Achievements and Challenges in Cocomo-Based Software Resource Estimation

Barry W. Boehm, *University of Southern California*

Ricardo Valerdi, *Massachusetts Institute of Technology*

A look at the Cocomo suite of models provides an overview of the achievements of software resource estimation over the last 40 years.

Software resource estimation methods and models have had a major impact on successful software engineering practice. They provide milestone budgets and schedules that help projects determine when they are making satisfactory progress and when they need corrective action. They help decision makers analyze software cost-schedule-value trade-offs and make decisions regarding investments, outsourcing, COTS products, and legacy software phaseouts. They help organizations prioritize investments in improving software productivity, quality, and time to market. They are included as essential capabilities in virtually all major software capability maturity models, software engineering textbooks, and software engineering bodies of knowledge.

A counterpart appreciation of their contribution involves what happens to projects that go forward without good estimates of their milestone budgets and schedules. All too frequently, such projects commit to develop too much software within the agreed-on budget and schedule, have no framework to determine whether they are on track, and end up with serious overruns or are terminated. The Standish Group's 2000–2006 Chaos surveys produced data on the percentage of projects that underperformed in certain areas in a particular year (see Table 1).[1]

In light of these results, the importance of software resource estimation is at center stage. This article provides an overview of early and intermediate achievements from the perspective of the Cocomo (Constructive Cost Model) suite of models as well as a look to future challenges. (For more on the relationship between estimation and other software engineering areas, see the related sidebar on p. 76.)

Early achievements: 1965–1985

The earliest achievements in software resource estimation are tied to specific models developed, calibrated, and published as early as 1965. Table 2 lists the most prominent of these models. For background, we explain how some of these models influenced Cocomo's evolution over the last 27 years.

The search for good model forms

The most critical early resource-estimation-modeling issue was to find the right parametric forms for estimating software project effort and schedules. The experiences in analyzing the SDC database convinced people that a purely linear additive model didn't work well. The behavioral phenomenology of software development clearly wasn't consistent with effort estimators combining such factors as size and complexity in linear additive forms.

0740-7459/08/$25.00 © 2008 IEEE

For a while, the best relationships people could find involved estimating effort as a linear function of size, modified by a complexity multiplier. The initial complexity multiplier came from the nonlinear distribution of programming rates in the 169-project SDC sample shown in Figure 1. For example, consider a project developing 10,000 object instructions of software that's deemed more complex than 80 percent of the projects in the SDC sample. For that project, the programming rate would be roughly 7 person-months per 1,000 object instructions. So, the estimated project effort would be $7 \times 10 = 70$ person-months.

Most successful early effort estimation models employed variants of this approach. TRW Wolverton, Boeing Black, and early versions of RCA Price S employed different programming-rate curves for different software classes (scientific versus business versus embedded real time).

By the late '70s, the software community was finding that simple complexity ratings weren't adequate for many software situations that produced different programming rates. Some organizations found that their programming rates were more, rather than less, productive for higher-complexity software because they assigned their best people to the most complex projects. Most important, though, the complexity rating was purely subjective. There was no objective way of determining whether a project was at the 60 or 80 percent level, but going from 80 to 60 percent in Figure 1 will reduce the estimated effort by roughly a factor of two. Some organizations were also finding that their software projects exhibited economies or diseconomies of scale involving estimation relations with exponential functions of size.

Many estimation models were developed in the late '70s. Doty, IBM function points, the Walston-Felix model,[3] and intermediate versions of RCA Price S employed multiple combinations of multiplicative cost drivers. The Bailey-Basili Meta-Model experimented with an additive combination of productivity multipliers and an exponential scale factor. Putnam SLIM developed exponential relationships linking size, productivity, and schedule. Alternative sizing methods such as function points[4] were being developed to support better early size estimation.

