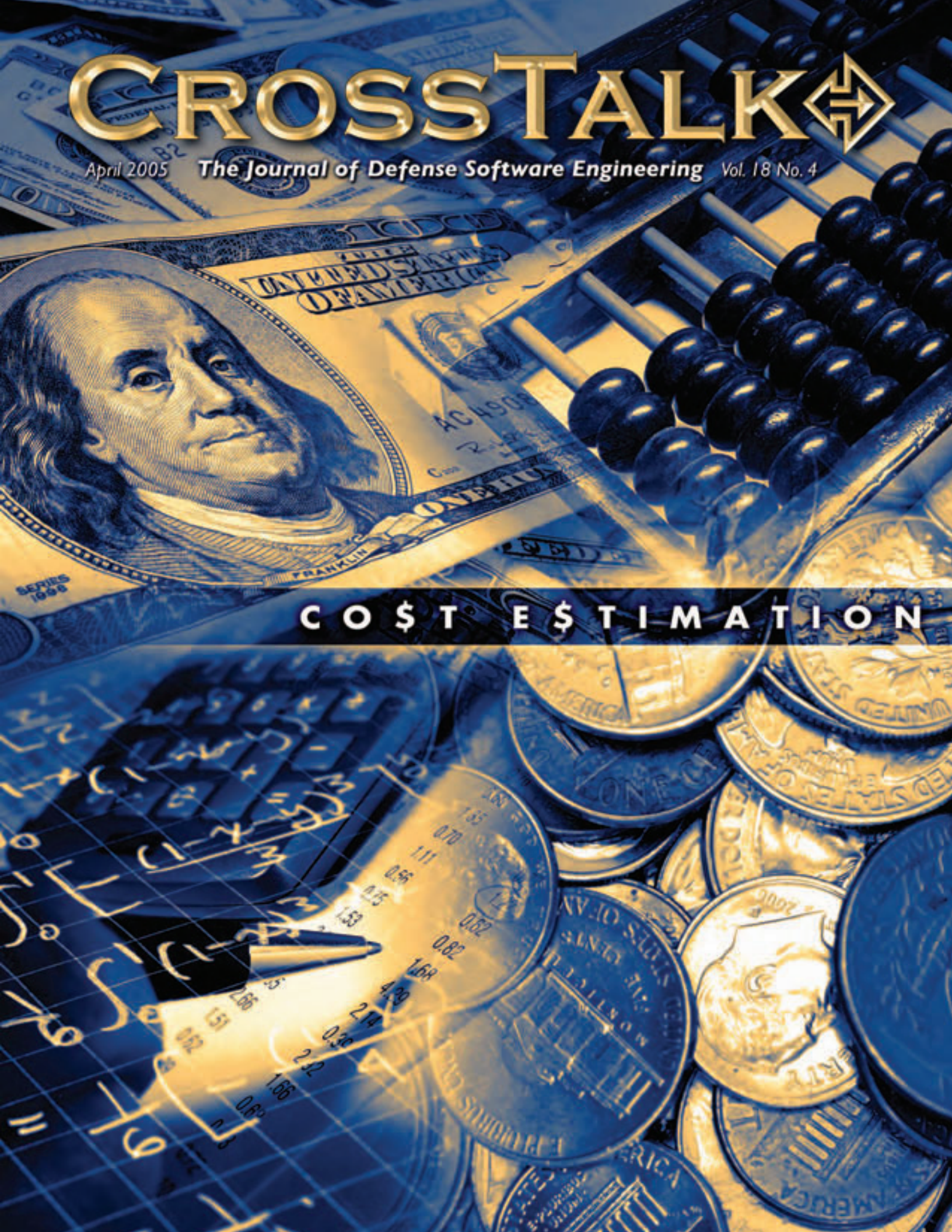
CrossTalk

April 2005 The Journal of Defense Software Engineering Vol. 18 No. 4



COST ESTIMATION

Cost Estimation

4 Estimating and Managing Project Scope for New Development
This article walks the reader through the basic process and considerations needed to determine the project scope for new development, including maintenance builds.
by William Roetzheim

8 Software Cost Estimating Methods for Large Projects
Larger projects have a greater need for commercial software estimating tools, which often outperform human estimates in terms of accuracy, and always in terms of speed and cost effectiveness.
by Capers Jones

13 Creating Requirements-Based Estimates Before Requirements Are Complete
While not recommended, guesstimating auditable and more realistic numbers before requirements have been fully fleshed out is possible using the practices outlined in this article.
by Carol A. Dekkers

16 A Method for Improving Developers' Software Size Estimates
These authors outline a model-based process for mapping requirements to intermediate units to elementary units of work, using the resulting output for estimating.
by Lawrence H. Putnam, Douglas T. Putnam, and Donald M. Beckett

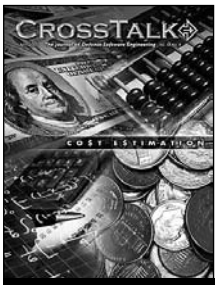
Software Engineering Technology

20 COCOMO Suite Methodology and Evolution
Here is an overview of the models in the COCOMO suite, and how they can be used together to support larger software system estimation needs.
by Dr. Barry Boehm, Ricardo Valerdi, Jo Ann Lane, and A. Winsor Brown

26 Inside SEER-SEM
This article provides insight into the System Evaluation and Estimation of Resources - Software Estimating Model's inner workings and basis of estimation, which are built upon a mix of mathematics and statistics.
by Lee Fischman, Karen McRitchie, and Daniel D. Galorath

Open Forum

29 The Statistically Unreliable Nature of Lines of Code
This author uses a series of Personal Software Process courses to contend that the line-of-code measure is a vague, ambiguous, and unsuitable parameter for sizing software projects.
by Joe Schofield



ON THE COVER
Cover Design by Kent Bingham.

Departments

3 From the Sponsor
From the Publisher

12 Coming Events
Web Sites

34 CrossTalk 101
SSTC 2005 Conference

35 BACKTALK

CROSSTALK

OC-ALC/ MAS Co-Sponsor *Kevin Stamey*
OO-ALC/MAS Co-Sponsor *Randy Hill*
WR-ALC/MAS Co-Sponsor *Tom Christian*
Publisher *Tracy Stauder*
Associate Publisher *Elizabeth Starrett*
Managing Editor *Pamela Palmer*
Associate Editor *Chelene Fortier-Lozancich*
Article Coordinator *Nicole Kentta*
Creative Services Coordinator *Janna Kay Jensen*
Phone (801) 775-5555
Fax (801) 777-8069
E-mail crosstalk.staff@hill.af.mil
CrossTalk Online www.stsc.hill.af.mil/crosstalk

Oklahoma City-Air Logistics Center (OC-ALC), Ogden-Air Logistics Center (OO-ALC), and Warner Robins-Air Logistics Center (WR-ALC) MAS Software Divisions are the official co-sponsors of CrossTalk, The Journal of Defense Software Engineering. The MAS Software Divisions and the Software Technology Support Center (STSC) are working jointly to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

The STSC is the publisher of CrossTalk, providing both editorial oversight and technical review of the journal.

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 25.

OO ALC/MASE
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CrossTalk does not pay for submissions. Articles published in CrossTalk remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD . Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, or the STSC. All product names referenced in this issue are trademarks of their companies.

Coming Events: Please submit conferences, seminars, symposiums, etc. that are of interest to our readers at least 90 days before registration. Mail or e-mail announcements to us.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-7026, or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

From the Sponsor

The Cost Estimation Conundrum

As a former System Program Management Office guy, or *SPO dog*, and now as a software engineering manager, aka *code toad*, there has rarely been a software topic that has given me pause more than cost estimating. Much attention has been focused on software projects that have gone terribly awry with budget and schedule overruns. These are truly unhappy situations that you never want to repeat. So what can you do? One approach is to apply a factor of ignorance, i.e., pad the estimate so that there is sufficient funding. Since estimation is by definition less than precise, this is ethically allowable, and besides, everyone does it. The trouble is everyone does it so managers know it and react to it. We may discount it to bid whatever it takes to win, or we may add to it to build a management reserve that we can blame on software. Either situation is not good.

We owe our customers quality software within cost and on schedule. Few things are harder for customers than trying to find more money for an over-budget project. Going to the well twice means admitting you were wrong originally, something most of us don't like doing. Then there is the often bitter debate about where to find the money. Another project will be taxed unfairly, or your project will be restructured, perhaps even be in danger of being terminated. Regardless of where the money is found, everyone loses. The customer didn't get what was needed when it was needed, and confidence has been lost in the project team's ability to deliver.

The sorry state of affairs that I have outlined above clearly needs to change. That is the emphasis of this issue – how we can do better cost estimating. In my view, we all have a great need for good cost estimation techniques. As a Capability Maturity Model® Integration Level 5 organization, WR-ALC/MAS is committed to constant process improvement. These techniques will aid in that endeavor, and they are beneficial regardless of maturity or capability level. I hope you will take the time not only to read about these techniques, but also apply them.

Thomas F. Christian Jr.

Thomas F. Christian Jr.
Warner Robins Air Logistics Center Co-Sponsor

From the Publisher

Increasing Confidence in Estimates

Cost estimation is certainly one of the biggest challenges software managers face. With large software development or sustainment efforts, developers are increasingly dependent on automated tools to help quantify cost estimates. However, there is not one *silver bullet* modeling tool. As Capers Jones reports this month, a best practice for software cost estimation is to use a combination of estimation modeling tools in conjunction with project management tools.

This month's issue is aimed at increasing confidence in software estimates. Furthermore, industry experts discuss how several cost models are evolving to address technology and process improvements that impact the cost of developing military and commercial software today. William Roetzheim discusses project scope estimation and the difficulties early in the life cycle with indefinite requirements. Capers Jones defines estimation methods for large projects, including non-coding work. Carol A. Dekkers provides helpful guidance when working with incomplete requirements. Barry Boehm et al. present an overview of the Constructive Cost Model tool suite, which can now address commercial off-the-shelf integration, system engineering, and system of systems. Other authors address mapping requirements to units of work, an overview of the SEER-SEM model, and the reliability of lines of code to indicate software size.

I hope we've provided a better understanding of cost estimation, and how estimating models are evolving to keep pace with industry changes. As Lee Fischman et al. states, "The future of software project estimating has just begun."

Tracy L. Stauder

Tracy Stauder
Publisher

Estimating and Managing Project Scope for New Development

William Roetzheim
Cost Xpert Group

Many consider estimating project scope to be the most difficult part of software estimation. Parametric models have been shown to give accurate estimates of cost and duration when given accurate inputs of the project scope, but how do you input scope early in the life cycle when the requirements are still vaguely understood? How can scope be estimated, quantified, and documented in a manner that is understandable to management, end users, and estimating tools? This article focuses on scope estimates for new development, and is applicable for the new development portion of maintenance builds.

The life cycle of software cost estimation is made of many parts, beginning with input parameters at the concept stage and continuing through the function and implementation stages. Many consider estimating project scope to be the most difficult part of software estimation. After all, how do you input scope early in the life cycle when the requirements are still vaguely understood? Consider also that scope must be estimated, quantified, and documented in a manner that is understandable to management, end users, and estimating tools. The focus in this article is scope estimates for new development, including maintenance builds

The Estimating Life Cycle

First, it is important to recognize the limitations of software cost estimating at the macro level. As shown in Figure 1, the typical accuracy of cost estimates varies based on the current software development stage. Early uncertainty in the estimate is largely based on variances in the estimate's input parameters. Later uncertainty in the estimate is based on the variances of the estimating models.

The percentages shown in Figure 1 match this author's personal experience and are roughly comparable with figures found in the Project Management Institute's "A Guide to the Project Management Body of Knowledge" [1]. However, actual numbers will vary widely based on the type of applications involved, the estimators' experience and policies, and other factors.

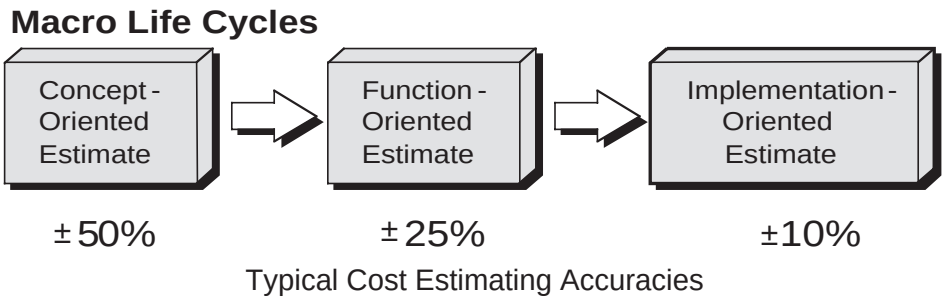
Initially, at the concept stage you may be presented with a vague project definition. Though the requirements may not yet be fully understood, the general purpose of the new software can be recognized. At this point, estimates with an accuracy of ± 50 percent are typical for an experienced estimator using informal techniques (i.e., historical comparisons, group consensus, and so on).

> *"The first step in preparing an estimate is to determine an estimate of the project scope, or volume."*

After the requirements are reasonably well understood, a function-oriented estimate may be prepared. At this point, estimates with an accuracy of ± 25 percent are typical for an experienced estimator using the techniques described above.

Finally, after the detailed design is complete, an implementation-oriented estimate may be prepared. This estimate is typically accurate within ± 10 percent.

Figure 1: *Macro Life Cycle*



Estimating Program Scope

The first step in preparing an estimate is to determine an estimate of the project scope, or volume. Scope is typically estimated using a variety of metrics, as different portions of the application may be compatible with different scope metrics.

One measure of program scope is the number of source lines of code (SLOC). A source line of code is a human-written line of code that is not a blank line or comment. Do not count the same line more than one time even if the code is included multiple times in an application[1]. We typically work with a related number – thousands of SLOC (KSLOC) – when estimating. The Constructive Cost Model (COCOMO) popularized SLOC as an estimating metric. The basic COCOMO model and the new COCOMO II model remain the most well-known estimating approaches because of their prevalence in both academic research settings and as models embedded into estimating tools.

Let us jump ahead and look at how we can convert from the number of KSLOC to an estimate for the project. We will then discuss approaches to estimating KSLOC in more detail.

Begin with the simplest estimate as shown in Table 1. If you are aware of the number of KSLOC your developers must write, and you know the effort required per KSLOC, you then could multiply these two numbers together to arrive at the person months of effort required for your project. This concept is the heart of the estimating models. Table 1 shows some common values that Cost Xpert researchers have found for this linear productivity factor. The COCOMO II value comes from research by Barry Boehm [2] at the University of Southern California. The values for embedded, e-commerce, and Web development come from the Cost Xpert Group's [3] research working with a vari-

ety of organizations, including IBM and Marotz.

Now, let us apply this approach. Suppose we were going to build an e-commerce system consisting of 15,000 LOC. How many person-months of effort would this take using just this equation? The answer is computed as follows:

$$\text{Effort} = \text{Productivity} \times \text{KSLOC} = 3.08 \times 15 = 46 \text{ Person Months}$$

If all of your projects are small, then you can use this basic equation. Researchers have found, however, that productivity does vary with project size. In fact, large projects are significantly less productive than small projects. The probable causes are a combination of increased coordination and communication time plus more rework required due to misunderstandings.

This productivity decrease with increasing project size is factored in by raising the number of KSLOC to a power greater than 1.0. This exponential factor then penalizes large projects for decreased efficiency. Table 2 shows some typical size penalty factors for various project types. Again, the COCOMO II value comes from work by Boehm [2], and values for embedded, e-commerce, and Web development come from work by Cost Xpert Group [3] and its customers. These values have been validated by hundreds of Cost Xpert Group customers/projects, and are updated over time as warranted by the research. Note that because the size factor is an exponential factor rather than linear, it does not change with project size, but changes in impact on the end result with project size.

After we do a size penalty adjustment, how many person-months of effort would our 15,000 lines of code e-commerce system require? The answer is computed as follows:

$$\text{Effort} = \text{Productivity} \times \text{KSLOC}^{\text{Penalty}} = 3.08 \times 15^{1.030} = 3.08 \times 16.27 = 50 \text{ Person Months}$$

All of this is pretty straightforward. The next logical question is, "How do I know my project will end up as 15,000 SLOC?"

There are two approaches to answering this question that I will address: direct estimation and function points (FPs) with backfiring. Using either approach, the fundamental input variables are determined through expert opinion, often with your developers as the experts. The Delphi technique, involving multiple experts iterating toward a consensus decision, is a good way to cross-check the input variables.

Project Type	Linear Productivity Factor (Person Months/KSLOC)
COCOMO II Default	3.13
Embedded Development	3.60
E-Commerce Development	3.08
Web Development	2.51

Table 1: *Estimate Example*

Project Type	Exponential Size Penalty Factor
COCOMO II Default	1.072
Embedded Development	1.111
E-Commerce Development	1.030
Web Development	1.030

Table 2: *Typical Size Penalty Factors*

Normally, the first step in estimating the number of LOC is to break down the project into modules or some other logical grouping. For example, a very high-level breakdown might be front-end processes, middle-tier processes, and database code. Your developers then use their experience building similar systems to estimate the number of LOC required.

We strongly recommend that you obtain three estimates for each input variable: a best-case estimate, a worst-case estimate, and an expected-case estimate. With these three inputs, you can then calculate the mean and standard deviation as follows:

$$\text{Mean} = \frac{(\text{best} + \text{worst} + (4 \times \text{expected}))}{6}$$

$$\text{Standard Deviation} = \frac{(\text{worst} - \text{best})}{6}$$

The standard deviation is a measure of how much deviation can be expected in the final number. For example, if the statistical description of the project is correct and we ignore risk factors not included in the statistical spread, the mean plus three times the standard deviation will ensure that there is a 99 percent probability that your project will come in under your estimate.

For more information, refer to [4].

Estimating Function Points

An alternative to direct SLOC estimating is to start with FPs, then use a process called backfiring to convert from FPs to SLOC. Backfiring is described on page 6, and consists of converting from FPs to SLOC using a language-driven table lookup function. FPs were first utilized by IBM as a measure of program volume. Counting FPs has evolved over time as computer programming techniques and user interface metaphors became more complex; correct function point counting is defined in [5] and is often accomplished using certified FP counting specialists. The original, basic idea is simple and illustrates how it works at a simplified level. True FP counts are more complicated, of course. The program's delivered functionality (and hence, cost) is measured by the number of ways it must interact with the users.

To determine the number of FPs, start by estimating the number of external inputs, external interface files, external outputs, external queries, and logical internal files. External inputs are largely your data-entry screens. External interface files are file-based inputs or outputs. External outputs are your reports and static outputs. External queries are message or external function-based communication into or out of your application. Finally, logical internal files are the number of tables in the database, assuming the database was third normal form or better. As mentioned earlier, these definitions are simplified, but they serve to illustrate the basic concept.

To convert from these raw values into an actual count of FPs, you multiply the raw numbers by a conversion factor from Table 3 on page 6 (again, this approach is a simplification).

Raw Type	Function Point Conversion Factor
External Inputs	4
External Interface files	7
External Outputs	5
External Queries	4
Logical Internal Tables	10

Table 3: *Function Point Conversion Factor*

Language	SLOC per Function Point
C++ Default	53
COBOL Default	107
Delphi 5	18
HTML 4	14
Java 2 Default	46
Visual Basic 6	24
SQL Default	13

Table 4: *Language Equivalencies*

So, if we had a system consisting of 25 data-entry screens, five interface files, 15 reports, 10 external queries, and 20 logical internal tables, how many FPs would we have? The answer is computed as follows:

(25 x 4) + (5 x 7) + (15 x 5) + (10 x 4) + (20 x 10) = 450 FPs

Backfiring

The only remaining step is to use backfiring to convert from FPs to an equivalent number of SLOC. This is done using a table of language equivalencies. Some common values are shown in Table 4 (C++, COBOL, and SQL from work by Capers Jones [6] and other values from research by Cost Xpert Group [3]):

So, to implement the above project (450 FPs) using Java 2 would require approximately the following number of SLOC:

450 x 46 = 20,700 SLOC

and would require the following effort to implement, assuming that this was an e-commerce system:

$$\text{Effort} = \text{Productivity} \times \text{KSLOC}^{\text{Penalty}} = 3.08 \times 20.7^{1.030} = 3.08 \times 22.67 = 69.8 \text{ Person Months}$$

There are also other approaches to calculating equivalent SLOC from a higher-level input value. These other approaches include Internet points, Domino points, and class-method points to name just a few. All of them work in a fashion analogous to FPs as just described.

