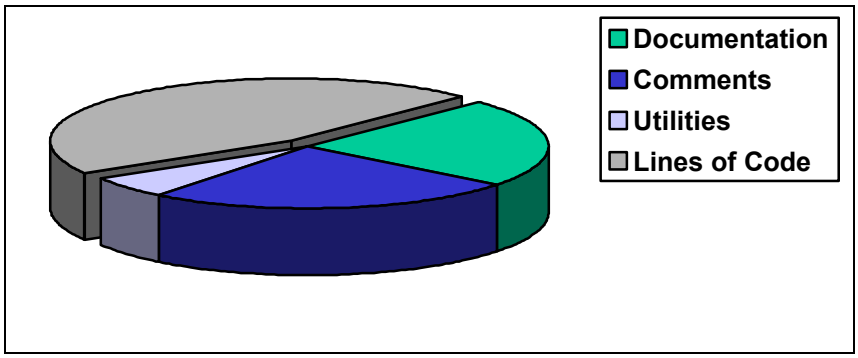

Figure 5. Importance of proper line definition

Lines of Code

One of the oldest and most popular methods of describing software size is by measuring the lines of programming instructions (commonly called lines of code (LOC), source lines of code (SLOC), executable lines of code (ELOC) or delivered source instructions (DSI)). This size metric can be used in almost all current estimating algorithms and tools. One advantage of line-based metrics is that they have been used for many years, and are easily measured for existing programs. Thus a large database of programs of known size exists.

What Is A Line? Although counting lines sounds like a simple process, it can actually be quite difficult, since different interpretations of "line of code" may be used. For instance, should blank lines be counted? Comment lines? Debug code? When developing a line-based measure, it is critical to define what counts and what does not. Some common conventions include (Jones, 1986):

Executable Lines only
Executable lines and data definitions
Executable lines, data definitions and comments
Executable lines, data definitions, comments and job control language (JCL)
Physical lines on an input screen
Logical delimiters, such as semicolons

Figure 5 shows the how the difference in line definitions affected one real-world project (Galorath Associates, 1997). The original size estimate presented for cost estimation of this project was 1.6 million lines. The cost estimation tool used discounted documentation, comments and utilities, which were included in the original estimate and accounted for 54% of that estimate. Obviously, using the original size estimate with this tool would have resulted in an unrealistically high estimate of cost and schedule. Other investigators have postulated differences of up to 500% due to line counting variations (Matson, Barrett & Mellichamp, 1994; Bozoki, 1986).

How Much Effort Is Associated With A Line? In addition to the distinctions in types of lines mentioned above, line based measures must also account for the progeny of a line. If the effort to write a new line of code is taken as the basic unit, different levels of effort will be expended in including modified or reused lines in the product. Most commonly, the effort associated with modifying existing code will be less than that required to write new code. However, if significant integration, reverse engineering, redesign, revalidation, retesting and redocumentation are required, it may be more expensive to use pre-existing code than to build it from scratch. Effort associated with each line should also consider

more than just the effort to write it. In addition to the direct coding effort, design, documentation, testing and QA effort should be considered. Coding itself may account for as little as 10-15% of total effort (Emrick, 1987).

How Much Functionality Is Associated With A Line? Language also accounts for differences both in the effort to write a line of code and the functionality contained in each line. A very complex language may require a great deal of effort to write relatively few lines. Conversely, the functionality contained in relatively few lines of a higher order language may require several times as many lines to implement in assembly language. In addition, programming styles and conventions can affect the size of a program. For instance, writing two modules that do exactly the same thing will double the lines of code used with little or no increase in functionality. Simply breaking a "do" loop into individual serial commands might expand the size of a program by many times with no additional functionality.

Consistency Is The Key. When using line-based sizing to estimate effort, it is most important to ensure that a common definition of a line is used—to know what is and is not included, and use the same definition for all programs under consideration. Particularly when using estimation models and automated tools, it is critical to ensure that the line definition used in the size estimate is the same as the definition used by the model.