Positive and negative experiences with these and other models led to a set of criteria for developing additive, exponential, and multiplicative model factors. The following logic provided the general form for Cocomo in 1981:

$$PM = A \times (Size)^B \times (EM)$$

Table 1

The performance of 8,000 projects in 350 organizations[1]

	2000	2002	2004	2006
Percentage of projects delivered within budget and schedule	28	34	29	35
Percentage of projects cancelled before completion	23	15	18	19
Percentage of projects overrun on budget and schedule	49	51	53	46

Table 2

Prominent software estimation models

Model	Year
SDC (Systems Development Corp.)	1965
TRW Wolverton	1974
Putnam SLIM (*S*oftware *Li*fecycle *M*anagement)	1976
Boeing Black	1977
Doty	1977
IBM-FSD (IBM Federal Systems Division)	1977
RCA Price S	1977
Walston-Felix	1977
IBM function points	1979
Bailey-Basili Meta-Model	1981
Cocomo (Constructive Cost Model)	1981
SoftCost-R	1981
Estimacs	1983
Jensen/SEER (Software Evaluation and Estimation of Resources)	1983
SPQR (Software Productivity Quality and Reliability)/Checkpoint	1985

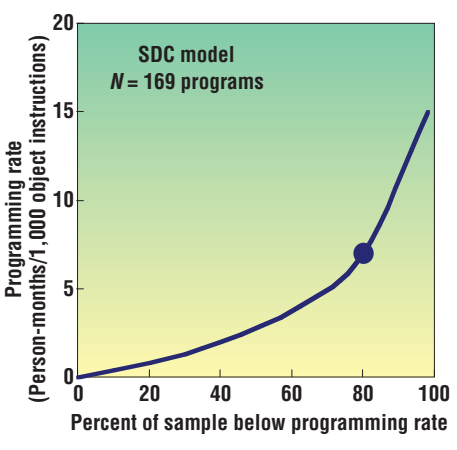

Figure 1. An SDC (Systems Development Corp.) model example.[2] This was the first parametric model demonstrating the nonlinear effects of software size.

Relationship to Other Software Engineering Areas

Progress in software resource estimation is deeply intertwined with progress in other software engineering areas, particularly software architectures, processes, programming, and requirements engineering. Estimation results demonstrating that software costs escalate nonlinearly as utilization of hardware resources approaches 100 percent led to information architectures with significant memory and CPU cycle margins.[1] Software cost-schedule trade-off relations indicating that excessive schedule compression leads to asymptotic cost increases[2] led to more realistic project schedules and research on processes enabling more rapid development.[3,4]

A well-calibrated Cocomo II software cost driver showing that inadequate requirements engineering and architecting leads to disproportionate extra rework focused more emphasis on these activities.[5] The need to determine appropriate milestone endpoints for estimating spiral-process-model costs and schedules led to the development of spiral anchor-point milestones[6] (subsequently used in many projects) and in the Rational Unified Process (RUP).[7–9] On the other hand, we used the RUP work breakdown structure to define the project activities in Cocomo II software cost estimates. Another well-calibrated Cocomo II scale factor, Software Process Maturity, provided the best evidence to date that increased process maturity correlates with increased software productivity. Regression analysis of 161 projects indicated a 4 to 11 percent increase per maturity level, excluding the effect of other factors.[10]

Resource estimation also interacts deeply with software product engineering. Most leading cost and schedule estimation models have parameters relating software project costs and schedules to such product attributes as complexity, required reliability, hardware constraints, database size, and software reuse. These contribute greatly to the key practice of software architecture trade-off analysis. This is a necessary discipline for meeting simultaneous stakeholder needs for high performance, reliability, usability, evolvability, and so on, within project budget and schedule constraints.[11] Additional estimation models in this regard include performance models,[12] reliability models,[13] and real-options models for analyzing investments in software modularity to provide future evolvability options.[14]

Software product engineering and resource estimation are particularly highly coupled in the area of software-product-line reuse. Most books in this area integrate cost and schedule estimation considerations in software domain architecting.[9,15–18] An example success story is Hewlett-Packard's investment in product-line reuse to reduce time to market from 48 to 12 months.[19]

These interactions all come together in the emerging area of value-based software engineering. This involves not just business case analysis in the early stages but also the integration of value considerations into all parts of software engineering, including life-cycle processes, requirements engineering, architecting, development, test, release planning, and project management. Chapters on these topics appear in *Value-Based Software Engineering*.[20]