Table 5: *Project Scope Table*

Function	1
Object	2
Object Library	4
Proof of Concept	5
Evolutionary Prototype	6
Internal Application	8
External Application	9
Shrink-Wrap Application	10
Component of System	11
New System	12
Compound System	13

Heuristic Approaches to Approximating Scope

Estimating Scope by Analogy

This is the software equivalent of market comps in appraising real estate: You look for a project that is as close as possible to your project. Count the physical LOC or function points in that application. Then, use a detailed analysis to adjust things up or down based on differences between the proposed project and this historic project.

You might find that a new proposed project is much more complicated than your database of historic projects. Perhaps you can combine multiple historic projects, each corresponding to a piece of the new project, to arrive at a total estimate of the scope.

Note that it is better to use this approach to estimate scope and then use an estimating tool to estimate effort, rather than using this approach directly to estimate effort. Basically, the scope will be somewhat consistent between similar projects; however, the effort will have a high degree of variability due to things like the people doing the work, the standards and life cycles used, and the development environments. By using historic data to approximate scope and then using project-specific data for all of these other variables, you obtain a much more accurate effort estimate.

Design to Budget and Time-Box Approaches

It is not unusual for a software development budget to be defined before the requirements are defined or perhaps even understood. Market factors might drive the budget. Competitors might define the budget. Resource limitations might determine the budget. In these cases, does estimating make any sense? In fact, estimates are particularly critical under these circumstances.

The approach is to initialize an estimating tool with appropriate values for all of the environmental variables (e.g., development team capabilities, development language, life cycle, standard, etc.). Then, start plugging in values for scope until you obtain a scope estimate that meets the external budget constraint. This then becomes the amount of functionality that you can deliver for the specified budget.

Throughout the development process you must manage expectations to ensure that each step in the process is defining a system that is no larger in size than the budgeted scope. The requirements must be managed along with the design effort, the physical implementation, and so on.

Project Type Taxonomies

It is possible to use project type taxonomies to approximate the FP count of a system to be built (this approach was initially proposed by Capers Jones in "Estimating Software Costs" [6]). The values shown in Table 5 come from Cost Xpert Group research and vary somewhat from the specifics in [6]. It works as follows: In Table 5, select the numerical value that corresponds to your selected project

scope. In other words, are you simply developing a function? Are you writing an object? Are you writing a library of objects? Is this a new shrink-wrap application, or a completely new system (e.g., missile system)?

Using Table 6, select the numerical value that corresponds to your selected project class. In other words, is this development for your personal use? Is it shareware? Is this a civilian-contract programming project? Is this a military project?

Using Table 7, select the numerical value that corresponds to your selected project type. In other words, is this a drag-and-drop fourth generation language development? Is this a batch program? Is it a client-server application? Is it a mathematical application? Is it a new social services program?

Add the three values just obtained together, and then raise this number to the 2.35 power as shown in the following equation:

$$\mathbf{FPs = (Value_{Scope} + Value_{Class} + Value_{Type})^{2.35}}$$

This will give you an approximate value for the number of FPs in the final delivered application. The actual values (e.g., 2.35) are simply mathematical curve-fitting techniques to force this early estimation equation to fit databases of historic projects.

Let us look at a few examples. Suppose we were asked by a commercial communication company to estimate the effort required to create an object that would perform some signal processing functions. This object will be our deliverable. We would use the following values:

Scope = Object
Class = Contract Project - Civilian
Type = Communications

$$\mathbf{FPs = (Value_{Scope} + Value_{Class} + Value_{Type})^{2.35} = (2 + 7 + 11)^{2.35} = 1{,}141\ FPs}$$

Now, suppose we were asked to create a user interface proof-of-concept for a fixed-asset tracking system for internal use only:

Scope = Proof of Concept
Class = Single Location-Internal
Type = No Programming (4GL/Drag and Drop)

$$\mathbf{FPs = (Value_{Scope} + Value_{Class} + Value_{Type})^{2.35} = (5 + 5 + 1)^{2.35} = 280\ FPs}$$

Finally, suppose we need to estimate the effort required to build a new welfare system to be used by a single state with consolidated rules (e.g., there would be no requirement to deliver tailored versions for different counties within the state):

Scope = External Application
Class = State Government-Federally Funded
Type = Social Services

$$\mathbf{FPs = (Value_{Scope} + Value_{Class} + Value_{Type})^{2.35} = (9 + 13 + 15)^{2.35} = 4{,}845\ FPs}$$

Individual Use	1
Shareware	2
Academic/Engineering	3
Single Location - Internal	5
Multilocation - Internal	6
Contract Project - Civilian	7
Contract Project - Local Government	8
Marketed Commercially	9
State Government	11
State Government - Federally Funded	13
Federal Project	14
Military Project	15

Table 6: *Project Class Table*

No Programming (4GL/Drag and Drop)	1
Batch	2
3GL Programming	4
Embedded - Single Board	5
Database Oriented	6
Client-Server	8
Mathematical	9
Systems	10
Communications	11
Process Control	12
Embedded - Multi-Board	13
Embedded - Complete System	14
Social Services	15

Table 7: *Project Type Table*

Conclusion

While determining the scope of new development is never easy, there are techniques that should help you get into the right ballpark. Once there, it becomes a matter of tracking and managing to that scope, either by ensuring that requirements do not grow to exceed the budgeted scope or by using engineering change proposals to obtain additional resources and time when the requirements do exceed the planned scope.◆

References

1. Project Management Institute. A Guide to the Project Management Body of Knowledge (PMBOK Guide). 3rd ed. Newton Square, PA: PMI, 2004.
2. Boehm, Barry, et al. Software Cost Estimation With COCOMO II. Upper Saddle River, N.J.: Prentice-Hall, 2000.
3. Cost Xpert Group. "Data Load 3.3(b)." Internal Report. Rancho San Diego, CA: Cost Xpert Group, 2004.
4. Boehm, Barry. Software Engineering and Project Management. IEEE Press, 1987.
5. ISO [International Organization for Standardization]/International Electrotechnical Commission 20926: 2003 <www.iso.org>.
6. Jones, Capers. Estimating Software Costs. McGraw-Hill, New York, 1998 <www.iso.org/en/prods-services/IOSstore/store.html>.

Note

1. This is a slightly simplified version of the definition from the Software Engineering Institute's Definition Checklist for a Logical Source Statement by R. Park in "Software Size Measurement: A Framework for Counting Source Statements" SEI, Pittsburgh, PA, 1992.

About the Author

William Roetzheim has 25 years experience in the software industry and is the author of 15 software-related books and over 100 technical articles. He is the founder of the Cost Xpert Group, Inc., a Jamul-based organization specializing in software cost-estimation tools, training, processes, and consulting.

Cost Xpert Group
2990 Jamacha RD STE 250
Rancho San Diego, CA 92019
E-mail: william@costxpert.com

Software Cost Estimating Methods for Large Projects©

Capers Jones
Software Productivity Research, LLC

For large projects, automated estimates are more successful than manual estimates in terms of accuracy and usefulness. In descending order, the costs of large projects include defect removal, production of paper documents, coding, project management, and dealing with new requirements that appear during the development cycle. In addition, successful estimates for large projects must be adjusted to match specific development processes, to match the experience of the development team, and to match the results of the programming languages and tool sets that are to be utilized. Simple manual estimates cannot encompass all of the adjustments associated with large projects.

Software has achieved a bad reputation as a troubling technology. Large software projects have tended to have a very high frequency of schedule and cost overruns, quality problems, and outright cancellations. While this bad reputation is often deserved, it is important to note that some large software projects are finished on time, stay within their budgets, and operate successfully when deployed.

The successful software projects differ in many respects from the failures and disasters [1]. One important difference is how the successful projects arrived at their schedule, cost, resource, and quality estimates in the first place. From an analysis of the results of using estimating tools published in "Estimating Software Costs" [2], using automated estimating tools leads to more accurate estimates. Conversely, casual or manual methods of arriving at initial estimates are usually inaccurate and often excessively optimistic.

A comparison of 50 manual estimates with 50 automated estimates for projects in the 5,000-function point range showed interesting results [2]. The manual estimates were created by project managers who used calculators and spreadsheets. The automated estimates were also created by project managers or their staff-estimating assistants using several different commercial-estimating tools. The comparisons were made between the original estimates submitted to clients and corporate executives, and the final accrued results when the applications were deployed.

Only four of the manual estimates were within 10 percent of actual results. Some 17 estimates were optimistic by between 10 percent and 30 percent. A dismaying 29 projects were optimistic by more than 30 percent. That is to say, manual estimates yielded lower costs and shorter schedules than actually occurred, sometimes by significant amounts. (Of course several revised estimates were created along the way. But the comparison was between the initial estimate and the final results.)

In contrast, 22 of the estimates generated by commercial software estimating tools were within 10 percent of actual results. Some 24 were conservative by between 10 percent and 25 percent. Three were conservative by more than 25 percent. Only one automated estimate was optimistic, by about 15 percent.

> *"The conclusion of the comparison was that both manual and automated estimates were equivalent for actual programming, but the automated estimates were better for predicting non-coding activities."*

One of the problems with performing studies such as this is the fact that many large projects with inaccurate estimates are cancelled without completion. Thus, for projects to be included at all, they had to be finished. This criterion eliminated many projects that used both manual and automated estimation.

Interestingly, the manual estimates and the automated estimates were fairly close in terms of predicting coding or programming effort. But the manual estimates were very optimistic when predicting requirements growth, design effort, documentation effort, management effort, testing effort, and repair and rework effort. The conclusion of the comparison was that both manual and automated estimates were equivalent for actual programming, but the automated estimates were better for predicting non-coding activities.

This is an important issue for estimating large software applications. For software projects below about 1,000 function points in size (equivalent to 125,000 C statements), programming is the major cost driver, so estimating accuracy for coding is a key element. But for projects above 10,000 function points in size (equivalent to 1,250,000 C statements) both defect removal and production of paper documents are more expensive than the code itself. Thus, accuracy in estimating these topics is a key factor.

Software cost and schedule estimates should be accurate, of course. But if they do differ from actual results, it is safer to be slightly conservative than it is to be optimistic. One of the major complaints about software projects is their distressing tendency to overrun costs and planned schedules. Unfortunately, both clients and top executives tend to exert considerable pressures on managers and estimating personnel in the direction of optimistic estimates. Therefore, a hidden corollary of successful estimation is that the estimates must be defensible. The best defense is a good collection of historical data from similar projects.

Because software estimation is a complex activity there is a growing industry of companies that market commercial software estimation tools. As of 2005, some of these estimating tools include COCOMO II, CoStar, CostModeler, CostXpert, KnowledgePlan, PRICE S, SEER, SLIM, and SoftCost. Some older automated cost-estimating tools are no longer being actively marketed but are still in use such as CheckPoint, COCOMO, ESTIMACS, REVIC, and SPQR/20. Since these tools are not supported by vendors, usage is in decline.

While these estimating tools were devel-

© 2005 Capers Jones. All Rights Reserved.

oped by different companies and are not identical, they do tend to provide a nucleus of common functions. The major features of commercial software-estimation tools circa 2005 include these attributes:

- Sizing logic for specifications, source code, and test cases.
- Phase-level, activity-level, and task-level estimation.
- Adjustments for specific work periods, holidays, vacations, and overtime.
- Adjustments for local salaries and burden rates.
- Adjustments for various software projects such as military, systems, commercial, etc.
- Support for function point metrics, lines of code (LOC) metrics, or both.
- Support for backfiring or conversion between LOC and function points.
- Support for both new projects and maintenance and enhancement projects.

Some estimating tools also include more advanced functions such as the following:

- Quality and reliability estimation.
- Risk and value analysis.
- Return on investment.
- Sharing of data with project management tools.
- Measurement models for collecting historical data.
- Cost and time-to-complete estimates mixing historical data with projected data.
- Support for software process assessments.
- Statistical analysis of multiple projects and portfolio analysis.
- Currency conversion for dealing with overseas projects.

Estimates for large software projects need to include many more activities than just coding or programming. Table 1 shows typical activity patterns for six different kinds of projects: Web-based applications, management information systems (MIS), outsourced software, commercial software, systems software, and military software projects. In this context, *Web* projects are applications designed to support corporate Web sites. *Outsource* software is similar to MIS, but performed by an outside contractor. *Systems* software is that which controls physical devices such as computers or telecommunication systems. Military software constitutes all projects that are constrained to follow various military standards. Commercial software refers to ordinary packaged software such as word processors, spreadsheets, and the like.

Table 1 is merely illustrative, and the actual numbers of activities performed and the percentages of effort for each activity can vary. For estimating actual projects, the estimating tool would present the most likely set of activities to be performed. Then the project manager or estimating specialist would adjust the set of activities to match the reality of the project. Some estimating tools allow users to add additional activities that are not part of the default set.

Activities Performed	Web	MIS	Outsource	Commercial	System	Military
01 Requirements	5.00%	7.50%	9.00%	4.00%	4.00%	7.00%
02 Prototyping	10.00%	2.00%	2.50%	1.00%	2.00%	2.00%
03 Architecture		0.50%	1.00%	2.00%	1.50%	1.00%
04 Project plans		1.00%	1.50%	1.00%	2.00%	1.00%
05 Initial design		8.00%	7.00%	6.00%	7.00%	6.00%
06 Detail design		7.00%	8.00%	5.00%	6.00%	7.00%
07 Design reviews			0.50%	1.50%	2.50%	1.00%
08 Coding	30.00%	20.00%	16.00%	23.00%	20.00%	16.00%
09 Reuse acquisition	5.00%		2.00%	2.00%	2.00%	2.00%
10 Package purchase		1.00%	1.00%		1.00%	1.00%
11 Code inspections				1.50%	1.50%	1.00%
12 Independent verification and validation						1.00%
13 Configuration management		3.00%	3.00%	1.00%	1.00%	1.50%
14 Formal integration		2.00%	2.00%	1.50%	2.00%	1.50%
15 User documentation	10.00%	7.00%	9.00%	12.00%	10.00%	10.00%
16 Unit testing	30.00%	4.00%	3.50%	2.50%	5.00%	3.00%
17 Function testing		6.00%	5.00%	6.00%	5.00%	5.00%
18 Integration testing		5.00%	5.00%	4.00%	5.00%	5.00%
19 System testing		7.00%	5.00%	7.00%	5.00%	6.00%
20 Field testing				6.00%	1.50%	3.00%
21 Acceptance testing		5.00%	3.00%		1.00%	3.00%
22 Independent testing						1.00%
23 Quality assurance			1.00%	2.00%	2.00%	1.00%
24 Installation/training		2.00%	3.00%		1.00%	1.00%
25 Project management	10.00%	12.00%	12.00%	11.00%	12.00%	13.00%
Total	**100.00%**	**100.00%**	**100.00%**	**100.00%**	**100.00%**	**100.00%**
Activities	**7**	**18**	**21**	**20**	**23**	**25**

Table 1: *Typical Software Development Activities for Six Application Types (Data indicates the percentage of work effort by activity.)*

Cost Drivers for Large Software Systems: Paperwork and Defect Removal

In aggregate, large software projects devote more effort to producing paper documents and to removing bugs or defects than to producing source code. (Some military software projects have been observed to produce about 400 English words for every Ada statement.) Thus, accurate estimation for large software projects must include the effort for producing paper documents, and the effort for finding and fixing bugs or defects, among other things.

The invention of function point metrics [3] has made full sizing logic for paper documents a standard feature of many estimating tools. One of the reasons for the development of function point metrics was to provide a sizing method for paper deliverables. (For additional information on function points, see the Web site of the non-profit International Function Point Users Group <www.ifpug.org>.)

Table 2 (see page 10) illustrates selected documentation size examples drawn from systems, Web projects, MIS, outsource, commercial, systems, and military software domains.

At least one commercial software-estimating tool can even predict the number of English words in the document set, and also the numbers of diagrams that are likely to be present. The document estimate can also change based on paper size such as European A4 paper. Indeed, it is now possible to estimate the sizes of text-based documents in several national languages (i.e. English, French, German, Japanese, etc.) and even to estimate translation costs from one language to another for projects that are deployed internationally.

Software Defect Potentials and Defect Removal Efficiency Levels

A key aspect of software cost estimating is predicting the time and effort that will be needed for design reviews, code inspections, and all forms of testing. To estimate

	Web	MIS	Outsource	Commercial	System	Military	Average
Requirements	0.25	0.50	0.55	0.30	0.45	0.85	0.48
Function Specifications	0.10	0.55	0.55	0.60	0.80	1.75	0.73
Logic Specifications		0.50	0.50	0.55	0.85	1.65	0.81
Test Plans	0.10	0.10	0.15	0.25	0.25	0.55	0.23
User Guides	0.05	0.15	0.20	0.85	0.30	0.50	0.34
Reference		0.20	0.25	0.90	0.34	0.85	0.51
Reports	0.15	0.50	0.60	0.40	0.65	2.00	0.72
Total	**0.65**	**2.50**	**2.80**	**3.85**	**3.64**	**8.15**	**3.60**

Table 2: *Document Pages per Function Point for Six Application Types (Data expressed in terms of pages per function point.)*

defect removal costs and schedules, it is necessary to know about how many defects are likely to be encountered.

The typical sequence is to estimate defect volumes for a project and then to estimate the series of reviews, inspections, and tests that the project utilizes. The defect removal efficiency of each step will be estimated also. The effort and costs for preparation, execution, and defect repairs associated with each removal activity also will be estimated.

Table 3 illustrates the overall distribution of software errors among the same six project types shown in Table 1. In Table 3, bugs or defects are shown from five sources: requirements errors, design errors, coding errors, user documentation errors, and *bad fixes*. A bad fix is a secondary defect accidentally injected in a bug repair. In other words, a bad fix is a failed attempt to repair a prior bug that accidentally contains a new bug. On average, about 7 percent of defect repairs will themselves accidentally inject a new defect, although the range is from less than 1 percent to more than 20 percent bad fix injections.

The data in Table 3, and in the other tables in this report, are based on a total of about 12,000 software projects examined by the author and his colleagues circa 1984-2004. Additional information on the sources of data can be found in [2, 4, 5, 6].

Table 3 presents approximate average values, but the range for each defect category is more than 2-to-1. For example, software projects developed by companies who are at Capability Maturity Model® (CMM®) Level 5 might have less than half of the potential defects shown in Table 3. Similarly, companies with several years of experience with the Six Sigma quality approach will also have lower defect potentials than those shown in Table 3. Several commercial estimating tools make adjustments for such factors.

"One important aspect of estimating is dealing with the rate at which requirements creep and, hence, make projects grow larger during development."

A key factor for accurate estimation involves the removal of defects via reviews, inspections, and testing. The measurement of defect removal is actually fairly straightforward, and many companies now do this. The U.S. average is about 85 percent, but leading companies can average more than 95 percent removal efficiency levels [7].

It is much easier to estimate software projects that use sophisticated quality control and have high levels of defect removal in the 95 percent range. This is because there usually are no disasters occurring late in development when unexpected defects are discovered. Thus, projects performed by companies at the higher CMM levels or by companies with extensive Six Sigma experience often have much greater precision than average.

Table 4 illustrates the variations in typical defect prevention and defect removal methods among the six domains already discussed. Of course, many variations in these patterns can occur. Therefore it is important to adjust the set of activities and their efficiency levels to match the realities of the projects being estimated. However, since defect removal in total has been the most expensive cost element of large software applications for more than 50 years, it is not possible to achieve accurate estimates without being very thorough in estimating defect removal patterns.