Function Points

Recognizing the need for a more language- and programmer- independent size metric, Allan Albrecht developed the function point metric at IBM in the mid 1970s. This metric proposed to measure size based on what the program does, rather than how many instructions it contains. Albrecht's methodology has been extended, updated and codified by the International Function Point Users Group (IFPUG), which defines counting procedures, publishes a counting manual and conducts training and provides certification in function point counting.

Used as a basic unit of cost, a function point is independent of the language or programming style used. This metric is particularly useful in cost estimation since the basic functionality of a program can usually be defined early in the project life cycle—in the concept and requirements phases—when accurate cost estimates are most useful.

What Are Function Points? Function points are basic measures of "goods and services" (Jones, 1995) that the user receives from the software, based on its logical design. Function points types are described in table 1 (International Function Point Users Group, 1994).

Function Type	Definition	Examples
External Inputs (EI)	Inputs to the application	Input screens, interactive inputs, batch input streams, hardware inputs
External Outputs (EO)	Outputs from the application	Output screens, batch outputs, printed reports, hardware & software outputs
External Inquiries (EQ)	User inquiries	Menus, context-sensitive help, embedded inquiries
Internal Logical Files (ILF)	Data files updated by the application	Data tables, database files
External Interface Files (EIF)	Interfaces to other applications, data files not maintained by the application	Shared data files, reference data, fixed messages

Table 1. Function Types

Use Line of Code Measures when	Use Functional Measures when
Lines can be counted (preexisting software)	New or unique program
Similar programs of known size available	Early in development cycle—no design work done
Expert judgement is available	Detailed requirements & specifications available
Comparison with actual results is important	Considerable input-output or file activity
Considerable real-time or logical code	Trained & experienced in counting function points

Table 2. Line vs. Function Based Sizing

In addition to the raw number of function points in a program, function-based size considers the relative complexity of each function. For transactional functions (EI, EO and EQ), the number of different file types and data element types referenced by the function are counted, and a complexity rating is assigned. For data function types (ILF and EIF), data element types and record element types are considered. These complexity factors are used to rate each function type's contribution to the overall count. The resultant count is known as the unadjusted function point count.

This unadjusted count is then further weighted by a Value Adjustment Factor, which estimates the general functionality of the application based on fourteen general system characteristics such as data communications, transaction rate, complex processing, ease of installation and multiple sites. The calculated value adjustment factor is applied to the unadjusted count to arrive at the final adjusted function point count. These function points can then be used as effort units, much as a line of code is used.

Extensions to Function Points

A major criticism of the function point methodology is that while it works well for transactional and MIS systems, it is less useful for characterizing applications like operating systems, real time and embedded software which include significant logical activity. Several extensions to the basic methodology have been developed to account for this shortcoming, including SEER-FBS (Galorath, 1993) and Feature Points (Jones, 1995). Methods have also been developed to "backfire" function based size estimates into lines of code (Jones, 1996).

SIZE ESTIMATION METHODS

Estimation Approach

In estimating software size, the one decision which must be made is the basic approach to estimation. Two popular approaches are the "bottom-up" approach and the "top-down" methodology. In the bottom up approach, the application is deconstructed into its component parts, and the size of each part is estimated. The sum of the size of the parts is then the size of the whole. The top-down approach takes a more global view of the system, arriving at a size estimate based on the functionality of the entire program. The overall size can then be distributed among the component parts. Using both approaches on the same application and reconciling the resulting estimates can be a good technique for exposing all the issues involved in sizing. The choice of approach(s) to be used should be based on the information and expertise available to the estimator.

Estimation Methods

Many methods for estimating software size have been developed, and several are described below. Most of these methods can use either lines of code or function based metrics, so the estimator should choose the size metric most appropriate to the application and available information. It is important to understand that short of counting lines in a finished program, no sizing method will be 100% accurate. Any point value for a size estimate should be accompanied by an estimated range of probability.