References

1. B.W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, May 1973, pp. 48–59.
2. L. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," *IEEE Trans. Software Eng.*, vol. 4, no. 4, 1978, pp. 345–361.
3. L. Arthur, *Rapid Evolution Development*, John Wiley & Sons, 1992.
4. S. McConnell, *Rapid Development*, Microsoft Press, 1996.
5. B.W. Boehm et al., *Software Cost Estimation with Cocomo II*, Prentice Hall, 2000.
6. B.W. Boehm and D. Port, "Escaping the Software Tar Pit: Model Clashes and How to Avoid Them," *ACM Software Eng. Notes*, vol. 24, no. 1, 1999, pp. 36–48.
7. W. Royce, *Software Project Management*, Addison-Wesley, 1998.
8. P. Kruchten, *The Rational Unified Process*, 2nd ed., Addison-Wesley, 2001.
9. I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
10. B. Clark, "Quantifying the Effects on Effort of Process Improvement," *IEEE Software*, vol. 17, no. 6, 2000, pp. 65–70.
11. P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures*, Addison-Wesley, 2002.
12. C. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990.
13. J. Musa, *Software Reliability Engineering*, McGraw-Hill, 1999.
14. K. Sullivan et al., "The Structure and Value of Modularity in Software Design," *Joint Proc. European Software Eng./ACM SIGSOFT Foundations of Software Eng. Conf.* (ESEC/FSE 05), 2005.
15. J. Poulin, *Measuring Software Reuse*, Addison-Wesley, 1997.
16. D. Reifer, *Practical Software Reuse*, John Wiley & Sons, 1997.
17. P. Clements and L. Northrop, *Software Product Lines*, Addison-Wesley, 2002.
18. I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse*, Addison-Wesley, 1997.
19. W. Lim, *Managing Software Reuse*, Prentice Hall, 1999.
20. S. Biffl et al., eds., *Value-Based Software Engineering*, Springer, 2005.

where

- *PM* is person-months;
- *A* is the calibration factor;
- *Size* is a software module's functional size, expressed in terms of lines of code, which has an additive effect on software development effort;
- *B* represents the scale factors, which have an exponential effect on software development effort; and
- *EM* represents the effort multipliers, which have a multiplicative effect on software development effort.

The following criteria determine whether a factor is additive, exponential, or multiplicative:

- A factor is additive if it has a local effect on the

included entity (that is, it adds another source instruction).
- A factor is multiplicative or exponential if it has a global effect across the overall system being estimated (that is, it adds a security requirement for the entire system).
- A factor is exponential if it's more influential for larger projects than for smaller projects, often because of the amount of rework due to architecture and risk resolution, team compatibility, or readiness for system-of-systems integration.[5]

We applied these criteria to the development of the Cocomo and associated models. In order to ensure relevance, these models' assumptions about the cost-estimating relationships require validation by historical projects.

Development of model evaluation criteria

Models are frequently evaluated for their ability to estimate software development. To evaluate the Cocomo model's utility for practical estimation, we used these criteria:[2]

- *Definition*. Has the model clearly defined the costs it's estimating and the costs it's excluding?
- *Fidelity*. Are the estimates close to the actual costs expended on the projects?
- *Scope*. Does the model cover the class of software projects whose costs you need to estimate?
- *Objectivity*. Does the model avoid allocating most of the software cost variance to poorly calibrated subjective factors (such as complexity or personnel factors)? That is, is it difficult to jigger the model to obtain any results you want?
- *Constructiveness*. Can a user tell why the model gives the estimates it does? Does it help the user understand the software job to be done?
- *Detail*. Does the model easily accommodate the estimation of a software system consisting of a number of subsystems and units? Does it give (accurate) phase and activity breakdowns?
- *Stability*. Do small differences in inputs produce small differences in output cost estimates?
- *Ease of use*. Are the model inputs and options easy to understand and specify?
- *Prospectiveness*. Does the model avoid using information that won't be known until the project is complete?
- *Parsimony*. Does the model avoid using highly redundant factors or factors that make no appreciable contribution to the results?