The overall efficiency values in Table 4 are calculated as follows: If the starting number of defects is 100, and there are two consecutive test stages that each remove 50 percent of the defects present, then the first test will remove 50 defects and the second test will remove 25 defects. The cumulative efficiency of both tests is 75 percent, because 75 out of a possible 100 defects were eliminated.

Table 4 oversimplifies the situation, since defect removal activities have varying efficiencies for requirements, design, code, documentation, and bad fix defect categories. Also, bad fixes during testing will be injected back into the set of undetected defects.

The low efficiency of most forms of defect removal explains why a lengthy series of defect removal activities is needed. This, in turn, explains why estimating defect removal is critical for overall accuracy of software cost estimation for large systems. Below 1,000 function points, the series of defect removal operations may be as few as three. Above 10,000 function points, the series may include more than a dozen kinds of review, inspection, and test activity defect removal operations.

Table 3: *Average Defect Potentials for Six Application Types (Data expressed in terms of defects per function point.)*

	Web	MIS	Outsource	Commercial	System	Military	Average
Requirements	1.00	1.00	1.10	1.25	1.30	1.70	1.23
Design	1.00	1.25	1.20	1.30	1.50	1.75	1.33
Code	1.25	1.75	1.70	1.75	1.80	1.75	1.67
Documents	0.30	0.60	0.50	0.70	0.70	1.20	0.67
Bad Fix	0.45	0.40	0.30	0.50	0.70	0.60	0.49
Total	**4.00**	**5.00**	**4.80**	**5.50**	**6.00**	**7.00**	**5.38**

Requirements Changes and Software Estimation

One important aspect of estimating is dealing with the rate at which requirements creep and, hence, make projects grow larger during development. Fortunately, function point metrics allow direct measurement of the rate at which this

® Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

phenomenon occurs since both the original requirements and changed requirements will have function point counts.

Changing requirements can occur at any time, but the data in Table 5 runs from the end of the requirements phase to the beginning of the coding phase. This time period usually reflects about half of the total development schedule. Table 5 shows the approximate monthly rate of creeping requirements for six kinds of software, and the total anticipated volume of change.

For estimates made early in the life cycle, several estimating tools can predict the probable growth in unplanned functions over the remainder of the development cycle. This knowledge can then be used to refine the estimate and to adjust the final costs in response.

Of course, the best response to an estimate with a significant volume of projected requirements change is to improve the requirements gathering and analysis methods. Projects that use prototypes, joint application design (JAD), requirements inspections, and other sophisticated requirements methods can reduce later changes to a small fraction of the values shown in Table 5. Indeed, the initial estimates made for projects using JAD will predict reduced volumes of changing requirements.

Adjustment Factors for Software Estimates

When being used for real software projects, the basic default assumptions of estimating tools must be adjusted to match the reality of the project being estimated. These adjustment factors are a critical portion of using software estimating tools. Some of the available adjustment factors include the following:

- Staff experience with similar projects.
- Client experience with similar projects.
- Type of software to be produced.
- Size of software project.
- Size of deliverable items (documents, test cases, etc.).
- Requirements methods used.
- Review and inspection methods used.
- Design methods used.
- Programming languages used.
- Reusable materials available.
- Testing methods used.
- Paid overtime.
- Unpaid overtime.

Automated estimating tools provide users with abilities to tune the estimating parameters to match local conditions. Indeed, without such tuning the accuracy of automated estimation is significantly reduced. Knowledge of how to adjust estimating tools in response to various factors is the true heart of software estimation. This kind of knowledge is best determined by accurate measurements and multiple regression of analysis of real software projects.

	Web	MIS	Outsource	Commercial	System	Military
Prevention Activities						
Prototypes	20.00%	20.00%	20.00%	20.00%	20.00%	20.00%
Clean rooms					20.00%	20.00%
JAD sessions		30.00%	30.00%			
QFD sessions					25.00%	
Subtotal	*20.00%*	*44.00%*	*44.00%*	*20.00%*	*52.00%*	*36.00%*
Pretest Removal						
Desk checking	15.00%	15.00%	15.00%	15.00%	15.00%	15.00%
Requirements review			30.00%	25.00%	20.00%	20.00%
Design review			40.00%	45.00%	45.00%	30.00%
Document review				20.00%	20.00%	20.00%
Code inspections				50.00%	60.00%	40.00%
Independent verification and validation						20.00%
Correctness proofs						10.00%
Usability labs				25.00%		
Subtotal	*15.00%*	*15.00%*	*64.30%*	*89.48%*	*88.03%*	*83.55%*
Testing Activities						
Unit test	30.00%	25.00%	25.00%	25.00%	25.00%	25.00%
New function test		30.00%	30.00%	30.00%	30.00%	30.00%
Regression test			20.00%	20.00%	20.00%	20.00%
Integration test		30.00%	30.00%	30.00%	30.00%	30.00%
Performance test				15.00%	15.00%	20.00%
System test		35.00%	35.00%	35.00%	40.00%	35.00%
Independent test						15.00%
Field test				50.00%	35.00%	30.00%
Acceptance test			25.00%		25.00%	30.00%
Subtotal	*30.00%*	*76.11%*	*80.89%*	*91.88%*	*92.69%*	*93.63%*
Overall Efficiency	**52.40%**	**88.63%**	**96.18%**	**99.32%**	**99.58%**	**99.33%**
Number of Activities	**3**	**7**	**11**	**14**	**16**	**18**

Table 4: *Patterns of Defect Prevention and Removal Activities*

Summary and Conclusions

Software estimating is simple in concept, but difficult and complex in reality. The larger the project, the more factors there are that must be evaluated. The difficulty and complexity required for successful estimates of large software projects exceeds the capabilities of most software project managers to produce effective manual estimates. In particular, successful estimation of large projects needs to encompass non-coding work.

The commercial software estimating tools are far from perfect and they can be wrong, too. But automated estimates often outperform human estimates in terms of accuracy, and always in terms of speed and cost effectiveness. However, no method of estimation is totally error-free. The current *best practice* for software cost estimation is to use a combination of software cost estimating tools coupled with software project management tools, under the careful guid-

Table 5: *Monthly Rate of Changing Requirements for Six Application Types (From end of requirements to start of coding phases)*

	Web	MIS	Outsource	Commercial	System	Military	Average
Monthly Rate	4.00%	2.50%	1.50%	3.50%	2.00%	2.00%	2.58%
Months	6.00	12.00	14.00	10.00	18.00	24.00	14.00
TOTAL	**24.00%**	**30.00%**	**21.00%**	**35.00%**	**36.00%**	**48.00%**	**32.33%**

Coming Events

April 18-21
2004 Systems and Software Technology Conference
Salt Lake City, UT
www.stc-online.org

May 2-6
Practical Software Quality and Testing (PSQT) 2005
Las Vegas, NV
www.qualityconferences.com

May 14-15
ACM Symposium on Software Visualization
St. Louis, MO
www.softvis.org/softvis05

May 15-21
27th International Conference on Software Engineering (ICSE)
St. Louis, MO
www.icse-conferences.org/2005

May 16-17
Military Embedded Electronics and Computing Conference
Long Beach, CA
www.meecc.com

May 16-20
STAREAST 2005 International Conference on Software Testing Analysis and Review
Orlando, FL
www.sqe.com/stareast

May 23-26
2005 Combat Identification Systems Conference
Portsmouth, VA
www.usasymposium.com/combatid

June 12-15
ACM Sigplan 2005 Programming Language Design and Implementation
Chicago, IL
http://research.ihost.com/pldi2005

ance of experienced software project managers and estimating specialists.◆

References

1. Jones, Capers. "Software Project Management Practices: Failure Versus Success." CrossTalk Oct. 2004 <www.stsc.hill.af.mil/crosstalk/2004/10/0410Jones.html>.
2. Jones, Capers. Estimating Software Costs. New York: McGraw Hill, 1998.
3. Albrecht, Allan. AD/M Productivity Measurement and Estimate Validation. Purchase, NY: IBM Corporation, May 1984.
4. Jones, Capers. Applied Software Measurement. 2nd ed. New York: McGraw Hill, 1996.
5. Jones, Capers. Software Assessments, Benchmarks, and Best Practices. Boston, MA: Addison Wesley Longman, 2000.
6. Kan, Stephen H. Metrics and Models in Software Quality Engineering. 2nd ed. Boston, MA: Addison Wesley Longman, 2003.
7. Jones, Capers. Software Quality – Analysis and Guidelines for Success. Boston, MA: International Thomson Computer Press, 1997.

About the Author

Capers Jones is founder and chief scientist of Software Productivity Research (SPR) LLC. He has almost 40 years of experience in software cost estimating. Jones designed IBM's first automated estimation tool in 1975, and is also one of the designers of three commercial software estimation tools: SPQR/20, Checkpoint, and KnowledgePlan. These software estimation tools pioneered the use of function point metrics for sizing and estimating. They also pioneered sizing of paper documents, and the estimation of quality and defect levels. To build these tools, SPR has collected quantified data from more than 600 companies.

Software Productivity Research, LLC
Phone: (877) 570-5459
E-mail: cjones@spr.com

Web Sites

International Function Point Users Group

www.ifpug.org

The International Function Point Users Group (IFPUG) is a non-profit organization committed to increasing the effectiveness of its members' IT environments through the application of function point analysis (FPA) and other software measurement techniques. IFPUG endorses FPA as its standard methodology for software sizing and maintains the Function Point Counting Practices Manual, the recognized industry standard for FPA. IFPUG serves more than 1,200 members in more than 30 countries.

COCOMO

http://sunset.usc.edu/research/COCOMOII

The Constructive Cost Model (COCOMO) Suite is a collection of six COCOMO-related estimation models in various stages of development. These models attempt to estimate impacts on software system cost, development schedule, and even return on technology investment associated with a variety of software development approaches and processes.

Space Systems Cost Analysis Group

http://sscag.saic.com

The Space Systems Cost Analysis Group (SSCAG) is a non-profit, international association of aerospace organizations representing industry and government. SSCAG members are involved in space systems cost analysis, including hardware or software related to launch systems, spacecraft, payloads, experiments, and space-related ground systems.

International Society of Parametric Analysts

www.ispa-cost.org

The International Society of Parametric Analysts (ISPA) is a non-profit educational society devoted to the promotion of parametrics, risk analysis, econometrics, design to cost, technology forecasting, and management. Many ISPA members currently participate in the Parametric Estimating Initiative, enabling them to rely on parametrics as the primary basis of estimate. ISPA chapters provide technical workshops, training, and networking opportunities.

Creating Requirements-Based Estimates Before Requirements Are Complete

Carol A. Dekkers
Quality Plus Technologies, Inc.

Despite advances in tools and techniques, it is interesting to note that on-time and on-budget projects account for a mere one-third of projects today. While overly optimistic estimates are part of the problem, missing and incomplete requirements, and poor estimating methods share the blame. Accurate estimating is further challenged when customers demand estimates before requirements development begins.

Project estimating is formidable from the start – especially during/before the requirements discovery process. Poor requirements lead to poor estimates and poor schedules. Subsequently, changes are difficult to assess when the requirements are poor. Sometimes a service request ends up deferred to the next release due to confusion about requirements – and the next release fares no better. This leads to classic project failure – over budget and behind schedule – similar to the two-thirds of projects cited in the 2003 Standish Group's Chaos report. [1]

Estimators can start the process by determining how big and how complex is the user problem, how hard it will be to build, and how much confidence is needed in the estimate. Most often, however, we do not estimate this way. We start with a seemingly arbitrary end date and then count backwards to get the schedule, cost, and resources that can fit into the timeframe. Called *date-driven estimating* by author Steve McConnell, it is the most commonly used method. [2]

To complicate matters, date-driven estimates are usually task-based and rely on the *experience* and mental models of team members or cost estimators. Problems emerge on new projects with new technology or new subject matter where there is no prior experience from which to draw. An estimator is forced to seek other data on which to base an estimate.

Sophisticated parametric-based estimating models such as COCOMO II, SLIM, and SEER/SEM serve to provide the missing data with databases or proven industry equations. In most cases, however, some form of project size is a required input variable, along with other variables covering functional, quality, design, and technical drivers. Because any estimate is only as accurate as its least accurate input variable, we should not be surprised when projects exceed estimates for cost, schedule and duration. The Standish Group report [1] proclaimed a mere 33 percent of projects a success; however, this is double the results a mere decade ago.

As one of the first authors to recognize that software engineering differs from traditional engineering, David Card stated, "Engineering projects usually can wait until after design to provide an estimate, while software engineering requires an estimate before design" [3].

In the author's experience, software projects can be even worse – some projects need estimates before requirements! If we are to increase information technology (IT) credibility, we need to figure out ways to create auditable and reliable project estimates from initial project realization all the way through to project completion. One of the best ways to do this is to augment our current estimating method(s) with at least one requirements-based estimate. This additional approach serves to validate or invalidate the other estimate(s) and ensures that at least one method considered the size of the problem as an important project estimating variable.

Requirements Demystified

Given that project requirements are the source of 60 percent to 99 percent of defects delivered into production [4], and that project size based on requirements is a key input driver for project estimates [5], it makes sense to examine what can be done to clarify and further exploit the discovery of complete requirements early in the project.

The requirements discovery and articulation process should strive to maximize the known requirements while managing to minimize the unknowns. To clarify project requirements, divide them into three types: functional, non-functional, and technical requirements, as outlined in the following sections.

Functional Requirements

This type of requirements represents the unit work processes performed or supported by the software, (e.g., software for an altimeter records the ambient temperature). These requirements are part of the users'[1]/customers' responsibility to define, even though they may abdicate the initial specifications to the development team. Functional requirements can be thought of similar to a software floor plan – they are independent of any design constraints or technical implementation. Functional requirements can be documented with use cases and sized using functional size measurement (function points).

Once the functional requirements are sized, and other project requirements are known (see non-functional and technical requirements), cost estimates can be prepared using a Project Cost Ratio for comparable completed projects (see Table 1).

Non-Functional Requirements

This type of requirements represents how the software must perform once it is built. Also referred to as quality requirements, these requirements address the *ilities*: (suitability, accuracy, interoperability, compli-

Table 1: *Project Requirements Size-Based Estimating Equations*

Metric	Units	Equation
Project Cost Ratio (completed projects)	$/Function Point (FP)	Project Cost Rate = (Total Hours x Hourly Cost) + Other Costs / Project Functional Size
Annual Support Cost Ratio	Actual Support Costs per 1,000 FP (or Full Time Resources/Application)	Support Cost Ratio = (Yearly Support Hours x Hourly Cost) + Other Costs / Application Functional Size
Repair Cost Ratio	$/FP (or per fix)	Repair Cost Ratio = (Repair Hours x Hourly Cost) / Functional Size of Repair

ance, security, reliability, efficiency, maintainability, portability, and quality in use) as described by ISO [International Organization for Standardization] standards in [7] and performance criteria.

More often, non-functional requirements are discussed only at a high level and are often found scattered throughout various requirements documents. Using a construction analogy, the non-functional requirements are like the *contracted specifications* for software and outline the necessity for data accuracy (e.g., trajectory systems), response time (e.g., service level agreements), security (e.g., encryption), performance (e.g., 24x7 operation with replicated databases to prevent data loss), etc.

Technical (Build) Requirements

These project requirements are defined by how the software will be *built* to satisfy the functional and non-functional requirements. Technical requirements include the physical implementation characteristics of the project and include, for example, programming language, Computer-Aided Software Engineering (CASE) or other tools, methods, work-breakdown structure, type of project, etc. In practice, it is the technical requirements that document the design, and with the functional and non-functional requirements give rise to project specifics like Gantt charts, development methodology, reuse, etc. Technical requirements are to software as plumbing is to building construction.

All three types of project requirements are necessary to do a realistic project estimate. Functional size measurement strictly pertains only to the size of the software's functional user requirements.

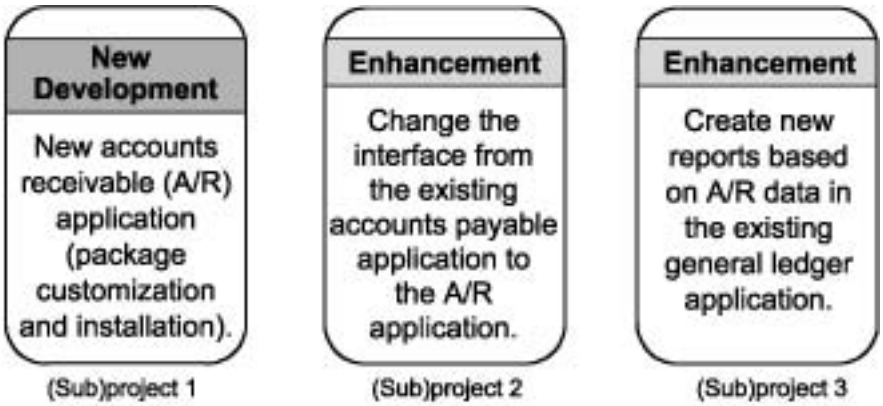
Modern software development approaches such as use cases and agile development attempt to categorize and keep these three types of requirements distinct and separate. Unfortunately in a manner similar to the contractor who only has a hammer and everything looks like a nail, some software developers cannot overcome the need to insert technical requirements into modern method deliverables such as use cases and agile user stories.

Estimating Challenges

The more information you know before making an estimate, the better the estimate should be. However, estimating faces challenges even with skilled estimators and high-quality teams. A few challenges include these: accuracy of input values (size, complexity, technical requirements, etc); availability of input variables; applicability of historical databases; completeness of the requirements (including functional, non-functional, and technical); tasks to be included; and risk factors. In spite of the challenges, cost estimators do produce estimates of duration, cost, and effort, which are turned into project schedules. Estimates made early in the development life cycle face large variations because of uncertainty. Estimates based on guessed input values are unreliable, yet many managers treat them as predictive project forecasts. We can alleviate this problem with a few guidelines: *Frame the guesstimate (an estimated guess) as preliminary*. When providing a guesstimate, frame it as a range of values (e.g., based on assumptions, the project could cost $250,000 to $600,000). Giving a range instead of an *exact* answer provides greater traceability.

Overly optimistic estimates create project failures because dates pass and slip, functionality gets reduced, project budgets get surpassed, and quality suffers (i.e., testing time is cut out). Remember that an estimate is only as good as its least reliable input variable; garbage in equals garbage out. While it is the American way for faster, better, and cheaper solutions, sometimes they are so compelling that management will attempt the impossible through the overly optimistic estimate. The result is that the project will only be done right the second time around [4].

Figure 1: *Sample Project Components*



Estimating During or Before Requirements

When asked to perform an overall project estimate using a requirements-based estimating method, the first step is to decide how many separate (sub)projects are included within the scope of the overall business project if more than one software application is involved. If there is only one software application involved, this step can be skipped. If there is more than one application to be enhanced or developed, each usually has its own set of requirements and will need its own (sub)project estimate[2]. (Usually, each application that undergoes new development or enhancement will be classified and estimated as its own (sub)project, and the overall project effort, cost, and duration can be calculated as combined values. Consider a single overall project with several subprojects: (a) new development project, and (b) two enhancement projects (see Figure 1). Each one would be estimated separately, and the results added together. Additionally, the entire project might also require an estimate for the integration testing of the component subproject pieces. The overall project estimate for cost and effort would be the sum of the subproject estimates, while the duration would depend on task dependencies between and within subprojects[3].

The second step is to identify and estimate the size or impact of the three types of project requirements for each of the subprojects. Consider the fictional subproject 1.

Functional Requirements

The requirements for *what* the software must do might not be defined in enough detail to do functional sizing, but could be approximated [5]. If even one functional component (such as number of entities) is known, an approximation can be done. Several approximation methods are outlined in [6]. Documenting the assumptions about the entities helps to substantiate the estimate[4]. If there is enough data, the functional size approximation can be more accurate and use more accurate techniques. For the two subprojects, each would be assessed based on an approximation of how many function points would be added (new functions as in subproject 3), modified (changed or renovated functions), or removed. The functional size of the subproject is the sum of new plus modified plus removed functions.