Expert Judgement. Perhaps the most basic method of estimating size is simply to ask the expert. Experienced software engineers and programmers can frequently offer an estimate of size "off the top of their heads," with very little effort required. Especially when the application type and development environment are familiar to the estimator, expert judgement estimates can be uncannily accurate. Expert judgement also forms the basis or input for any more sophisticated analysis, so deriving an expert judgement estimate at some level of accuracy is an important first step in any estimation of size.

Consensus of Experts. Expert judgement estimates can be further refined using an iterative

process known as the Delphi technique, originated by the Rand Corporation (Humphrey, 1989). This method requires several experts, and is most accurate when applied in a "bottom-up" approach, breaking down the application into the smallest possible elements for estimation. The steps in the Delphi process include:

1. Each expert is given all available information about the application to be estimated. Experts are usually also given an estimation form, in order to keep all estimates in a common format

2. Experts discuss the product and any estimation issues. (Note: this is an optional step known as the Wideband Delphi Technique. In the classic Delphi method, experts work completely independently).

3. Each expert completes an estimation form.

4. Estimates are tabulated and collated. Results are returned to each expert identifying the expert's estimate, the mean of all estimates, and the range of estimates (highest and lowest). All individual estimates remain anonymous except the expert's own.

5. The experts may again meet to discuss the results of the round.

6. Experts reassess their estimates individually, based on the results of the previous round.

7. The process is repeated. With each repetition, the range can be expected to narrow until eventually an acceptable mean value is reached.

One particular advantage of the Delphi technique is that it tends to normalize biases and hidden agendas of the individual estimators. As long as a representative group of estimators is chosen, the effect of overoptimism, overpessisimism, political factors or other extraneous issues often associated with expert judgement estimates can be reduced or eliminated.

Relative Comparison/Analogy. Sizing by analogy involves comparing the application to be estimated to one or more programs of known size. It can be inferred that programs that are implemented in a similar language and environment and perform similar functions will be similar in size. It is also much easier for the human expert to make relative judgements of size rather than absolute judgements. Where an expert may have trouble developing an ***a priori*** estimate of program size, he may be able to make a very accurate assessment of whether a program is larger or smaller (and even "much larger" or "slightly smaller") than a properly described reference program.

Reference programs are crucial to the analogy method. Size of the reference program must of course be known. It is also important to understand the metric used to describe the reference program's size. In order to make a fair comparison, the size metric of the reference and estimated applications must be normalized. Lastly, it is important that the functionality and characteristics of a reference program be described adequately, so that differences or similarities in function can be translated into differences or similarities in size.

Analytic Methodologies. Analytic methods attempt to further refine size estimates supplied through expert judgement or analogy, applying statistical analysis in order to arrive at a probabilistic estimate of application size. Analytic techniques are particularly useful in constructing risk-based estimates. One sizing tool, SEER-SSM, applies several techniques using one or more reference modules to develop a ranking matrix, from which is developed a probabilistic estimate of size (Bozoki, 1993). The validity and accuracy of this method has been demonstrated in several studies (Lucas, 1992; Bozoki, 1991 & 1992).

Several of these methods are combined in the SEER-SSM sizing model. The model uses four separate techniques to compare modules to be estimated with a number of reference modules of known size. It then combines the four data sets into single, probabilistic estimate of size.
The first data set results from a random pairwise comparison. The model chooses pairs of unknown and reference modules randomly, and asks the user to assess which is larger. The model then places the unknown and reference modules on an ordinal scale in order of size.

The second comparison uses the PERT technique to assess the size of the unknown module or

modules. PERT uses a weighted average of expert judgement estimates for unknown modules defined by the following equation:

$$MEAN = \{S + (4 * L) + H\} \div 6$$

Where:

S = the smallest estimate
L = the most likely estimate
H = the highest estimate

The model uses the size of the reference models and the PERT mean size estimate of the unknown modules to place all modules on another ordinal scale in order of magnitude.