For the most part, each of these criteria's significance is reasonably self-evident. From a generic standpoint, the criteria have also proven helpful in the development and evaluation of other cost estimation models in the Cocomo suite and elsewhere.

Emergence of a model marketplace and community of interest

The early '80s marked the development of a software resource estimation community of interest, including conferences, journals, and books. This helped socialize both the addressing of the issues we just discussed and the emergence of several estimation models that passed both usage tests and tests of market viability. These models included refined versions of earlier models such as RCA Price S and Putnam SLIM and new models such as SPQR/Checkpoint, Estimacs, Jensen/SEER, Softcost-R, and Cocomo and its commercial implementations such as PCOC (Personal Computer Cocomo), Gecomo (General Electric [UK] Cocomo), Costar (Cocomo Star), and Before You Leap. These models were highly effective for the largely waterfall-model, build-from-scratch software projects of the '80s but began to encounter new classes of challenges, as we discuss next.

Intermediate achievements: 1985–2005

We can divide these two decades roughly into two equal periods: one of mainstream refinements and one of proliferation of software development styles.

1985–1995: Mainstream refinements

This period primarily involved proprietors of the leading cost models addressing problems that users brought up in the context of their existing mainstream capabilities. Good examples are the risk analyzers, either Monte Carlo based or agent based, and the breakdown of overall cost and schedule estimates by phase, activity, or increment.

The most significant extensions during this period were in software sizing. Under the prospectiveness criterion, accurate early estimation of SLOC is a major challenge. Some comparison-oriented methods involving paired size comparisons, ranking methods, and degree-of-difference comparisons were developed. They were helpful, but their performance was spotty and expert-dependent.

During 1985–1995, the function-point community made a major step forward in defining uniform counting rules for its key size elements of inputs, outputs, queries, internal files, and external interfaces, along with associated training and certification capabilities.[6] This made function points a good

An important criterion for model evaluation is whether the model avoids using highly redundant factors or factors that make no appreciable contribution to the results.

Alternative Model Forms

Researchers have been concurrently exploring alternative software resource estimation model forms. Using metadata about a project (size, type, process, domain, and so on), analogy or case-based estimation produces estimates of the project's required resources on the basis of the resources required for the most similar projects in a large database.[1,2] One such database is the International Software Benchmarking Standards Group (ISBSG) database of 3,000 software projects.[3] In one study, analogy-based estimation performed better than alternative estimation methods in 60 percent of the cases but performed worst in 30 percent of the cases,[4] indicating some promise but need of further refinement.

Neural-net models use layouts of simulated neurons and training algorithms to adjust neuron connection parameters to learn the best fit between input parameters and values to be estimated. Implementations of these models have produced accuracies of plus or minus 10 percent error,[5] but in many cases, estimation of projects outside the training set has been much less accurate. Under the constructiveness criterion (see the section "Development of model evaluation criteria" in the main article), these models do not provide constructive insights on the software job to be done. Researchers have recently used other machine-learning techniques to successfully determine reduced-parameter versions of parametric cost models.[6]

Systems dynamics models integrate systems of differential equations to determine the flow of effort, defects, or other quantities through a process as a function of time. They are good for understanding the effects of dynamic relations among software development subprocesses, such as the conditions under which Brooks' law holds (adding more people to a late software project will make it later).[7] Pioneering research in this area has been done for general software project relationships[8] and for interactions among effort, schedule, and effect density[9] in performing software inspections.

Each of these model forms provides complementary perspectives to those of parametric models. Challenges for the future include finding better ways to integrate their contributions.