Non-Functional Requirements

Assessment of the *ilities* is based on a

comparison to similar projects or a value adjustment factor (which is part of a sizing method such as IFPUG). If the non-functional requirements are unknown, it is best to overestimate their impact as usually they turn out to be more complex than anticipated (e.g., security requirements). Even if estimators and software developers intuitively know that estimates are too low, customers and user managers have an insatiable optimism that maybe, *just this once* it might come true. Time and time again, overly optimistic estimates become self-fulfilling prophecies as dates slip, functionality is reduced, and project budgets are surpassed.

Barry Boehm remarked on the impact of non-functional requirements: "A tiny change in NFRs [non-functional requirements] can cause a huge change in the cost." Boehm cited the tripling of a $10 million project to $30 million when the response time (of a NFR) went from four seconds to one. [8]. It is important to document assumptions for NFRs, especially if project complexity is likely to increase.

Technical Requirements

IT project teams often use a standard suite of development tools and technologies. The technical requirements are usually the least risk prone of the three requirement types – particularly technologies and subject matter are standard. For major changes in technology, further care must be taken to assess this requirements area.

Results should be documented along with the method used, the date, and source documents used for the estimate so that guesstimates and estimates become more traceable and auditable.

Need an estimate for a project that has few or no known input variables? Are there options for an estimator? He or she could attempt these tactics: (a) refuse to do an estimate, (b) delay the estimate repeatedly until requirements are at least partially done, (c) provide a wild guess (which is common), (d) try to find similar completed projects within your own environment and use their actual values, (e) cite *professional* ethics and hide out, or (f) (this is the preferred method) document assumptions and use them together with the estimate (guesstimate) to substantiate the estimation results.

What Can You Do to Improve Project Estimates?

Project estimating can be more auditable and more realistic by applying some of the aforementioned practices. Document as many of your assumptions about the project as you can; revise them and the estimate according to the same/updated assumptions later. Separate, document, and assess (approximation or count) the project into subprojects according to application; address each set of requirements clearly; and objectively split them into the three types: functional, non-functional, and technical. Use an established requirements-based estimating tool or benchmarking database such as COCOMO II or the International Software Benchmarking Standards Group with proven track records for your environment. Label results as preliminary. Teach customers about the estimating process. Educate them that an estimate too early in the life cycle cannot remain fixed throughout the project, nor can it be accurate. And finally, combine the subproject estimates into a single overall project estimate. Present the guesstimate as a range (when information is premature or missing) with a level of accuracy commensurate with what is known about the project at the time (e.g., rounded to the closest $100,000).◆

References

1. The Standish Group. "Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved by 50 Percent." Press Release, 25 Mar. 2003 The Standish Group, <www.standishgroup.com/press/article.php?id=2>.
2. McConnell, Steve. "After the Gold Rush." 2004 Systems and Software Technology Conference, Salt Lake City, UT, 19-22 Apr. 2004.
3. Card, David N. The Role of Measurement in Software Engineering. July 1998.
4. U.S. Army. Insight. Summer, 2003.
5. Hill, Peter R., Ed. Practical Project Estimation. 2nd ed. International Software Benchmarking Standards Group, 2005 <www. isbsg.org>.
6. International Organization for Standardization/International Electrotechnical Commission. ISO/IEC 14143-1:1998 Information Technology – Software Measurement – Functional Size Measurement – Part 1: Definition of Concepts. ISO/IEC <www.jtcl-sc7.org>.
7. ISO/IEC. ISO/IEC 9126 Series of Standards for Measuring Software Quality. ISO/IEC <www.jtcl-sc7.org>.
8. Robertson, Suzanne and James. Preface. Requirements-Led Project Management – Discovering David's Slingshot. By Barry Boehm. Pearson Education, 2005.

Notes

1. *Users* refers to any person, thing, other application, other software, hardware, etc., outside the boundary of the software that has the requirement to send or receive data from the software [6].
2. Even if requirements are collectively listed in a single document, specific requirements will pertain to a specific software application. It is important to divide the requirements among various applications within the overall project to facilitate subproject estimates.
3. The overall duration may not be the summation of the subproject durations; some tasks may proceed concurrently while others may have precedence in other subprojects before they can commence.
4. The one file model or rule of 31 is an approximation technique whereby each identified entity is assumed to have add, change, delete, query, output, and storage function. Using IFPUG FP average values, the total is 31 FP for each entity. For three entities, this equates to 93 FP – or roughly in the range of 100 FP.

About the Author

Carol A. Dekkers is president of Quality Plus Technologies, Inc., a management consulting firm that specializes in helping companies improve their software and systems success. She is a past chair and founder of the American Testing Board, a former president of the International Function Point Users Group, and is active in the Project Management Institute, the American Society for Quality, and the International Organization for Standardization. She is a Certified Management Consultant, a Certified Function Point Specialist, a professional engineer (Canada), an Information Systems Professional, and an International Software Testing Qualifications Testing Board Certified Tester – Foundation Level.

Quality Plus Technologies, Inc.
8430 Egret LN
Seminole, FL 33776
Phone: (727) 393-6048
Fax: (727) 393-8732
E-mail: dekkers@qualityplustech.com

A Method for Improving Developers' Software Size Estimates

Lawrence H. Putnam, Douglas T. Putnam, and Donald M. Beckett
Quantitative Software Management, Inc.

Traditional software estimating is effort-based and follows a bottom-up approach. This approach does not show the impact of different team sizes or the impact of schedule, cost, and quality constraints. The authors propose a method that decomposes programming artifacts into elementary units of work that form the size used for model-based estimating. The process is simple to implement, flexible, can be tuned with actual project performance data, and fosters developer buy-in by involving them in the estimating process.

It is common to hear this question during project development: "How large is this project?"

One type of answer might be, "Oh, let me see. I believe that this is about a 500 effort-hour project."

In the world of software development, size means different things to different groups of people. Those who specify functional requirements – and perhaps pay for the project, too – may conceptualize size in financial terms. Since project effort is a key component of cost, sizing in effort-units allows them to place the project in a cost-and-resource framework.

Software developers, while keenly aware of the effort required to complete their tasks, are more likely to describe size in terms of the things they have to produce to implement the requirements. Their sizing units are screens to be developed and modified, reports, database tables, Web pages, scripts, object classes, and a host of others.

To the software estimator, size quantifies what a project delivers or proposes to deliver. Concrete measures such as source lines of code or more abstract ones such as function points are the estimator's size units, and are indeed the ones needed to use a commercial estimating tool.

What is certain is that in software development, the word size may mean effort, programming artifacts, or elementary units of work depending on who is using the term. This is a potential source of confusion and miscommunication.

This article outlines a process for mapping requirements to intermediate units to elementary units of work, as shown in Figure 1, and uses the resulting output for estimating. The process is flexible and uses historical data to tune its algorithms.

Traditional Sizing and Estimating

Historically, software estimating has followed a pattern similar to the following:

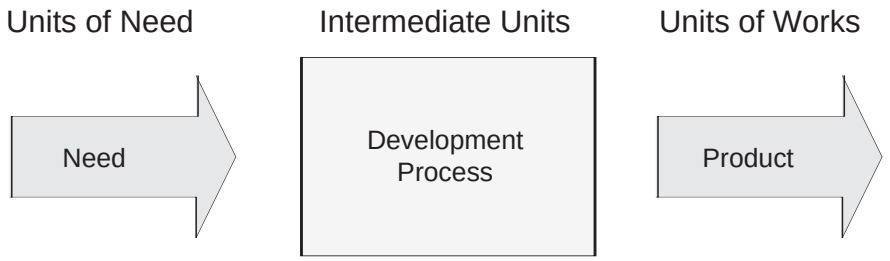
- Requirements are broken down into software elements.
- Effort-hours for the tasks to create the software elements are estimated.
- The effort-hours are summed and a management reserve (fudge factor) is added to give an effort-estimate.
- Resources are leveled and a critical path is determined that allow project staff and duration to be estimated.

Unfortunately, this bottom-up approach is fraught with problems:

- It underestimates the overhead required and the non-software tasks associated with a larger project, often dramatically.
- Bottom-up estimating cannot be done effectively early in the project life cycle when bid/no bid or go/no go decisions are made and money, time, and staff are allocated to the effort. There is simply insufficient detail to determine all of the software elements, much less the project elements.
- It ignores the impact on schedule and effort of different sized teams. Schedule is simply effort divided by staff.
- It does not account for the non-linear impacts of time, cost, and quality constraints.
- It is not suitable for rapid, cost-effective, *what-if* analysis.

A critical element is missing from this approach and that element is project size.

Figure 1: *Development Process*



An Alternative Approach to Sizing/Estimating

Parametric or model-based estimating takes the following different approach:

- It determines the size of the software elements breaking them down into common low-level software implementation units (IUs). (This will be discussed in the following section.)
- It creates a model-based *first cut estimate* using a productivity assumption (preferably historically based), the project size, and the critical constraints.
- It performs *what-if* modeling until an agreed-upon estimate has been created.
- It creates the detailed plans for the project.

Figure 2 illustrates this approach. Key to the success of this methodology is an accurate size and a productivity assumption that is consistent with the organization's capabilities.

Translating Requirements Into IUs

Customers have needs. These take the form of requirements that software must fulfill. Developers translate these requirements into intermediate units that they must create or modify to implement the requirements. These can be screens, programs, reports, tables, object classes, interfaces, etc. The list is fluid. Estimators must decompose the intermediate units into IUs to determine a size for estimating.

Conceptually, an IU is the lowest level of programming construct that a software developer performs. It will vary in form depending on what is being developed. It could be setting a property on a Web form, indicating the data type of a field on a database table, or writing a line of procedural code. In each case it is the most elementary activity that the developer performs. Intermediate units are the tangible results of several or many IUs.

Two traditional size measures for estimating are source lines of code and function points. Both of these can work in some cases; however, each has limitations. The lines of code that a project generates are strongly influenced by the software languages used, individual coding style, and organizational standards. They are a measure of output that can be difficult to estimate.

Function points can be estimated from requirements and design documents but require training in the function point methodology and actual counting experience that many organizations lack. Function point counting is also a manual process that requires an investment of time and effort to perform. Although there are software tools that can capture the results of a function point count, there are none certified by the International Function Point Users Group as being capable of conducting the count.

An Alternative Sizing Approach

Here is a process for obtaining a size estimate that is conceptually simple, easy to implement, and encourages developer *buy-in*:

- Hold a facilitated session with the developers. Have them identify all of the intermediate units that they have to create. Determine what they physically have to do to create them. Ask if there are other things that they have to create on other projects. The purpose here is to establish a comprehensive list of the artifacts the developers may create. Good interviewing skills are the key to success here. Ask follow-up questions and keep asking if there is anything more. Developers may take some time to warm to this approach, but asking people to talk about themselves and what they do is a time-tested method of keeping a conversation going!
- For each item, have them define in quantifiable terms what makes that item simple, average, or complex. For instance, a simple screen might only have retrieval capability, while an average screen would also allow data entry. A complex screen would have update and delete capabilities, as well. Record the intermediate units in both effort-hours and IUs, which may be a ratio of effort at this stage. It is especially important to have several developers involved in this. Individual productivity can vary significantly between individuals, which influences their perspectives. Also, having the group of developers determine effort ranges will help balance overly optimistic and pessimistic estimates and help create buy-in.
- Construct a sizing worksheet that captures the results of the session. Figure 3 is a simple illustration of this concept.
- For a medium-size project with a small team, this process will normally take between four and six hours with between four and eight developers; this is where you get buy-in from the developers. Very large projects may well require additional time, but the method remains the same.

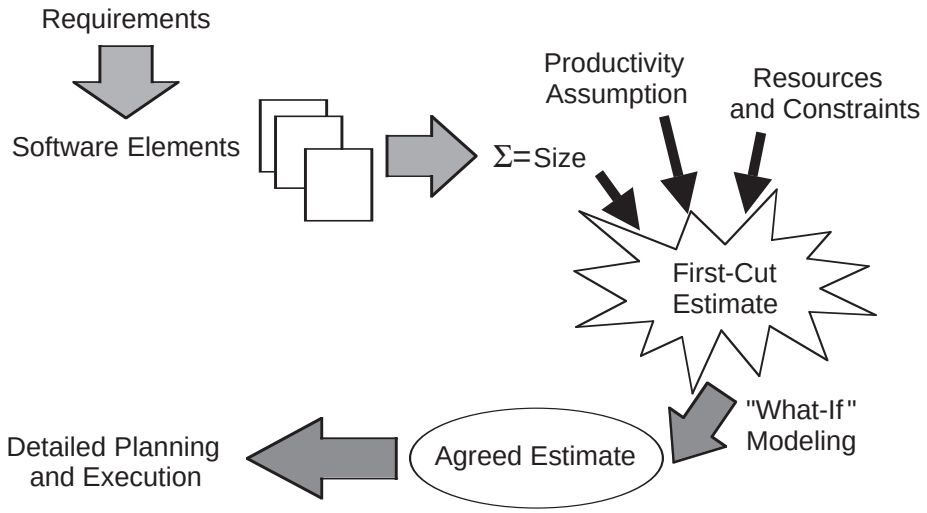


Figure 2: *Alternative Sizing and Estimating Approach*

Figure 3 is an example of a sizing spreadsheet. Using the intermediate units specified by the developers during the interview for this particular project type, data has been captured for a hypothetical project. The intermediate units that the developers have identified as being in their environment are in the first column. The second column contains the developers' estimate of the average hours required to create each intermediate unit. The IUs in

Figure 3: *Sizing Spreadsheet Example*

Intermediate Units	Effort Hours	IUs	Count	Total IUs	Total Effort
Forms - Simple	8	70		0	0
Forms - Average	15	170	8	1,360	120
Forms - Complex	30	400		0	0
New Report - Simple	13	140		0	0
New Report - Average	32	300	8	2,400	256
New Report - Complex	42	440		0	0
Changed Report - Simple	10	90		0	0
Changed Report - Average	24	250	4	1,000	96
Changed Report - Complex	31	320		0	0
Table Changes - Simple	5	60		0	0
Table Changes - Average	13	140	10	1,400	130
Table Changes - Complex	20	220		0	0
JCL Changes - Simple	1	12		0	0
JCL Changes - Average	4	50		0	0
JCL Changes - Complex	6	70		0	0
SQL Procedures - Simple	1	14		0	0
SQL Procedures - Average	10	140		0	0
SQL Procedures - Complex	20	225		0	0
Total Implementation Units				**6,160**	0
Total Effort Hours					**602**

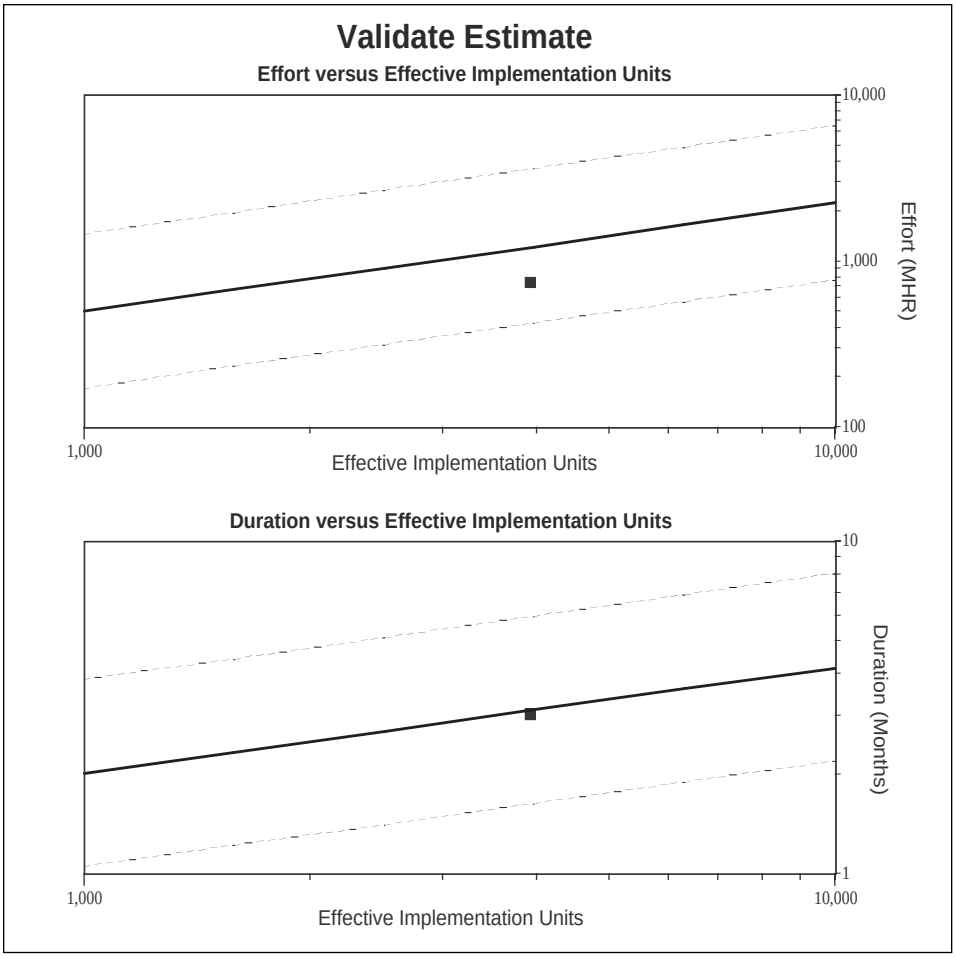


Figure 4: *Comparing an Estimate to Historical Data*

the third column are a weighting factor for each intermediate unit. If empirical data from other sizing spreadsheets is unavailable or if this is the first time this activity has been conducted, the IUs may simply be a multiple of effort. Column four contains the number of a particular intermediate unit that a project is estimated to have. The fifth column is the total estimated IUs for the intermediate unit. Column six is the total estimated effort-hours to create the intermediate units.

This is only a starting point and the IUs will be fine-tuned if required later in the process. If the developers have the project effort and intermediate units from a recently completed project, this is an excellent time to validate the estimated effort-hours on the worksheet. For instance in Figure 3, the total effort hours for the project were 602 to create the listed intermediate units. If the actual project hours of the project from which this was modeled were close to this, it lends credence to the effort estimates provided for the intermediate units. If not, it may indicate that adjustments need to be made to the effort estimates for the intermediate units or that there were intermediate units that should have been included in the project, or excluded.

Creating Estimating Templates

While interviewing the developers, it is important to have them define their project types. These may be as simple as small, medium, and large based on estimated effort hours. They may be platform-based such as Web, client server, or mainframe projects. They can also be application-specific or customer-specific.

Have the developers define what types of intermediate units are typical for each project type and a range of how many are normally found. Ask them to identify the effort range associated with each project type and identify typical durations. Tailor the estimating spreadsheets to each project type so they include only the intermediate units that project type is likely to have.

At this point the estimator can use the estimated size in IUs to create a template and calculate time, effort, and cost with a commercial parametric estimating tool.

Tuning the Process

The templates created in Figure 3 are starting points and will need to be fine-tuned. They are based on assumptions about the number of IUs per intermediate unit. How can this be refined? One method is to model completed projects using the sizing information captured on the templates as inputs to a parametric estimating tool. In this situation, project effort and duration are already known and there is an estimated size in IUs. The variable to be determined is the productivity parameter required to re-create an estimate scenario whose effort and duration match the completed project. This is a relatively easy thing to do with a parametric estimating tool. Even though solving a calculation for a missing variable will produce a result, it does not guarantee that the result is realistic. It is important to verify that the productivity parameter is reasonable when compared to industry data or organizational history.

Figure 4 demonstrates a method of comparing an estimate scenario to historical data to see if that scenario is internally consistent and reasonable. There are two graphs in Figure 4. Each has a set of trend lines calculated from a database of over 6,200 software projects. The darker line in the middle is the average. The dotted lines represent plus and minus one standard deviation, respectively. Note that a logarithmic scale is used to account for the non-linear relationship between project size and effort or duration. The X axis of both charts is project size in IUs. The Y axis on the top chart is project effort in manhours.