Thirdly, the SSM model uses a sorting technique, constructing monotonically increasing size intervals and asking the user to place each unknown module in the correct interval.

Finally, a ranking comparison is performed. This process is similar to the pairwise comparison, except that ordered vice random pairings are presented.

The result of these four techniques is four independent ordinal rankings of reference and unknown modules, each of which might be expanded into an independent size estimate. The SSM model, however, uses these scales to form a ranking matrix. The ranking matrix is then transformed into an interval scale using the Logarithmic Least Squares Method (also known as the Geometric Mean Scale) (Rand Corporation, 1985). Once all modules are placed on an interval scale, ratios between sizes can be established and the known size for the reference modules can be used to derive size for the unknown modules.

Criteria for Assessing Sizing Methodologies and Models

Choice of a size estimation method and a related sizing model or tool should be based on a number of factors. The amount, type and character of information available must be considered, as should the validity and usability of the method chosen. Table 3 summarizes the key areas and criteria to consider when choosing a sizing methodology or tool (Bozoki, 1986).

Area	Criteria
User Input	Is required knowledge available? How accurate or firm is input data? Quantitative or qualitative inputs required? Level of training or expertise required?
Use of Historical Data	Uses earlier user data for subsequent estimates? Statistical or database dependent? Support for sensitivity analysis?
Underlying Concepts & Algorithms	Open model or black box? Applicable to current development environments? Historical validation of results?
Model Output	Probability ranges? Summary of user inputs?
Usability	User-friendly interface? Availability of support?

Table 3. Criteria for Assessing Sizing Methodologies

REFERENCES

Bozoki, G. J. (1986). Software Project Sizing Workshop. (Available from GA SEER Technologies, 100 N. Sepulveda Blvd, El Segundo, CA 90245)

Bozoki, G. J. (1991). Performance Simulation of SSM (Software Sizing Model), Proceedings of the 13th Annual Conference of the International Society of Parametric Analysts, CM-14.

Bozoki, G. J. (1992). A Trade Study of PERT and SSM Software Sizing Models, Proceedings of the 14th Annual Conference of the International Society of Parametric Analysts, SW41-SW61.

Bozoki, G. J. (1993). An Expert Judgement Based Software Sizing Model. Journal of Parametrics, XIII (1).

Emrick, R. D. (1987). In Search of a Better Metric for Measuring Productivity of Application Development, Proceedings: International Function Point Users Group Conference.

Galorath Associates, Inc. (1997). SEER-SEM Workbook. (Available from GA SEER Technologies, 100 N. Sepulveda Blvd, El Segundo, CA 90245)

Galorath, D. D. (1993). Function Based Sizing, Proceedings of the 15th Annual Conference of the International Society of Parametric Analysts, K29-K33.

Humphrey, W. S. (1989). Managing the Software Process. Reading: Addison-Wesley.

International Function Point Users Group. (1994). IFPUG Function Point Counting Practices Manual (Release 4.0). (Available from the International Function Point Users Group, 5008-28 Pine Creek Drive, Westerville, OH 43081)

Jones, T. C. (1986). Programming Productivity. New York: McGraw-Hill.

Jones, T. C. (1995). What Are Function Points? (Available from Software Productivity Research, Inc., 1 New England Executive Park, Burlington, MA 01803)

Jones, T. C. (1996). Programming Languages Table, Release 8.2. (Available from Software Productivity Research, Inc., 1 New England Executive Park, Burlington, MA 01803)

Lucas, S. (1992). Software Size Estimation Using SEER-SSM, Proceedings of the 14th Annual Conference of the International Society of Parametric Analysts, SW27-SW40.

Matson, J. E., Barrett, B. E., & Mellichamp, J. M. (1994). Software Development Cost Estimation Using Function Points. IEEE Transactions on Software Engineering, 20, (4), 275-287.

Rand Corporation. (1985). The Analysis of Subjective Matrices (R-2572-1-AF). Crawford, G and Williams, C.