References

1. T. Mukhopadhyay, S. Vincinanza, and M. Prietula, "Examining the Feasibility of a Case-Based Reasoning Model for Software Effort Estimation," *MIS Quarterly*, June 1992, pp. 155–171.
2. M. Shepperd and C. Schofield, "Estimating Software Project Effort Using Analogies," *IEEE Trans. Software Eng.*, vol. 23, no. 11, 1997, pp. 736–743.
3. *Data R8*, Int'l Software Benchmarking Standards Group, 2005.
4. M. Ruhe, R. Jeffery, and I. Wieczorek, "Cost Estimation for Web Applications," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 03), IEEE CS Press, 2003, p. 285.
5. G. Wittig, *Estimating Software Development Effort with Connectionist Models*, paper 33/95, Dept. of Information Systems, Monash Univ., 1995.
6. T. Menzies et al., "Selecting Best Practices for Effort Estimation," *IEEE Trans. Software Eng.*, vol. 32, no. 11, 2006, pp. 883–895.
7. A. Endres and D. Rombach, *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*, Addison-Wesley, 2003.
8. T. Abdel-Hamid and S. Madnick, *Software Project Dynamics*, Prentice Hall, 1991.
9. R. Madachy, *Software Process Dynamics*, Wiley-IEEE Press, 2008.

match for business applications, which tend to have simple internal business logic. However, this approach was not as good for scientific and real-time control applications with more complex internals.

1995–2005: Proliferation of development styles

The development of Cocomo II, starting in 1995, was based primarily on the realization that the 1981 Cocomo model's assumptions of sequential waterfall-model development, three stratified development modes, and occasional software reuse with linear savings were becoming obsolete. We projected software applications development classes (end-user programming, application generators, application composition, system integration, and infrastructure) out to 2005 and developed Cocomo II to address them.[7] This involved

- developing new anchor-point milestones to serve as the endpoints for concurrent spiral-model cost and schedule estimates;
- developing a more realistic nonlinear reuse model;
- adding exponential scale factors for such scalability controllables as process maturity and architecture/risk resolution;
- adding new cost drivers for such phenomena as development for reuse, distributed software development, and personnel continuity;
- dropping obsolete cost drivers such as turnaround time; and
- enabling the use of alternative early sizing methods such as function points.

(For a discussion of alternative model forms, see the related sidebar.)

Each of these changes was supported by valuable research results and by our experiences in trying to tailor the original Cocomo to new situations. We based the nonlinear effects of software reuse on research at the NASA Software Engineering Laboratory,[8] maintenance-effort distributions,[9] and nonlinear software integration effects.[10] Anchor-point milestones and their phase distributions were supported by research at Rational and AT&T and by spiral-model use at TRW.[11,12] Process maturity characterization was supported by collaboration with the

Carnegie Mellon Software Engineering Institute and its associated definitions and data on productivity effects.[13] Estimation of the relative cost of writing for reuse was supported by software reuse experiences. We also addressed exponential-diseconomies-of-scale effects in software development.[14]

We successfully calibrated the resulting Cocomo II model to 161 carefully collected and verified project data points. Its predictions within this sample were within 30 percent of the actuals 75 percent of the time for effort (80 percent with local calibration) and 64 percent of the time for schedule (75 percent with local calibration). It has been broadly incorporated into several commercial cost estimation models.

However, although Cocomo II does a good job for the 2005 development styles projected in 1995, it doesn't cover several newer development styles well. This led us to develop additional Cocomo II-related models such as

- the Chinese government version of Cocomo (Cogomo),
- Constructive Incremental Cocomo (Coincomo),
- the Constructive Quality Model (Coqualmo),
- Orthogonal Defect Classification (ODC) Coqualmo,
- Information Dependability Attribute Value Estimation (iDAVE),
- the Constructive Product Line Investment Model (Coplimo),
- the Constructive Productivity-Improvement Model (Copromo),
- the Constructive Phased Schedule and Effort Model (Copsemo),
- the Constructive Rapid Application Development Model (Coradmo),
- the Constructive Security Cost Model (Cosecmo),
- the Constructive Commercial-off-the-Shelf Cost Model (Cocots),
- the Constructive Systems Engineering Cost Model (Cosysmo), and
- the Constructive System-of-Systems Integration Model (Cososimo).

(For more information on these models, visit the Center for Systems and Software Engineering, http://csse.usc.edu.)

The maturity of the models in the Cocomo suite varies depending on the amount of validation obtained through expert opinion and historical data, as indicated in Figure 2. For more information on the model development methodology, see the "Coping with Future Trends" sidebar (see p. 80).