In this case, the effort-hours for the project (represented by the square) are slightly below the average line for similarly sized projects. The Y axis of the lower chart is project duration in calendar months. This project falls right on the average line. For this estimate scenario, both effort and duration are historically consistent with similar sized projects.

The productivity parameter used is also historically consistent[1]. If the effort were very high compared to the trend lines, it could indicate that the IUs were understated (too much effort for the amount of output). Extremely low effort compared to the trend lines would suggest that the IUs were overstated.

Similar comparisons apply for the bottom graph, too. If after modeling several projects, effort, duration, or both are consistently very high or very low, then it is a strong indication that the number of IUs for some of the intermediate units requires adjustments.

Sizing templates can be further refined as projects complete. There is one final word of caution to consider when modeling projects: The projects

should be as normal and representative of the work usually done as possible. The intent is to build a model that reflects how work is usually done. Projects with cherry-picked teams or ones that suffered from extreme schedule pressure or rework due to requirements changes are not good candidates to model. They will only skew the results.

Benefits

As Frederick Brooks [1] warned us nearly 30 years ago, there is no silver bullet. This approach to sizing may not be the best fit for every software development situation. But, it will work in many situations and has some real benefits:

- It speaks the developer's language. It describes the system in the components that developers work with: screens, reports, tables, programs, and Web pages. This improves communication.
- It involves the developers in the estimating process creating buy-in and reducing the chance of obtaining bogus data.
- It is adaptable. It allows new tools and components to be incorporated easily.
- It is an excellent way to get a handle on a new technology. It provides the ability to articulate what and how developers build a product.
- It is applicable to many different development paradigms, some of which have been difficult to estimate with parametric estimating tools:
  - o Enterprise Resource Planning (PeopleSoft and SAP [Systems, Applications, Products]).
  - o Rational Unified Process.
  - o Traditional Development.
- It can (and should) be tuned on actual project data.

If you are running into roadblocks when estimating the size of your application development projects, give this method a try. You might be pleasantly surprised by the cooperation that you receive from the technical staff, and the increased value that is attached to your end-product estimates.◆

Reference

1. Brooks, Frederick. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, 1 Jan. 1975.

Note

1. Estimating tools look at productivity from different perspectives. What is important is that however productivity is measured, there needs to be a method in place to validate it for reasonableness against organizational or industry data.

About the Authors

Lawrence H. Putnam is the founder and chief executive officer of Quantitative Software Management, Inc., and a developer of commercial software estimating, benchmarking, and control tools known under the trademark SLIM. He served 26 years on active duty in the U.S. Army and retired as a colonel. He has been deeply involved in the quantitative aspects of software management for the past 30 years. He is the co-author of five books on software estimating, control, and benchmarking. He is a member of Sigma Xi, the Association for Computing Machinery, the Institute of Electrical and Electronic Engineers (IEEE) and the IEEE Computer Society. He was presented the Freiman Award for outstanding work in parametric modeling by the International Society of Parametric Analysts. Putnam has a Bachelor of Science from the United States Military Academy and a master's degree in physics from the Naval Postgraduate School.

Quantitative Software Management, Inc.
2000 Corporate Ridge STE 900
McLean, VA 22101
Phone: (703) 790-0055
Fax: (703) 749-3795
E-mail: larry_putnam_sr@qsm.com

Douglas T. Putnam is the managing partner of Professional Services at Quantitative Software Management, Inc. (QSM). Putnam has over 24 years of experience in the software measurement industry. He has written and lectured extensively throughout the world and has participated in more than 200 estimation and measurement engagements in his career at QSM. QSM is the supplier of the trademarked SLIM suite: SLIM-Estimate, SLIM-Master Plan, SLIM-Control, SLIM-Metrics.

Quantitative Software Management, Inc.
2000 Corporate Ridge STE 900
McLean, VA 22101
Phone: (703) 790-0055
Fax: (703) 749-3795
E-mail: doug_putnam@qsm.com

Donald M. Beckett is a consultant for Quantitative Software Management with more than 20 years of software development experience, including 10 years specifically dedicated to software metrics and estimating. Beckett is a Certified Function Point Specialist with the International Function Point Users Group and has trained over 300 persons in function point analysis in Europe, North America, and Latin America. He was a contributing author to "IT Measurement: Practical Advice from the Experts." Beckett is a graduate of Tulane University.

Quantitative Software Management, Inc.
2000 Corporate Ridge STE 900
McLean, VA 22101
Phone: (703) 790-0055
Fax: (703) 749-3795
E-mail: don_beckett@qsm.com

COCOMO Suite Methodology and Evolution

Dr. Barry Boehm, Ricardo Valerdi, Jo Ann Lane, and A. Winsor Brown
University of Southern California

Over the years, software managers and software engineers have used various cost models such as the Constructive Cost Model (COCOMO) to support their software cost and estimation processes. These models have also helped them to reason about the cost and schedule implications of their development decisions, investment decisions, client negotiations and requested changes, risk management decisions, and process improvement decisions. Since that time, COCOMO has cultivated a user community that has contributed to its development and calibration. COCOMO has also evolved to meet user needs as the scope and complexity of software system development has grown. This eventually led to the current version of the model: COCOMO II.2000.3. The growing need for the model to estimate different aspects of software development served as a catalyst for the creation of derivative models and extensions that could better address commercial off-the-shelf software integration, system engineering, and system-of-systems architecting and engineering. This article presents an overview of the models in the COCOMO suite that includes extensions and independent models, and describes the underlying methodologies and the logic behind the models and how they can be used together to support larger software system estimation needs. It concludes with a discussion of the latest University of Southern California Center for Software Engineering effort to unify these various models into a single, comprehensive, user-friendly tool.

In the late 1970s and the early 1980s as software engineering was starting to take shape, software managers found they needed a way to estimate the cost of software development and to explore options with respect to software project organization, characteristics, and cost/schedule. Along with a number of commercial and proprietary cost/schedule estimation models, one of the answers to this need was the open-internal Constructive Cost Model (COCOMO). This and other models allowed users to *reason* about the cost and schedule implications of their development decisions, investment decisions, established project budget and schedules, client negotiations and requested changes, cost/schedule/performance/functionality tradeoffs, risk management decisions, and process improvement decisions [1].

By the mid-1990s, software engineering practices had changed sufficiently to motivate a new version called COCOMO II, plus a number of complementary models addressing special needs of the software estimation community. Figure 1 shows the variety of cost models that have been developed at the University of Southern California (USC) Center for Software Engineering (CSE) to support the planning and estimating of software-intensive systems as the technologies and approaches have evolved since the development of the original COCOMO in 1981.

Figure 1: *Historical Overview of COCOMO Suite of Models*

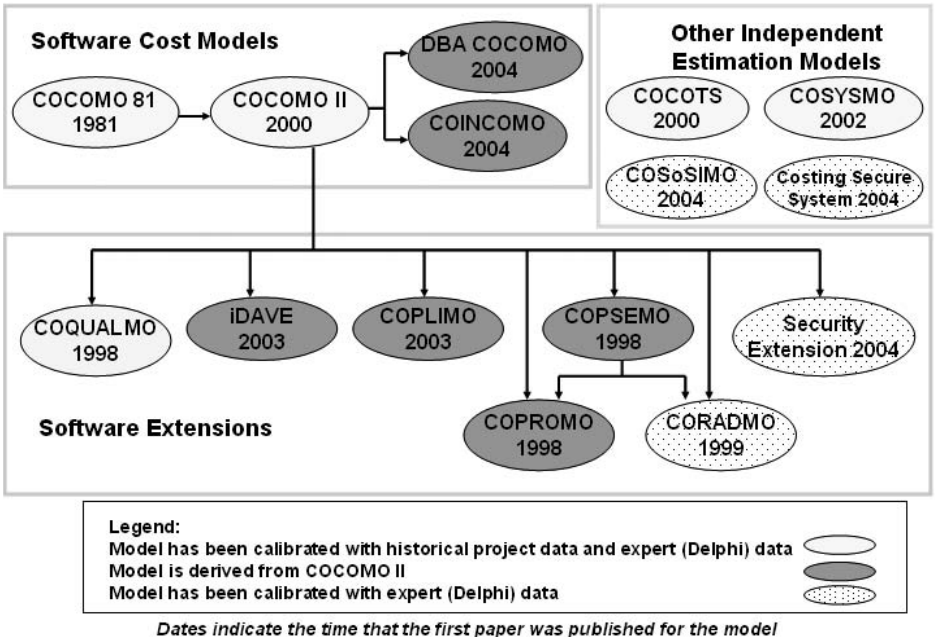


Figure 1 also shows the evolution of the COCOMO suite categorized by software models, software extensions, and independent models. The more mature models have been calibrated with historical project data as well as expert data via Delphi surveys. The newer models have only been calibrated by expert data.

Table 1 includes the status of the 12 models in the COCOMO suite. All of these models have been developed using the following seven-step methodology [2]: (1) analyze existing literature, (2) perform behavior analysis, (3) determine form of model and identify relative significance of parameters, (4) perform expert-judgment/Delphi assessment, (5) gather project data, (6) determine Bayesian A-Posteriori update, and (7) gather more data, refine model.

The checkmarks in Table 1 indicate the completion of that step for each model. Step 4 of the methodology can often involve multiple rounds of the Delphi survey that provide model developers some insight into the effects of the model parameters on development effort. The Delphi surveys attempt to capture what the experts believe has an influence on development effort.

Step 5 of the methodology involves collecting historical project data to validate the cost-estimating relationships in the model. This process depends on the support of the CSE affiliates to provide data that is relevant to the model being calibrated. The COCOMO model has more data than the other models com-

bined mostly because it has been around the longest, and it has been shown to be robust as well as accurate.

Step 6 involves combining the project data with the expert judgment captured in the Delphi survey to produce a calibrated model. This is done using Bayesian statistical techniques that provide the ability to balance expert data and historical data [2].

Model priorities, definitions, Delphi, and calibration data are collaboratively provided by the practical needs and experiences of USC CSE's supporting affiliates. These have included the major aerospace, computing, and telecommunications companies along with many of the major software and manufacturing companies, non-profits, professional societies, government organizations, and commercial cost model proprietors. For the list of CSE affiliates, visit <http://sunset.usc.edu/cse/pub/affiliate/general.html>.

The first three models (COCOMO II, COINCOMO, and DBA COCOMO) are fundamentally the same model but tailored for different development situations. In addition, commercial versions of COCOMO such as Costar <www.softstarsystems.com> and Cost Xpert <www.costxpert.com> provide further estimation-related capabilities. COQUALMO is used to estimate the number of residual defects in a software product and to provide insights into payoffs for quality investments. iDAVE estimates and tracks software dependability return on investment. COPLIMO supports software product line cost estimation and return on investment analysis. COPSEMO provides a phased distribution of effort to support incremental rapid application development and is typically used with CORADMO. COPROMO predicts the most cost effective allocation of investment resources in new technologies intended to improve productivity. All of the models described thus far are derivatives of the COCOMO model because they somehow depend on the output of COCOMO and modify it for certain situations.

The final three models are independent extensions of COCOMO that require their own inputs and can be used in conjunction with COCOMO, if desired. COCOTS estimates the effort associated with the integration of commercial off-the shelf (COTS) software products. COSYSMO estimates the systems engineering effort required over the entire system life cycle. COSOSIMO estimates the lead system integrator (LSI) effort associated with the definition and integration of software intensive system-of-systems (SoS) components.

Model	Description	Literature	Behavior	Significant Parameters	Delphi	Data
COCOMO II	Constructive Cost Model					
COINCOMO	Constructive Incremental COCOMO	✓	✓	✓	✓	>200
DBA COCOMO	DataBase (Access) Doing Business As COCOMO II					
COQUALMO	Constructive Quality Model	✓	✓	✓	✓	6
iDAVE	Information Dependability Attribute Value Estimation	✓	✓	✓		---
COPLIMO	Constructive Product Line Investment Model	✓	✓	✓		---
COPSEMO	Constructive Phased Schedule and Effort Model	✓	✓			---
CORADMO	Constructive Rapid Application Development Model	✓	✓	✓		16
COPROMO	Constructive Productivity-Improvement Model	✓	✓	✓	✓	---
COCOTS	Constructive Commercial Off-the-Shelf Cost Model	✓	✓	✓	✓	29
COSYSMO	Constructive Systems Engineering Cost Model	✓	✓	✓	✓	14
COSOSIMO	Constructive System-of-Systems Integration Cost Model*	✓	✓	✓		---

* Literature, behavior, and variable analysis limited due to number of available SoS to evaluate.

Table 1: *Status of the Models*

For more information on the COCOMO suite of models, visit: <http://sunset.usc.edu>.

Underlying Methodologies and Logic

The key to understanding the model outputs and how to use multiple models together is by comprehending the underlying methodologies and logic. In the development of a software-related cost model, the general COCOMO form is:

$$PM = A \times (\Sigma\, Size)^{\Sigma B} \times \Pi(EM)$$

where,

PM = person months.
A = calibration factor.
Size = measure(s) of functional size of a software module that has an additive effect on software development effort.
B = scale factor(s) that has an exponential or nonlinear effect on software development effort.
EM = effort multipliers that influence software development effort.

Each factor in the equation can be represented by a single value or multiple values, depending on the purpose of the factor. For example, the size factor can be used to characterize the functional size of a software module via either software lines of code *or* function points, but not both. Alternatively, the project characteristics can be characterized by a set of effort multipliers, *EM*, that describe the development environment. These could include software complexity *and* software reuse. COCOMO II has one additive, five exponential, and 17 multiplicative factors. Other models have a different number of factors that depend on the scope of the effort being estimated by that model. The number of factors in each of the models is shown in Table 2 (see next page).

The general rationale for whether a factor is additive, exponential, or multiplicative comes from the following criteria:

1. A factor that has effect on only one part of the system – such as software size – has a local effect on the system. For example, adding another source instruction, function point entity, module, interface, operational scenario, or algorithm to a system has mostly local additive effects on project effort.
2. A factor is multiplicative or exponential if it has a global effect across the overall system. For example, adding another level of service requirement, development site, or incompatible customer has mostly global multiplicative or exponential effects. If the size of

Model Name	Scope of Estimate	Number of Additive Factors	Number of Exponential Factors	Number of Multiplicative Factors
COCOMO	Software development effort and schedule	1	1	15
COCOMO II	Software development effort and schedule	1	5	17
COSYSMO	Systems engineering effort	4	1	14
COCOTS	COTS assessment, tailoring, and integration effort	3	1	13
COSOSIMO	SoS architecture and integration effort	4	6	---

Table 2: *Model Factor Types*

the product is doubled and the proportional effect of that factor is also doubled, then it is a multiplicative factor. If the effect of the factor is more influential or less influential for larger projects because of the amount of rework due to architecture and risk resolution, team compatibility, or readiness for SoS integration, then it is treated as an exponential factor.

These rules have been applied to the development of the COCOMO model as well as the associated models that have been developed at the CSE. The assumptions made about the cost estimating relationships in these models require that they be not only developed but also validated by historical projects. A crucial part of developing these models is finding representative data that can be used to calibrate the size, multiplier, and exponential factors contained in the models. The COCOMO form is a hypothesis that is tested by the data. For example, COCOTS data analysis showed that the COCOMO form applied to COTS integration, but that other forms were needed for COTS assessment and tailoring.

Table 2 summarizes the factors for the various COCOMO-independent models. The decision to have a different number of factors is determined by the Delphi process and confirmed by the data analysis, either of which can add or subtract factors from a model. However, the same criteria for factor type are used in all of the models. The COCOMO II extensions (shown in Figure 1) are based on the initial COCOMO II estimates with additional factors incorporated for the software characteristic of interest.

Understanding the scope of each model is also a key element in understanding the output it provides. The models in the COCOMO suite provide a specialized set of estimates that address specific aspects of development effort for software-intensive systems. COCOMO users are now beginning to use multiple models in parallel to develop cost estimates that cover a broader scope that exceeds the boundaries of traditional software development. In this case, the models in the COCOMO suite provide a set of tools that enable more comprehensive cost estimates. However, there are some limitations that exist when using multiple models together. These limitations are discussed in the next section.

Using Current Models Together

Many benefits exist when using multiple models in parallel. For one, they provide a more comprehensive set of estimates that better reflect the true effort associated with developing a software system. The effort that is not accounted for in COCOMO may be covered by other models such as COCOTS, COSYSMO, and COSOSIMO. Secondly, they enable the estimator to characterize the system in terms of multiple views.

However, some complications can arise when any two of these models are used in parallel since each of the models was initially developed as an independent entity. Just as the process model community has found that software engineering, software development, system engineering, and other activities are integrated, have dependencies, and cannot be adequately performed and optimized independently of each other, the estimation community has also found that these activities cannot be estimated independently for many of the larger software-intensive systems and SoS. Activities need to be planned and estimated at a program or project level.

Feedback from USC CSE affiliates and other COCOMO model users [3, 4] indicates that users would like a single tool in which they can do the following:

- Identify system and software components comprising the software system of interest.
- Easily evaluate various development approaches and alternatives and their impacts to cost and schedule.
- Understand the overlaps between models, if any.

Moving Forward – COCOMO Suite Unification

Efforts have been initiated at the USC CSE to develop a framework in which the key cost models can be integrated to provide a comprehensive software-system development effort to users. Once the models that are most likely to be used together are integrated, efforts will focus on the integration of other more specialized models. We will also begin with the models that have a high degree of maturity.

The purpose of this unification effort is similar to that of the individual cost models [2], that is, to help software-intensive system and SoS developers and their customers reason about the cost and schedule implications of their development decisions, investment decisions, risk management decisions, and process improvement decisions.

Key to our approach is distinguishing between an *integrated* set of models versus a truly *unified* model. When a set of models is integrated, typically each model becomes an entity in the integrated set with inputs into one model creating outputs that are then fed into subsequent models. However, when a unified model is developed, there is a reengineering of the set of models to come up with an architecture where the whole of the unified set is greater than the sum of the parts. Developing a unified COCOMO suite model will support the goals to minimize or eliminate overlap between the models, provide a relatively comprehensive coverage of the SoS, system engineering, and software development activities, and develop a relatively simple interface for specifying inputs as well as a well-integrated set of outputs.

Key Unification Issues

In August 2004, the CSE held an internal workshop to identify key issues for model unification. The outcome of the workshop was the identification of four areas of focus for unification: (1) selection of models that must be unified to support various types of development, (2) identification of the overlap between these models, (3) identification of missing activities not covered by any of the current models, and (4) specification of the required parameters and outputs for the related models in a user-friendly, consistent, and usable manner. The following sections describe some of the more detailed issues identified as part of the four focus areas.

Model Selection

Many of today's large software-intensive systems integrate legacy capabilities, COTS software products, and new custom software subsystems. No single COCOMO model covers the full life-cycle effort for the development of these types of sys-

tems. The new software development effort is easily estimated using COCOMO II. COTS customization effort might be estimated using another COCOMO suite model: COCOTS. COSYSMO would typically be used to estimate the system-level engineering activities such as feasibility analysis to support the integration concept, functional analysis of the new requirements, trade-off studies, prototyping, performance evaluation, synthesis, and system verification and validation activities. And finally, COSOSIMO might be used to estimate the effort associated with the integration of the legacy system with the COTS system and the new custom software system. CSE corporate affiliates have identified potential combinations of cost models that would be of value to them, including COCOMO/COSYSMO/COCOTS and COCOMO/COSYSMO/COSOSIMO [4].

Model Overlap

Further analysis is required to determine the extent of any overlap between the various COCOMO models. Potential overlap issues were identified with respect to various combinations of the primary cost models as well as with respect to the general integration of software and system components.

- **COCOMO II and COSYSMO Model Overlap:** Currently, COCOMO II is designed to estimate the software effort associated with the analysis of software requirements and the design, implementation, and test of software. COSYSMO estimates the system engineering effort associated with the development of the software system concept, overall software system design, implementation, and test. Key to understanding the overlap is deciding which activities are considered *system engineering* and which are considered *software engineering/development*, and how each estimation model handles these activities.
- **COSYSMO and COSOSIMO Model Overlap:** COSOSIMO aims to estimate the effort associated with the architecture definition of a *SoS* as well as the effort associated with the integration of the highest level SoS components. On the other hand, COSYSMO estimates are done in the context of a single system and include the effort needed to define a single, system-level architecture, the design of the system components, and the integration of those components. COSYSMO also includes the effort required for the system development to support the integration of the system component in the target environment. Further work is required to understand the subtleties of these models and exact extent of any overlap between these models.

Missing Activities

Are there any key activities missing when the key models are viewed together? How are specialty engineering tasks for secure or sensitive systems handled? How are non-software system development tasks handled? What about logistics planning for operational support? Can effort from activities not supported by any current COCOMO model be easily integrated?

Effort Outputs

What granularity should be provided? One effort value? An effort value for each of the key models? By software component? By system component? By engineering category (e.g., software, systems engineering, LSI)? By phase/stage of development?

Understanding Unification Issues

To begin to understand these four unification issues better and to start developing a candidate approach for the unified COCOMO model, efforts were initiated to better understand the following:

- Current model boundaries.
- How the current models are typically used today.
- The activities associated with software development, system engineering, and SoS integration work performed by LSIs.
- What activities are included in each of the current primary cost models.

Current Model Boundaries and Usage

To address this first aspect, we developed a table to indicate when each model (or set of models) is typically used (Table 3). As part of this effort, we developed descriptions that tried to capture information about the current boundaries of each model and how those boundaries expand as the current models are used in an integrated manner.

Types of Effort Currently Estimated

The next step was to identify a comprehensive set of high level, software-intensive system life-cycle activities, the typical development organizations responsible for the performance of these activities, and the scope of the activity typically performed by each development organization. Then each activity covered by each of the primary cost models was identified. For example, the system engineering organization is typically responsible for the system/subsystem requirements and design, and the software development organization participates in a support or review role. Other activities, such as management, are often performed at various levels with each development organization having primary responsibility at their respective levels.

The results of this effort are shown in Table 4 (see next page). The shaded activities under Software Development are currently covered in COCOMO II and COCOTS. The shaded activities under System Engineering are currently estimated by COSYSMO. The shaded activities under LSI are currently estimated by COSOSIMO. The activities that are not shaded are currently not covered by any of the models in the COCOMO suite. And, since the focus of the COCOMO suite is on software-intensive systems, none of the items under the hardware development column are currently covered.

Some activities such as management and support, involve several organizations

Table 3: *How Current Primary Cost Models Are Typically Used*

Use ...	When scope of work to be performed is ...
COCOMO II	Development of software components (software development).
COCOTS	Assessment, tailoring, and integration of COTS products.
COSYSMO	Design, specification, and integration (system engineering) of system components to be separately developed for a single system.
COSOSIMO	Specification, procurement, and integration of two or more separately system-engineered and developed systems.
COCOMO II with COCOTS	Development of software components (software development), and a software system, including assessment, tailoring and glue-code for integration of COTS.
COSYSMO and COCOMO II	System engineering and software development for a single system with software-intensive components.
COSYSMO and COSOSIMO	System engineering of individual systems and integration of the multiple systems.
COCOMO II, COSYSMO, COCOTS, and COSOSIMO	System engineering, software development, and integration of multiple software-intensive systems and COTS products.

Activity	Responsibilities			
	Software Development (COCOMO II and COCOTS)	Hardware Development	System Engineering (COSYSMO)	LSI (COSOSIMO)
Management	Primary for Software Level	Primary for Hardware Level	Primary for System Level	Primary for SoS Level
Support Activities (e.g., Configuration Management and Quality Assurance)	Software Level	Hardware Level	System Level	SoS Component Level
SoS Definition			SoS Component	SoS Level
Source Selection and SoS Component Procurement				Lead
Subsystem Requirements	Review	Review	Elaboration* Lead	Inception Lead
System/Subsystem Design	Support	Support	Lead	Review
Hardware/Firmware Development		Lead		
Software Requirements Analysis	Elaboration* Lead		Inception Lead	
Software Product Design	Lead		Review	
Software Implementation/ Programming	Lead		Support	
Software Test Planning	Lead		Review/Support	
Software Verification and Validation	Lead		Review/ Support	
System Integration/Test	Support	Support	Lead	Review
System Acceptance Test	Support	Support	Lead	Review
SoS Integration/Test	Support	Support	Review/Support	Lead
SoS Acceptance Test	Support	Support	Review/Support	Lead
Manuals (User, Operator, Maintenance)	Software Lead	Hardware Lead	System Lead	SoS Level Lead
Transition (Deploy and Maintain)	Support	Support	System Lead	SoS Level Lead

* Model Based (System) Architecting and Software Engineering/Rational Unified Process phase of development.

Table 4: *Life Cycle Activities*

at different layers of the system. Extreme care needs to be taken when developing models that cover activities that have shared responsibilities with hardware, software, and other players.

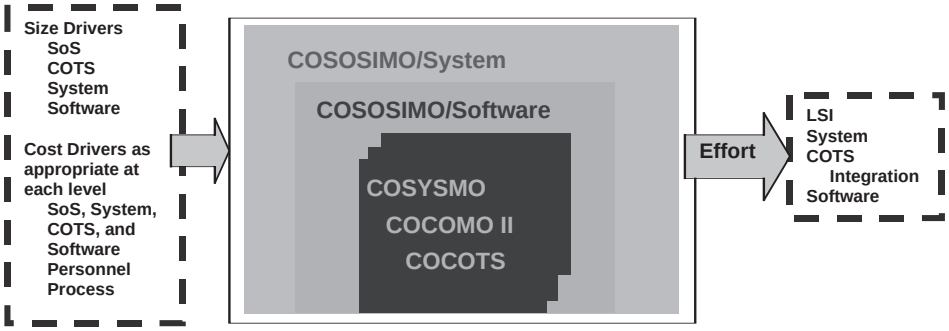
The identification of such activities is the first step in identifying possible overlaps between models. Further difficulties arise when dealing with different organizations that use customized work breakdown structures. These, along with the aforementioned challenges, will continue to be addressed as the model unification efforts continue at the CSE.

As seen from the discussions above, there is still much work to be done in order to support the unification of the COCOMO models. These include the following:

1. Develop a more complete description of activities covered by each model. These descriptions will allow us to identify, minimize, or eliminate any overlap between the models and identify software system-related activities not covered by any of the models.
2. Determine more precisely how traditional phase activities and Model Based (System) Architecting and Software Engineering/Rational Unified Process [1] phases map to cost-model activities and how these phases are integrated at the SoS, system, and software levels. Work in this area has already begun [5] but some unresolved issues remain in the context of unified models.
3. Refine counting rules/definitions for model inputs and outputs and then determine how they can be combined into an efficient, user-friendly unified model.
4. Determine typical distribution profiles for effort across all of the activities/ phases in a unified environment.

The initial goal of this effort is to develop a unified model that includes COCOMO II, COSYSMO, COCOTS and COSOSIMO as shown in Figure 2. As we learn from this process, we will begin to add other models from the COCOMO suite.

Figure 2: *Early Unification Goal*



The current unification effort will help establish a framework and define the context for the evolution of the unified model into something that can provide a comprehensive estimate for the development of software systems and software-intensive SoS. We will continue to collaborate with CSE affiliates with the goal of evolving the COCOMO suite so that it can help users make better decisions about the development of software-intensive systems.◆

References

1. Boehm, B. Software Engineering Economics. Prentice Hall, 1981.
2. Boehm, B., et al. Software Cost Estimation with COCOMO II. Prentice Hall, 2000.
3. Annual Research Review, Corporate Affiliate Survey. University of Southern California Center for Software Engineering, 16 Mar. 2004.
4. University of Southern California Center for Software Engineering. "Unification Workshop Minutes." 19th Forum on COCOMO and Software Cost Modeling, 26 Oct. 2004.
5. Boehm, B., A.W. Brown, V. Basili, and R. Turner. "Spiral Acquisition of Software-Intensive Systems of Systems." CrossTalk May 2004: 4-9.

About the Authors

Barry Boehm, Ph.D., is the TRW professor of software engineering and director of the Center for Software Engineering at the University of Southern California. He was previously in technical and management positions at General Dynamics, Rand Corp., TRW, and the Defense Advanced Research Projects Agency, where he managed the acquisition of more than $1 billion worth of advanced information technology systems. Boehm originated the spiral model, the Constructive Cost Model, and the stakeholder win-win approach to software management and requirements negotiation.

E-mail: boehm@usc.edu

Jo Ann Lane is currently a doctorate student at the University of Southern California in systems architecting. Prior to this, she was a key technical member of Science Applications International Corporation's Software and Systems Integration Group. She has over 28 years of experience in the areas of software project management, software process definition and implementation, and metrics collection and analysis. Lane has a Master of Science degree in computer science from San Diego State University.

E-mail: jolane@usc.edu

Ricardo Valerdi is a member of the Technical Staff at the Aerospace Corporation. Previously, he worked as a systems engineer at Motorola and General Instruments. He is a doctorate candidate at the University of Southern California (USC) in the systems architecting program and is a research assistant at USC's Center for Software Engineering. Valerdi has a Bachelor of Science in electrical engineering from the University of San Diego and a Master of Science in systems architecting from USC.

E-mail: rvalerdi@sunset.usc.edu

A. Winsor Brown is a senior research scientist and assistant director of the University of Southern California Center for Software Engineering. As an engineer with decades of experience in large and small commercial and government contracting companies, he started his career in computer hardware design but shifted to software within months and remains there today. He has a Bachelor of Science in engineering science from Rensselaer Polytechnic Institute and a Master of Science in electrical engineering from California Institute of Technology.

E-mail: awbrown@usc.edu

CrossTalk
The Journal of Defense Software Engineering

Get Your Free Subscription

Fill out and send us this form.

OO-ALC/MASE
6022 Fir Ave
Bldg 1238
Hill AFB, UT 84056-5820
Fax: (801) 777-8069 DSN: 777-8069
Phone: (801) 775-5555 DSN: 775-5555

Or request online at www.stsc.hill.af.mil

Name:_____

Rank/Grade:_____

Position/Title:_____

Organization:_____

Address:_____

_____

Base/City:_____

State:_____Zip:_____

Phone:(____)_____

Fax:(____)_____

E-mail:_____

Check Box(es) To Request Back Issues:

Nov2003	☐	Dev. of Real-Time SW
Dec2003	☐	Management Basics
Jan2004	☐	Info from Sr. Leadership
Mar2004	☐	SW Process Improvement
Apr2004	☐	Acquisition
May2004	☐	Tech.: Protecting Amer.
Jun2004	☐	Assessment and Cert.
July2004	☐	Top 5 Projects
Aug2004	☐	Systems Approach
Sept2004	☐	Software Edge
Oct2004	☐	Project Management
Nov2004	☐	Software Toolbox
Dec2004	☐	Reuse
Jan2005	☐	Open Source SW
Feb2005	☐	Risk Management
Mar2005	☐	Team Software Process

To Request Back Issues on Topics Not Listed Above, Please Contact Karen Rasmussen at <stsc.customerservice@hill.af.mil>.

Inside SEER-SEM

Lee Fischman, Karen McRitchie, and Daniel D. Galorath
Galorath, Inc.

The System Evaluation and Estimation of Resources - Software Estimating Model (SEER-SEM) is a commercially available software project estimation model used within defense, government, and commercial enterprises. Introduced over a decade ago and now in its seventh release, it offers a case study in the history and future of such models. SEER-SEM and its brethren are built upon a mix of mathematics and statistics; this article provides insight into its inner workings and basis of estimation.

If you follow the roots of software estimation models, you will find many have common ancestors. The System Evaluation and Estimation of Resources - Software Estimating Model (SEER-SEM) began with the Jensen model and diverged significantly in the early 1990s. Barry Boehm's Constructive Cost Model work provided for the redefinition of some of the original Jensen model parameters into SEER-SEM. Don Reifer and Dan Galorath's work on the NASA Softcost model also found its way into SEER-SEM in addition to Halstead's software science metrics. The Jensen model itself was first calibrated using some of the same data as the Putnam model. Earlier work by Doty Associates introduced the idea of factoring in development environment influences via parameters. Work on this model continues today.

SEER-SEM's Architecture

SEER-SEM is composed of a group of models working together to provide estimates of effort, duration, staffing, and defects. These models can be briefly described by the questions they answer:

- **Sizing.** How large is the software project being estimated?
- **Technology.** How productive are the developers?
- **Effort and Schedule Calculation.** What amount of effort and time are required to complete the project?
- **Constrained Effort/Schedule Calculation.** How does the expected project outcome change when schedule and staffing constraints are applied?
- **Activity and Labor Allocation.** How should activities and labor be allocated into the estimate?
- **Cost Calculation.** Given expected effort, duration, and the labor allocation, how much will the project cost?
- **Defect Calculation.** Given product type, project duration, and other information, what is the expected, objective quality of the delivered software?
- **Maintenance Effort Calculation.** How much effort will be required to adequately maintain and upgrade a fielded software system?

Software Sizing

Software size is a key input to any estimating model, SEER-SEM being no exception. Supported sizing metrics include source lines of code (SLOC), function-based sizing (FBS) and a range of other measures. They are translated for internal use into effective size (S_e). S_e is a form of common currency within the model and enables new, reused, and even commercial off-the-shelf code to be mixed for an integrated analysis of the software development process. The generic calculation for S_e is:

$$S_e = NewSize + ExistingSize \times (0.4 \times Redesign + 0.25 \times Reimpl + 0.35 \times Retest)$$

As indicated, S_e increases in direct proportion to the amount of new software being developed. S_e increases by a lesser amount as preexisting code is reused in a project. The extent of this increase is governed by the amount of rework (redesign, re-implementation, and retest) required to reuse the code.

Function-Based Sizing

While SLOC is an accepted way of measuring the absolute size of code from the developer's perspective, metrics such as function points capture software size functionally from the user's perspective. The function-based sizing (FBS) metric extends function points so that hidden parts of software such as complex algorithms can be sized more readily. FBS is translated directly into unadjusted function points (UFP).

In SEER-SEM, all size metrics are translated to S_e, including those entered using FBS. This is not a simple conversion, i.e., not a language-driven adjustment as is done with the much-derided *backfiring* method. Rather, the model incorporates factors, including phase at estimate, operating environment, application type, and application complexity. All these considerations significantly affect the mapping between functional size and S_e. After FBS is translated into function points, it is then converted into S_e as:

$$S_e = L_x \times (AdjFactor \times UFP)^{(Entropy/1.2)}$$

where,

L_x is a language-dependent expansion factor.
AdjFactor is the outcome of calculations involving other factors mentioned above.
Entropy ranges from 1.04 to 1.2 depending on the type of software being developed.

Effort and Duration Calculations

A project's effort and duration are interrelated, as is reflected in their calculation within the model. Effort drives duration, notwithstanding productivity-related feedback between duration constraints and effort. The basic effort equation is:

$$K = D^{0.4}(S_e/C_{te})^{1.2}$$

where,

S_e is effective size – introduced earlier.
C_{te} is effective technology – a composite metric that captures factors relating to the efficiency or productivity with which development can be carried out. An extensive set of people, process, and product parameters feed into the effective technology rating. A higher rating means that development will be more productive.
D is staffing complexity – a rating of the project's inherent difficulty in terms of the rate at which staff are added to a project.

The general form of this equation should not be a surprise. In numerous empirical studies, the effort-size relationship has been seen to assume the general form $y = a \times size^b$ with a as the linear multiplier on size, and the exponent ranging between 0.9 and 1.2 depending on available data. Most experts feel that $b>1$ is a reasonable assumption, translated as *effort increases at a proportionally faster rate than size*. While SEER-SEM's value of 1.2 is at the high end of this range, the formula above is only part of the estimating process.

Once effort is obtained, duration is solved using the following equation:

$$t_d = D^{-0.2}(S_e/C_{te})^{0.4}$$

The duration equation is derived from key formulaic relationships (not detailed here). Its 0.4 exponent indicates that as a project's size increases, duration also increases, though less than proportionally. This size-duration relationship is also used in component-level scheduling algorithms with task overlaps computed to fall within total estimated project duration.

Time/Schedule Tradeoffs

In software projects, a limited exchange can be made between required effort and schedule. In fact, SEER-SEM optimizes according to minimum time or optimal effort scenarios. The first implies that a software project will staff aggressively to finish in the minimum amount of time, while the alternative permits schedule slippage for the sake of effort savings. The trade between minimum time and optimal effort is shown in Figure 1.

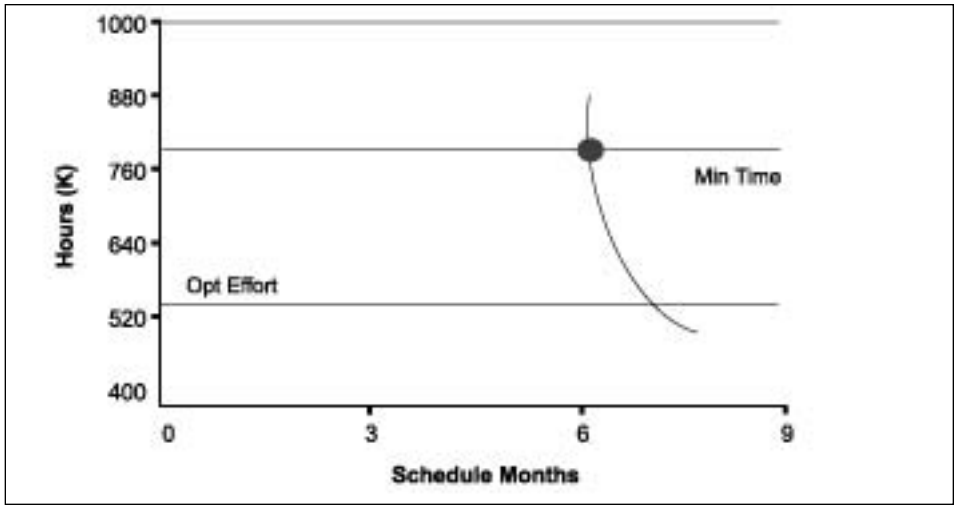


Figure 1: *Effort Schedule Tradeoff*

Staffing Constraints

Oftentimes specific staffing levels need to be factored into an estimate. Other factors aside, lower staffing leads to higher productivity per programmer while increased staffing reduces productivity. The dynamic relation between staffing and productivity can be described by an optimal staffing curve as shown in Figure 2.

The curve depicts optimal staffing over time for an idealized project. Its shape varies depending on project size and complexity. Areas around the curve illustrate the impact on individual productivity when staffing at any time varies from optimal. When staffing is too high, there is a productivity penalty as increased coordination is required while more staff must spend time getting up to speed. When staffing is too low, productivity increases due to tighter coordination among fewer staff and from team members who on average are more expert. Adding more staff may increase a team's ability to get work done but every additional person added is slightly less effective than the last.

Detailed Allocations of Effort and Duration

Project planners often need to know how a project's overall estimated effort and duration are allocated into specific activities and labor categories. While allocations are partially determined by patterns seen in past projects, they will vary for each project according to its unique characteristics. For example, there may be more or less requirements activity, testing, etc. Table 1 (see next page) provides a typical allocation, by percentage, of project effort into a matrix of labor types and activities.