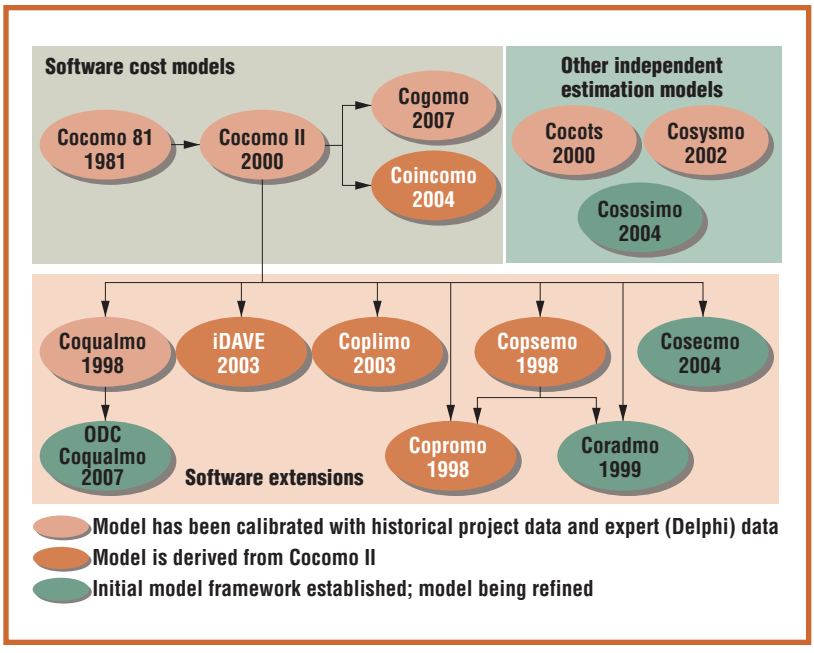

Figure 2. The Cocomo suite of models. Dates indicate the time that the first paper was published for the model.

Counterpart commercial software-cost-model companies, such as CostXpert, Galorath (SEER), Price Systems (Price S), and Softstar Systems (Costar), have become USC Center for Systems and Software Engineering (CSSE) Affiliates. As such, they have participated in the research on these model extensions and are developing counterpart extensions to their commercial offerings. This enables the software resource estimation field to share expertise while offering users a range of solutions.

Future challenges and responses

In the '80s, our vision of the future was that those in the software estimation field were like Tycho Brahe in the 16th century, compiling observational data that later Keplers and Newtons would use to develop a quantitative science of software engineering.[2] As we went from unprecedented to precedented software applications, we thought, our productivity would increase and our error in estimating software costs would continue to decrease (see Figure 3 on p. 82).

However, this view assumed that, like the stars, planets, and satellites, software projects would continue to behave in the same way as time went on. But, as we have repeatedly seen, this assumption was invalid. The software field is continually being reinvented via structured methods, abstract data types, information hiding, reusable components, commercial packages, very-high-level languages, rapid-application-development processes, model-driven

Coping with Future Trends

The need for specialized modeling of particular software engineering phenomena led to frameworks for developing additional cost models. This trend in turn led to the eight-step methodology (see Figure A) we successfully used to develop the Cocomo II family of models.

A case study of the Constructive Commercial-off-the-Shelf Cost Model (Cocots) model illustrates the eight steps. The model's development began in 1995 with a University of Southern California Center for Systems and Software Engineering (CSSE) Affiliates workshop on processes and architectures for the development and evolution of COTS-based systems. A breakout group at the workshop identified model needs (Step 1) and sources of further information for Step 2. This led to Christopher Abts's hypothesis that a good estimation model would consist of

- a Cocomo II-like model using source lines of COTS-integration glue code as a sizing parameter;
- some Cocomo II effort multipliers such as personnel capability, continuity, and experience; and