Calibrating SEER-SEM

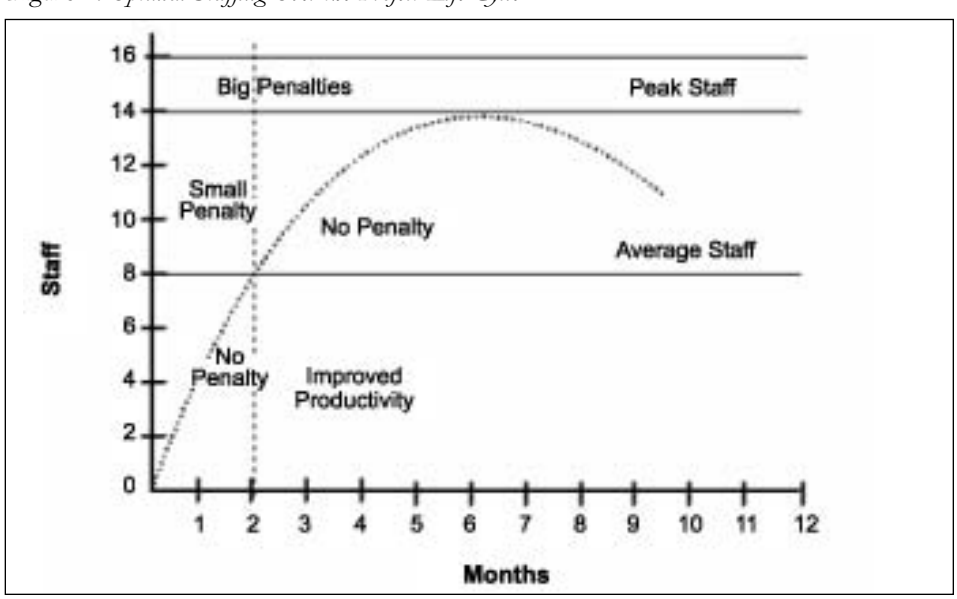
Key components of the SEER-SEM model have been described, but we have not discussed how it adapts to accurately estimate particular development scenarios, and how the model is kept current as software development technologies and methodologies evolve. The answer is simple: masses of ongoing research and analysis.

The modeling team regularly combs through raw data and industry studies to determine the latest trends and their impact on project productivity. As part of this effort, Galorath maintains a software project repository of approximately 6,000 projects (and growing). About 3,500 projects containing effort and duration outcomes are stored in a unified repository that can be readily accessed for studies. These are from both defense and commercial sources representing many development organizations, permitting calibration of the model to a wide array of potential projects. Additional project outcomes, in the hundreds, are also available to the company, which has also collected sizing and other information on thousands of additional projects.

Analysis involves running project data through SEER-SEM using a special calibration mode. The model is essentially run backwards to find calibration factors. These factors are evaluated across different data attributes (e.g. platform, application, etc.) to detect trends. A variety of methods are used to mitigate outlier data points and control for variation. The variance in the data set is also used to establish default parameter ranges; nearly all settings accommodate risk. Model settings are updated as new trends are established.

Galorath's work also is leveraged with findings from outside studies. For example, when examining relative language pro-

Figure 2: *Optimal Staffing Over the Project Life Cycle*



Labor Categories								
Activities	Management	Software Requirements	Design	Code	Data Prep	Test	Configuration Management	Quality Assurance
System Requirements Design	0.2%	0.7%	0.2%	0.0%	0.1%	0.2%	0.0%	0.0%
Software Requirements Analysis	0.5%	1.8%	0.6%	0.3%	0.3%	0.5%	0.1%	0.1%
Preliminary Design	0.9%	0.9%	3.6%	1.0%	0.7%	1.2%	0.2%	0.2%
Detailed Design	1.6%	1.5%	6.0%	1.8%	1.2%	2.1%	0.3%	0.3%
Code and Unit Test	1.6%	0.7%	1.4%	12.8%	1.4%	3.5%	0.9%	0.9%
Component Integrate and Test	2.2%	0.6%	1.1%	10.9%	2.2%	8.1%	1.4%	1.4%
Program Test	0.3%	0.1%	0.2%	1.3%	0.3%	0.9%	0.2%	0.2%
System Integrate Thru OT and E	1.4%	0.3%	0.7%	3.2%	0.2%	9.8%	0.9%	0.3%

Table 1: *Allocation of Activities and Labor for a Sample Project in SEER-SEM*

ductivity, the company first uses its repository to empirically determine the impact of using different languages. However, because not all languages are well covered, it turns to outside sources that provide language descriptions, evolution trees, multidimensional comparisons, etc. Putting all this information together permits the company to make informed judgments about even rarely occurring languages.

Cost estimation models must be able to estimate a wide array of projects. This is accomplished with a significant number of modeling instruments, most of which can be independently set by the user:

- **Sizing Measures.** Software's effective size varies according to many factors, and these factors change over time. As new languages are added to the developer's toolbox and old ones evolve, language mappings get updated. Sizing proxies also permit entirely new metrics to be added.
- **Knowledge Bases.** New platforms (or operating environments) and applications are regularly being identified and added to SEER-SEM by way of its knowledge bases. Knowledge bases actually represent collections of parameter settings. Parameters in turn cover many different facets of the development process and of a software product's potential characteristics; new platforms and applications usually can be defined with a collection of parameter settings.
- **Allocations.** According to project type, the balance shifts between types of activities and labor. Within SEER-SEM, detailed activity milestone and labor allocation tables are used to establish baseline allocations, which are then further adjusted depending on project-specific settings related to requirements, testing, and so forth.
- **Internal Calibrations.** Several internal instruments, both linear and nonlinear, permit high-level, systematic adjustments to estimates.

Beyond the Model

While this article has dealt exclusively with the core SEER-SEM model, other aspects of the tool are critically important to its practical application. Among its key design philosophies is the use of qualitative rating scales, user-selectable knowledge bases for basic calibration, and a work breakdown structure that differentiates between the system, program, and component levels. The SEER-SEM model will itself soon be complemented with a data mining system that produces entirely dynamic, data-driven estimates.◆

About the Authors

Lee Fischman is Special Projects director at Galorath Incorporated, where he develops new concepts, produces new software applications, and conducts research projects. His research interests include software metrics theory and novel applications of estimating algorithms. He received a Bachelor of Arts in economics from the University of Chicago and a Master of Arts in economics from University of California Los Angeles.

Galorath Incorporated
100 N Sepulveda BLVD
STE 1801
El Segundo, CA 90245
Phone: (310) 414-3222
E-mail: info@galorath.com

Karen McRitchie is vice president of Development at Galorath Incorporated, and is responsible for design, development and validation of current and new System Evaluation and Estimation of Resources tools. For her longstanding contribution to commercial cost prediction tools, the International Society of Parametric Analysts honored her with its 2002 Parametrician of the Year award. McRitchie has a Bachelor of Arts in mathematics and system science from the University of California Los Angeles, and completed Master of Art degree work at California State University, Northridge.

Galorath Incorporated
100 N Sepulveda BLVD
STE 1801
El Segundo, CA 90245
Phone: (310) 414-3222
E-mail: info@galorath.com

Daniel D. Galorath founded and is president of Galorath Incorporated. He has solved a variety of management, costing, systems, and software problems, performing all aspects of software development and management. His company has developed tools, methods, and training for software cost, schedule, risk analysis, and management decision support, including the industry standard System Evaluation and Estimation of Resources-Software Estimating Model. Galorath has a Bachelor of Arts and a Master of Business Administration from California State University, Dominguez Hills.

Galorath Incorporated
100 N Sepulveda BLVD
STE 1801
El Segundo, CA 90245
Phone: (310) 414-3222
E-mail: info@galorath.com

The Statistically Unreliable Nature of Lines of Code

Joe Schofield
Sandia National Laboratories

For the past three decades, the ill-defined line of code has been used to describe the size of a software project and often used as a basis for estimating schedule and resource needs. Concurrently, software projects are noted for cost and schedule overruns, and often, for poor quality. This article suggests that the venerable line of code measure is a major factor in poorly scoped and managed projects because it is itself a vague, ambiguous, and unsuitable parameter for sizing software projects. A series of Personal Software Process℠ courses is the source of the data in this article. Because the requirements, instructor, and the lines-of-code counting-specification for these programs were the same, the 60 sets of nine programs offers an extraordinary opportunity for comparing significant variation in software sizes for identical requirements. Given the variation, often greater than an order of magnitude for identical requirements, the use of lines of code as a reliable indicator of software size is challenged.

The Information Systems Development Center within Sandia National Laboratories began a journey with software process improvement using the Capability Maturity Model® for Software as its improvement yardstick in 1999. The Personal Software Process℠ (PSP℠) and Team Software Process℠ were adopted soon thereafter to improve the personal and team practices of the software engineers in the organization.

The rigorous and consistent collection of measurement data prescribed as part of the PSP (and in the examined classes) provides a fertile environment for understanding how software size is estimated versus its actual size upon completion. More interesting though is the study of the size of the software products developed by numerous classes and class attendees for class projects using homogenous and heterogeneous software languages. Both casual heuristic analysis and statistical analysis of these sets of data raise serious suspicions regarding the reliability of using lines of code (LOC) as a software sizing measure.

Software Size Has No Monopoly on Ambiguity

Parents deal with ambiguity when they ask their teenagers when they will be home, only to hear "pretty soon." Spouses experience ambiguity when asking, "How long until dinner?" only to hear, "In a minute." Most consumers at one time or another have purchased *jumbo* shrimp. Science describes distant galactic formations as *small* supernovas. Meteorologists contribute their share to ambiguity by using phrases like *partly cloudy*, *partly sunny*, and apparent synonyms like *mostly sunny* and *mostly cloudy*, respectively.

The least satisfying of these descriptions parallel software customers who are told that their proposed software will be 5 million LOC. Anyone who has ever suspected that the figure 5 million is neither reliable nor accurate will more fully understand some of that discomfort upon completing this article. Anyone who has provided similar numbers for project sizes in the past may be reluctant to ever do so again.

> *"This article suggests that using LOC as a measure for actual product delivery has such wide variation as to render the counts practically useless in the best case, harmful and misleading in the worst of cases."*

This article is not the first to raise questions surrounding the use of LOC. The Definition Checklist for Source Statements Counts identifies 66 variations in counting LOC to document, and as many as eight more that are language-specific [1]. Capers Jones offers this insight on LOC:

> This term is highly ambiguous and is used for many different counting conventions. The most common variance concerns whether physical lines of logical statements comprise the basic elements of the metrics. Note that for some modern programming languages that use button controls, neither physical lines nor logical statements are relevant. [2]

Why Substantial Data on LOC Studies Is Lacking

For data to be exchanged across organizations for benchmarking and eventual insights and learning, a standard definition of a line of code would need to be accepted and applied to participating groups. For an organization to apply data from its own projects for process insight and estimation, many factors need to be identified to minimize the sources of variation that could easily render any gleanings virtually useless. A preferred practice without a context is often a worst practice in another case. Some of the limitations of purported studies related to LOC suffer from one or more of the following challenges.

- **Too few controlled studies.** Many studies of LOC are merely reflections of the type of software, language, and environment in which it was developed. But requirements rigor, design constraints, and customer turnover often contribute as sources of undocumented variation in the development of software size and duration.
- **Too few controlled studies with multiple instantiations of the same set of specifications.** Few organizations can afford to sponsor the repeated development of software code by different software engineers for the

Course / Attendee	P1	P2	P3	P4	P5	P6	P7	P8	P9
2 / 5	193	137	48	102	107	207	118	67	134
1 / 2	77	163	168	123	134	164	238	178	135
3 / 1	73	37	36	95	101	138	51	66	181
3 / 2	74	97	143	153	279	146	176	80	305
3 / 4	114	71	108	80	219	189	142	95	163
Min. Value	73	37	36	80	101	138	51	66	134
Max. Value	193	163	168	153	279	207	238	178	305
Percent Variation	264	441	467	191	276	150	467	270	228
Mean	106	101	101	111	168	169	145	97	184
Std. Dev.	51	50	58	28	78	29	69	47	71

Table 1: *Lines of Code Counts for PSP Classes by Programming Language No. 1*

purpose of measuring variations in the size of the software.

- **Too few controlled studies with multiple instantiations for different languages.** Few organizations can afford to sponsor the repeated development of software code using different languages for the purpose of measuring variations in the size of the software.
- **Inconsistent measurement approaches.** Few organizations can afford to sponsor the repeated development of software code and then analyze the source of variation attributed to how the software was measured.

Addressing the Preceding Challenges

A PSP course provides an environment that addresses the challenges related to collecting software size measures in the preceding section. Thus, the software measures in the following six tables are extracted from a series of PSP classes taught by the same Software Engineering Institute-certified PSP course instructor. Each class required the attendee to write nine software programs in a language of their choosing – typically the language with which the attendee was most proficient. Each program had associated requirements and acceptance criteria evaluated by the same instructor.

The data from the PSP course was collected in a controlled environment facilitating the close examination of 60 sets of nine software programs (60 students wrote nine programs each). The LOC for each program were counted using the same counting techniques, a point that is proven with the data from the courses (discussed on page 31 and Table 6 on page 32). One of the programs was itself a line-counting program, thus its specification and review reduces one significant source of variation in the counts – the counting method. Reduced variation in counting technique increases the reliability in the numbers used. In those PSP classes in which different languages were used, also present were different levels of education; all participants had at least a bachelor's degree, and about one-half of the attendees had an advanced degree.

Examining the Data

Tables 1-3 cluster the LOC counts for PSP classes by programming language. Using the same format, each table includes columns for the course number and attendee identifier, and the number of LOC for each of the nine programs. The bottom rows include analytic data deriving the minimum and maximum line counts for that set of programs using the same language, the percent of variation between the minimum and maximum values, and the mean and standard deviation of the LOC counts.

The shaded *Percent Variation* (the shaded row in Table 1) for the first shaded cell should be read as a variance of 264 percent between the largest and the smallest programs in this data grouping. Recall that all of the values in each of the P1-P9 columns of this table are derived from software programs written from the same requirement set, validated by the same instructor, using the same language, and counted the same way. Note that a variance of 264 percent is probably not acceptable in purchasing a home (the same home, built to the same specification, inspected by the same inspector, and measured identically), a car, or most consumer or industrial products or services.

A second set of data in Table 2 demonstrates increasing concern. The data collected from this data set came from one class where all the attendees used the same language, but a different language than in Table 1. Note that the smallest percent variation with these programs is almost 400 percent and the largest is more than 2,200 percent. Imagine, for example, the variation on the amount of gasoline received at the local filling station varied between four and 22 times, or the accuracy on the fuel gauge in an aircraft varied this much, or the number of donuts in a dozen, or the amount of beef in your favorite hamburger.

A more troublesome question is, "Which value does the project leader use to make an estimate of the size and, eventually, the cost and resources associated with software?" Are the traditional reasons offered for runaway software projects likely to be as causal as the variations in the size of the code that is developed? Is requirements creep, requirements churn, or team turnover likely to cause a variation of 2,200 percent on a project? Is almost everything we believe about estimating and managing software projects incorrect? How might the true unpredictable size of software using LOC change what we believe about productivity, defects, or reuse?

Lastly, Table 3 contains the values of the third programming language used in the PSP courses. The range of variance is

Table 2: *Lines of Code Counts for PSP Classes by Programming Language No. 2*

Attendee (same course)	P1	P2	P3	P4	P5	P6	P7	P8	P9
1	221	128	103	227	186	306	155	61	283
2	35	143	114	13	110	63	113	84	85
3	113	106	36	34	53	51	54	61	125
4	90	38	51	61	134	99	43	58	126
5	117	311	271	289	142	122	190	383	219
6	131	179	56	150	202	185	155	118	144
7	184	30	15	30	61	116	69	43	147
8	73	96	102	197	64	158	85	87	126
9	64	63	36	169	56	23	99	73	83
10	101	116	108	49	66	103	71	51	73
Min. Value	35	30	15	13	53	23	43	43	73
Max. Value	221	311	271	289	202	306	190	383	283
Percent Variation	631	1037	1807	2223	381	1330	442	891	388
Mean	113	121	89	122	107	123	103	102	141
Std. Dev.	56	81	73	97	56	81	49	101	65

between 252 percent and almost 1,800 percent. The comments that introduce Table 1 (under the subhead Examining the Data) and the questions that are triggered by analyzing Table 2 apply here as well.

Caution: Quick Fixes Create Other Unanticipated Effects

Attempts to *quick fix* (or pursue the *low hanging fruit)* of the measured variation by eliminating the weakest link on the project – the software engineer who writes the most unneeded code – is unlikely to produce the desired results. While such an approach may seem fruitful based on an initial review of the tables above, consider the following data in Table 4 taken from a class where all attendees used the same language.

In the following example, attendee No. 3 had four of the largest of nine possible programs. (These larger-sized programs are shown in italic, bold typeface.) But attendee No. 3 also had the shortest program, Program 7. (Shortest programs are shaded in cells that have attendee identifiers.) Four other attendees (Nos. 1, 2, 6, and 8) also had the largest program to their credit, while six others (Nos. 1, 2, 5, 6, 7, and 8) had the shortest program.

Please note that overall, attendee Nos. 1, 2, 3, 6, and 8 had both at least one largest and at least one smallest program. The weakest link depends on more than merely who writes the largest program. The weakest link also depends on the program that is selected.

Another erroneous argument could be made for the removal (removal may be a little harsh, maybe retrain, reassign, or *promote*) of attendee No. 3 based on the largest number of most lengthy programs. However, the total number of LOC written for the nine programs was higher for attendee Nos. 1, 2, and 4 than for attendee No. 3. The answer to the question of the weakest link becomes less obvious as different quantitative perspectives are considered.

Further examination of the programs from five classes all written with the same language reveals a significant overlap among software engineers that write both shorter and longer programs (see Table 5). The potential for different software engineers to write programs on both ends of the length spectrum suggests that sometimes the apparently more efficient programmer turns out to be the least efficient, and sometimes the apparently least efficient programmer turns out to be the most (judging efficiency by length since each program met the same stated requirements).

Course / Attendee	P1	P2	P3	P4	P5	P6	P7	P8	P9
1 / 1	89	34	67	40	102	235	23	38	168
1 / 3	82	23	33	48	61	34	33	27	52
1 / 4	177	119	67	85	136	276	165	112	233
1 / 5	76	48	305	244	61	121	66	77	127
1 / 7	46	33	17	37	60	95	129	46	186
3 / 5	22	40	100	58	68	131	58	58	102
3 / 6	46	20	30	42	73	82	51	72	82
2 / 7	95	155	147	94	54	191	174	102	218
Min. Value	22	20	17	37	54	34	23	27	52
Max. Value	177	155	305	244	136	276	174	112	233
Percent Variation	805	775	1794	659	252	812	757	415	448
Mean	79	59	96	81	77	146	87	67	146
Std. Dev.	47	50	95	69	28	82	60	30	66

Table 3: *Lines of Code Counts for PSP Classes by Programming Language No. 3*

Variation then should be attributed to context, which includes both the problem space and the engineer's ability to recognize and utilize strengths and features of the software environment to narrow the solution space.

Another source of variance usually attributed to the differences in size of LOC is the process for counting the LOC. In one study shared by Capers Jones, one-third of the participants counted comment lines as a LOC, one-third did not count comment lines, and one-third could not determine if comment lines were included or excluded. As mentioned previously, the attendees of these PSP classes wrote a program that counted LOC. To determine the effects of how each programmer counted their own source sizes, willing attendees shared their line-counting software and their programs so that they could be counted by each others' software.

Each of the line counts in Table 6 (see next page) was calculated from the LOC counting program written by four attendees. The values correspond as follows:

Table 4: *Example of Attendees With Largest and Smallest Programs*

Attendee	P1	P2	P3	P4	P5	P6	P7	P8	P9
1	33	40	30	108	65	***176***	79	107	***284***
2	51	52	24	72	109	166	87	***145***	270
3	***76***	***56***	30	***115***	***175***	158	27	104	128
4	60	52	31	108	94	155	72	94	235
5	22	51	25	50	75	105	47	21	102
6	65	27	***80***	45	95	141	***91***	60	209
7	22	51	25	50	75	105	47	21	102
8	65	27	80	45	95	141	***91***	60	209
Min. Value	22	27	24	45	65	105	27	21	102
Max. Value	76	56	80	115	175	176	91	145	284
Percent Variation	345	207	333	256	269	168	337	690	278
Mean	49	45	41	74	98	143	68	77	192
Std. Dev.	21	12	24	31	34	26	24	44	73

Table 5: *Example of Attendees with Most Lengthy and Shortest Programs*

Class	Number of attendees	Number of attendees with largest program	Number of attendees with smallest program	Number of attendees with the smallest and largest program
1	10	3	6	0
2	8	5	4	2
3	13	7	6	2
4	8	5	6	5
5	10	5	4	1

Attendee No. 1 submitted the values for counting method No. 1, attendee No. 2 submitted the values for counting method No. 2, attendee No. 4 submitted the values for counting method No. 3, and attendee No. 5 submitted the values for counting method No. 4.

For the numbers used in Tables 1-5, please note that in every case, each attendee's submitted LOC values were consistent with the counts provided by others who counted their codes (the shaded rows). While attendee No. 2's software seems to overstate the value of attendee No. 5's sizes, these values were not submitted or included in the numbers used in Tables 1-5. Only the shaded rows below are used in Tables 1-5; that is, only counts submitted by their author are used in the first five tables.

The numbers in Table 6 demonstrate that *variation in counting approaches is not a source of the data variation in this study* because other attendees also counted the subject programs to be of very similar size. Nor did attendees inflate or deflate their own line-count totals, as evidenced by the counts.

Statistical Significance

The apparent differences in the data provoke questions around the statistical relevance of the data. A staff statistician was asked to independently review the data for statistical significance. After conducting a Box-Cox transformation on the data, and performing an analysis of variance, there was a 95 percent probability that the true number of line counts for an individual program from the given population was between 23 and 240 lines. And finally, as is often the case with count data and Poisson distributions, examined variability increased along with size of program.

What is the relevance of the statistical significance? Clearly a 95 percent probability of values that have a range of greater than 10 confirms earlier suspicions that estimating the number of LOC for a given problem is itself highly problematic. While the data in Tables 1-5 evidence this likelihood, the statistical analysis confirms it. A reasonable person, for example, would not procure a computer with such a potential order-of-magnitude variance in performance, cost, or delivery. But unpredictability and variation is the tolerated norm in constructing software.

> *"Because this analysis was conceived and conducted after the classes were conducted, the participants and instructor were unaware that analysis was forthcoming; they themselves were unable to introduce bias into the analysis."*

This norm is evidenced by project performance and by somewhat misdirected attempts at lessons learned and root-cause analyses to identify performance improvements for the future, all dealing with what is likely the wrong problem! The problem itself is often further masked in undocumented overtime and costs, scope containment or reduction, and attempted refinements in estimation variables.

Table 6: *Example From Attendees LOC Counting Program*

Counting Method	Attendee	P1	P2	P3	P4	P5	P6	P7	P8	P9
1	1	91	123	45	121	101	403	553	211	516
1	2	74	97	218	194	279	406	311	181	368
1	4	108	95	205	162	300	484	499	143	706
1	5	193	137	182	229	127	353	353	112	510
2	1	93	133	51	123	107	441	580	213	580
2	2	74	97	218	194	279	406	310	181	368
2	4	110	98	218	317	219	513	523	148	706
2	5	256	172	229	310	170	675	445	122	649
3	1	91	123	45	119	108	380	516	202	479
3	2	74	96	217	194	279	406	310	181	368
3	4	114	78	187	149	303	440	462	130	619
3	5	193	137	181	219	127	517	353	112	510
4	1	91	124	45	120	108	399	548	210	511
4	2	75	98	221	197	282	408	312	182	375
4	4	109	92	202	160	295	476	492	141	672
4	5	193	137	182	209	127	517	353	112	510

Rebuttals Refuted

The data in this article was presented in similar form at conferences and professional meetings. Not too surprisingly, some attendees are quick to defend the widely used LOC for estimating and sizing. Some attendees have doubts that the data applies to their own organization. Despite the rebuttals, each opinion seems to be characterized by one common attribute: no supporting data. The following are some of the most frequently expressed thoughts.

The PSP class is not a good forum for conducting research.

Response: Rarely does an environment exist that controls the requirements and the validation of requirements through the same *control gate* (instructor). Rarely are organizations afforded the opportunity to write the same software 60 times. Rarely are the same programs written in the same language by different authors for comparison. Rarely are the same programs written in different languages for comparison. Rarely are software programs counted using the same counting requirements. And rarely are software programs counted (and cross-counted) by software. Because this analysis was conceived and conducted after the classes were conducted, the participants and instructor were unaware that analysis was forthcoming; they themselves were unable to introduce bias into the analysis. Finding a better environment for conducting LOC sizing is difficult to imagine.

Statistically, the differences between estimates and actual performance average out over time (aka bigger software programs will average out over time).

Response: Apply this principle in other life examples: The buyer of a car with 10 to 15 times the number of typical defects is hardly consoled by the fact that the next buyer may get a vehicle with 10 to 15 times fewer defects than normal. New homeowners will not be comforted that their 2,000-square-foot home was delivered at 100 square feet merely because the purchaser that preceded them received a 25,000-square-foot home; after all, it is merely the luck of the draw. Statistically

the buyers got what they ordered.

A related lesson taught in the PSP course is that granular estimates are more accurate than those developed at a higher level because the *error range* is significantly smaller. For instance, to estimate the time required to build an application applying the error range for the parts (modules, programs, etc.) will provide a more accurate estimate (under similar conditions of knowledge and practice) than an estimate of the application as a whole. This principle, for example, holds true for estimating the size or cost of the rooms of a house, which is a smaller error range than for estimating the house as a whole unit; or for reading the chapters of a book versus reading the book as a whole.

Further, variations in granular estimates tend to offset each other, resulting in an estimate that is closer to actual performance when summed than merely an overall estimate of the time needed to complete the effort. However, a difference exists between the smoothing of variation in *estimates* for a more accurate estimate and the belief that variations in performance (actual) will nullify each other over time. Please note that this lines-of-code analysis was based on actual size variations for the same product; comparisons to estimates were not the subject of this study.

What estimating problem? I'm fine.
Response: This reaction is classic denial when one or more of the following symptoms also exists: project teams that use heroics to complete and deliver a project on time, project teams that use unrecorded overtime to maintain schedule, project teams that use unrecorded resources to complete tasks, projects that are usually late, project teams (not customers) that attempt to renegotiate scope when other project management constraints remain constant, and project deliverables that have unpredictable defects rates compared to projects that predict and manage defects. Admittedly, poor estimating is not the sole source of project delays; team turnover, poor risk management, and true scope changes are additional sources.

The programs' sizes from the course are obviously too small to represent the real world.
Response: Before the introduction of modular programming decades ago, this argument might have had more validity. However, the trend toward modularization, objects, reuse, and architecture-based components challenges the notion that the programs from the PSP course are not in some way representative of much of the software developed today. Certainly the number of LOC that can be peer reviewed in a reasonable two-hour session exceed those represented by many of the programs in the numbers in this study (200 LOC per hour and assuming a two-hour peer review [3]).

Here is what I think …
Response: The information in this analysis is often received with shock, sometimes relief, and sometimes anger. Many who will read this article are likely to say, "Well here's what I think," followed by a statement that reflects the world according to the lenses through which they choose to see reality. In this discussion, more than 60 sets of data were reviewed and more than 500 LOC counts. An appropriate response to doubters is, "Show me your data." The availability of similar data (same requirements, same environment, similar knowledge-base of participants, no inflation/deflation bias introduced because attendees did not know the study would be conducted, same counting techniques, same instructor/exit criteria, and multiple instantiations of the same requirements set) is quite limited.

Do Not Miss the Point

The PSP course provides a rich observatory for gathering data about software productivity. The course itself teaches the student needed principles for estimating, reviewing, defect removal and analysis, scripting, and process improvement. While the PSP course is the source of the data used in this study, this data does not suggest that PSP is the source of the variation in that data; if anything, the practices from the PSP narrow the variations in lines-of-code counts.

This article suggests that using LOC as a measure for actual product delivery has such wide variation as to render the counts practically useless in the best case, harmful and misleading in the worst of cases.

To record lines-of-code data for estimation and calibration of productivity measures seems troubling based on the data.

Conclusion

The purpose of this article is clear: Statistically significant variation in LOC counts render those counts undesirable for estimating and planning, and deceptive as an accurate portrayer of product size. To those left pondering, "What is a better approach for measuring software size?" despite criticisms, function point analysis, endorsed by International Organization for Standardization/International Electrotechnical Commission 20926:2003, is used by thousands of companies worldwide to measure software size. However, function point analysis has its critics as well.

Further understanding of software size for repeatable and quantifiable sizing to improve estimation and project predictability is still needed. The improved collection and use of software size measures will enhance the credibility of software engineers who are plagued with variation in project cost and schedule.◆

Acknowledgement

I gratefully acknowledge the statistical analysis conducted by Laura Halbleib, a technical staff statistician at Sandia National Laboratories, Albuquerque, N.M. Halbleib's contribution validated the intuitive inferences of the analysis by applying rigorous statistical methods. Her insights and knowledge increased the reliability and usefulness of this material.

References

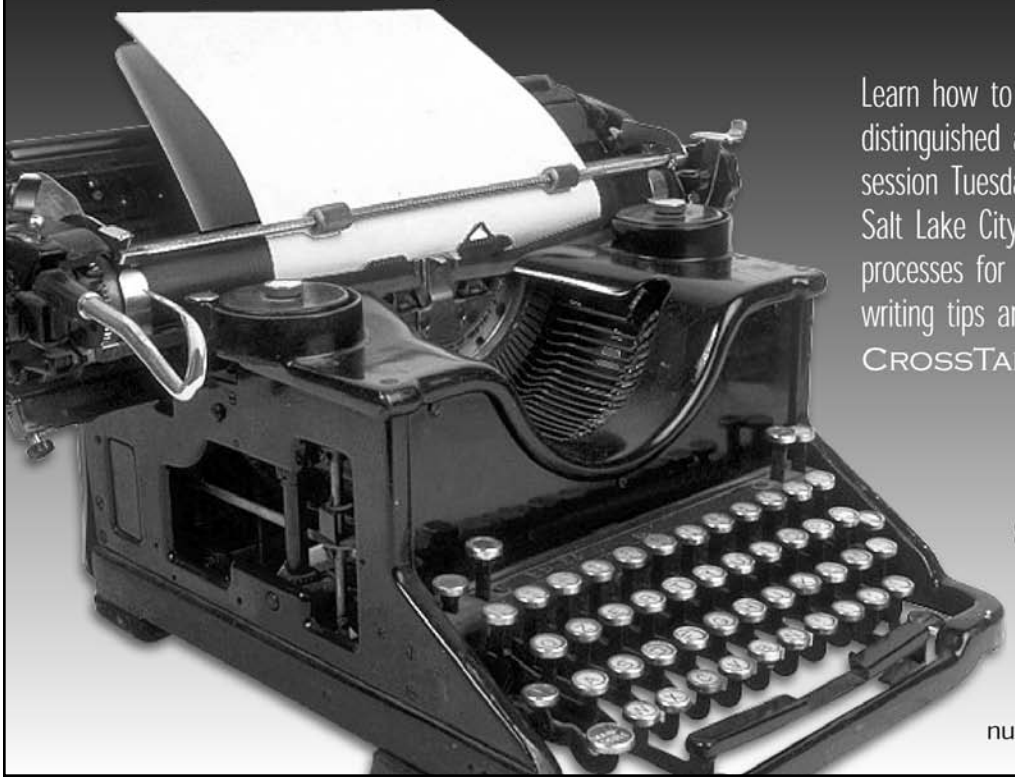
1. Boehm, B. Software Cost Estimation with COCOMO II. Prentice Hall PTR, 2000: 77-81.
2. Jones, C. Software Quality. International Thomson Computer Press, 1997: 333.
3. Humphrey, W. Introduction to the Team Software Process. Addison-Wesley. Dec. 1999.

About the Author

Joe Schofield is a technical staff member at Sandia National Laboratories. He chairs the organization's Software Engineering Process Group, is the Software Quality Assurance Group leader, and is accountable for introducing Personal Software Process℠ and Team Software Process℠ at Sandia. He has dozens of publications and conference presentations. Schofield is active in the local Software Process Improvement Network and has taught graduate-level software engineering classes since 1990.

Sandia National Laboratories
MS 0661
Albuquerque, NM 87185
Phone (505) 844-7977
Fax: (505) 844 2018
E-mail: jrschof@sandia.gov

CrossTalk 101:
Writing for The Journal of Defense Software Engineering



Learn how to become one of CrossTalk's distinguished authors by attending this one-hour session Tuesday, April 19 at the SSTC in Salt Lake City. Discover the benefits of and processes for submitting articles, pick-up writing tips and techniques, and meet the CrossTalk staff.

Tuesday, April 19
5:45 p.m. — 6:45 p.m.
Salt Palace Convention Center
Room 251 A-C

Visit us at our booth, number 608, for more information.

The Joint Services
STC
Systems & Software Technology Conference

"Capabilities: Building, Protecting, Deploying"

18 - 21 April 2005 • Salt Lake City, UT

SSTC is the premier forum in the Department of Defense (DoD) to enhance attendees' professional skills and knowledge of systems and software technologies and policies, enabling them to improve the capabilities they provide to the warfighter.

Don't miss this must attend event
Acquisition Professionals
Program/Project Managers
Programmers
System Developers
Systems Engineers
Process Engineers
Quality and Test Engineers

Design your own stimulating conference experience by choosing from the following sessions and by attending the cutting-edge trade show:

Sponsored/Special Sessions
Government Co-Sponsor Panel Disussion
Industry Panel Discussion
SEI Acquisition
OSD
INCOSE
STSC
CIO
Microsoft
DISA
IEEE
Plenary Sessions
Hands-On Computer Lab
Poster Sessions

Track Topics
Security
Software
Acquisition
Systems
Management/Policy
Process Improvement
Innovations

The Joint Services Systems & Software Technology Conference is co-sponsored by:

United States Army
United States Marine Corps
United States Navy
Department of Navy
United States Air Force
Defense Information Systems Agency

Conference & Exhibit Registration
Now Open - Register Today!
www.stc-online.org
800-538-2663

How Much for the Elephants?

It's a weird profession we have chosen for a living, right? I mean, after all, we work in a profession that considers a millisecond a very long time, and we consider a 128MB USB thumb drive (which, after all, holds almost 100 1.44MB floppies) totally obsolete ($9.95 on sale at a local computer store over Christmas). Ours is a profession where new computer languages come and go yearly, yet COBOL, one of the most commonly used programming languages in the world, still remains a language standardized in 1960 but with its roots embedded in the early 1950s (making it older than me!).

I have been teaching computer science for more than 30 years (first as a teaching assistant in 1974 at the University of Central Florida), starting back when there was barely a discipline known as software engineering[1]. While some things in the field of software engineering come and go, some *truths* need to be relearned by each generation.

1. No programming language ever developed will make it the least bit difficult to write a horrible program[2]!
2. You really can't complete a project until you know the requirements.
3. The first set of requirements is almost never the right requirements.
4. Neither are the second, third, or probably the fourth.
5. The final set of requirements isn't.
6. No matter how good a coder you are, you need a design.
7. Code that is so simple it can't go wrong – will.
8. There is always one error that error-checking routines will miss.
9. No matter what the problem is, it's usually management.
10. No matter how simple it is – you have to test it.
11. Everybody else writes code that needs testing. They say the same about you.
12. Almost any shortcuts you take to speed up the project make it take longer.
13. It's always going to take longer (and cost more) than you plan, even when you take, "It's always going to take longer (and cost more) than you plan" into account.

No getting around it – what we do for a living is hard. No. 13 is particularly difficult. How long does it take? How much will it cost? Even the most experienced developers are often *so* far off with their initial estimates.

Back in college, you never really knew how long a particular programming assignment was going to take. Some that appeared really easy turned out to be really hard (debugging pointers almost *always* involved more work than you thought). And, some jobs that appeared to be really hard turned out to take almost no time at all (the *quick sort* took what, about 10 lines of code?)

The roots of cost (and time) estimation go back a long way. I am reasonably sure that Hannibal, as he was planning to cross the Alps in 219 B.C. during the Second Punic War, was somehow thinking of the incremental cost of each additional elephant. It is interesting to note that the crossing of the Alps with elephants, the event that Hannibal is so famous for, was not really a success. He started out with 34 elephants, but lost many of the elephants on the crossing, and all but one were dead by the end of the Battle of Trebbia[3].

It is also interesting to note that Hannibal, while winning important battles, was beset by political jealousies at home, and this eventually proved his undoing. Because he was unable to get the necessary equipment and personnel, he was not able to take advantage of opportunities and his victories turned into a failure[4]. I could easily draw parallels between Hannibal and many other modern-day software project managers (especially the political jealousies), except for the fact that at around age 70, Hannibal committed suicide rather than face humiliation at the hands of his enemies (we offer early retirement as an option).

Hannibal is famous for the *elephant crossing*, yet the elephants proved to be of limited usefulness during the actual war. What caused Hannibal's eventual downfall was more simplistic – siege equipment (hardware) and people – are basic factors in cost estimation. Would you rather be famous, or succeed? If you want to be famous, see if you can convince 34 elephants to help you code your project in Visual C++. If you would rather succeed, why not have an accurate estimate of costs?

For one final comment on cost estimation, I would like to add that it is obviously a long-standing tradition to mock those whose projects fail due to a lack of cost estimation. In fact, such mockery of those committing cost-estimation failures is reliably documented:

> For which of you, desiring to build a tower, does not first sit down and count the cost, whether he has enough to complete it? Otherwise, when he has laid a foundation, and is not able to finish, all who see it begin to mock him, saying, "This man began to build, and was not able to finish[5]."

When your management suggests that cost estimation has been ordained from on high, you thought they just meant the Pentagon, right?

Hope to see you at SSTC 2005. It's well worth the cost!

— David A. Cook, Ph.D.
Senior Research Scientist
The AEgis Technologies Group, Inc.
dcook@aegistg.com

Can You BACKTALK?

Here is your chance to make your point, even if it is a bit tongue-in-cheek, without your boss censoring your writing. In addition to accepting articles that relate to software engineering for publication in CROSSTALK, we also accept articles for the BACKTALK column. BACKTALK articles should provide a concise, clever, humorous, and insightful perspective on the software engineering profession or industry or a portion of it. Your BACKTALK article should be entertaining and clever or original in concept, design, or delivery. The length should not exceed 750 words.

For a complete author's packet detailing how to submit your BACKTALK article, visit our Web site at <www.stsc.hill.af.mil>.

1. For you purists, the NATO Science Committee sponsored two conferences on software engineering in 1968 and 1969, which many feel gave the field its initial boost. Many also believe these conferences marked the official start of the profession. The term *software engineering* has been used since the late 1950s. See <http://en.wikipedia.org/wiki/History_of_software_engineering>.
2. I make no claim as to the originality of these *truths*. No. 1, for example, comes from "There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs," a quote by Lawrence Flon in "On Research in Structured Programming," SIGPLAN Notices 10:10 (Oct. 1975). This truism is proved again and again as newer and newer languages are developed.
3. See <www.barca.fsnet.co.uk/elephants.htm>.
4. See <www.carpenoctem.tv/military/hannibal.html>.
5. The Bible. Luke 14:28. Revised Standard Version.

DIRECTORATE OF MAINTENANCE

Software

MAS

Engineering

WARNER ROBINS ALC

A CMMI® MATURITY LEVEL 5 ORGANIZATION

For Warner Robins MAS information, please contact:

Dr. Thomas F. Christian Jr.
Chief, Software Engineering Division
Maintenance Directorate

WR-ALC/MAS
280 Byron St. Bldg 230
Robins AFB, GA 31098
478-926-2457 DSN 468-2457
E-Mail: thomas.christian@robins.af.mil

Software Engineering Division
Ogden Air Logistics Center

Co-Sponsored by
U.S. Air Force
Air Logistics Centers
MAS Software Divisions

CrossTalk / MASE
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737