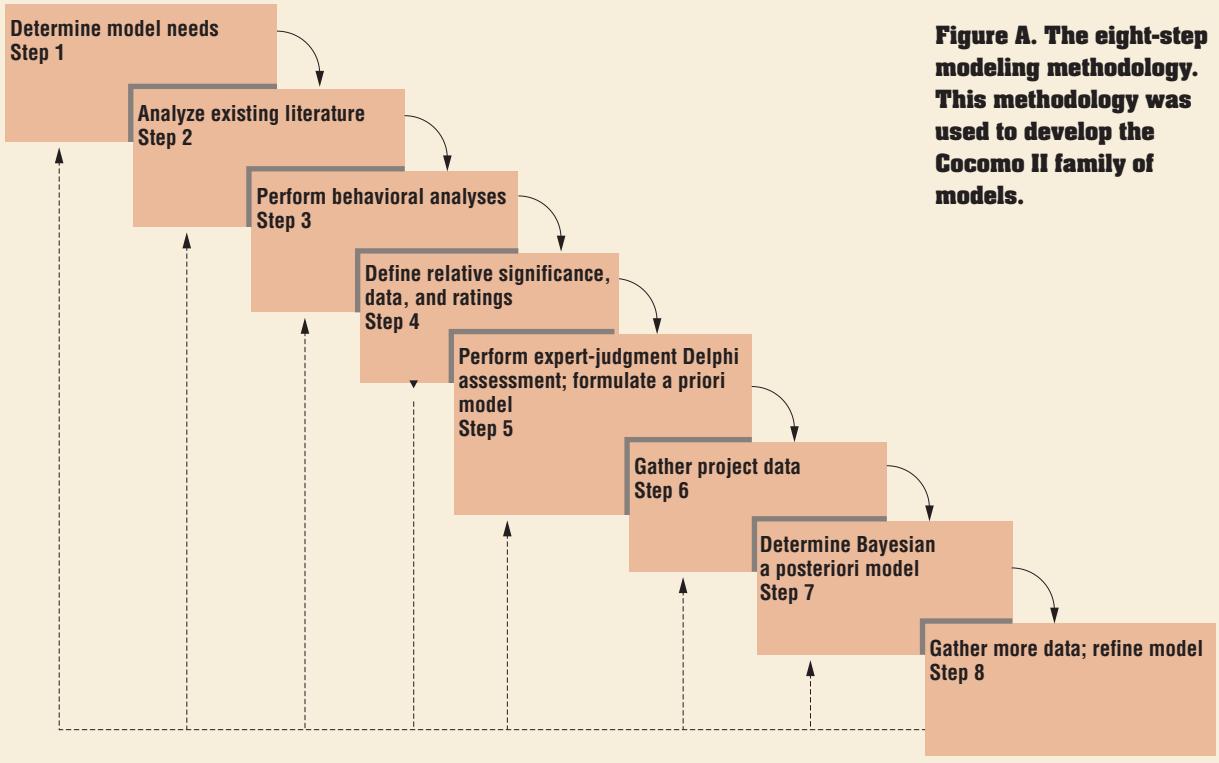

Figure A. The eight-step modeling methodology. This methodology was used to develop the Cocomo II family of models.

development, agile methods, autonomic software, and so on.

So, at point A in Figure 3, increased domain understanding led to the ability to develop and reuse software components. This boosted productivity but increased the estimation error of existing resource estimation models, until model refinements for software reuse were developed and calibrated.

With each reinvention (points B, C, and D in Figure 3), software cost estimation and other software engineering fields must also reinvent themselves just to keep up, resulting in the type of progress shown in the right side of Figure 3. The most encouraging thing in Figure 3, though, is that leading and new companies can build on this experience and continue to increase our relative productivity in delivering the huge masses of software our society increasingly depends on. Our biggest challenges are to figure out how to selectively prune the parts of the software engineering experience base that become less relevant, and to conserve and build on the parts with lasting value for the future.

Despite the challenges in software resource estimation, major technical achievements have enabled the development of sophisticated estimation models. The following achievements are among the most significant:

- some COTS-specific effort multipliers such as COTS product maturity, vendor support, interface complexity, and performance limitations.

A behavioral analysis of the candidate cost drivers was performed (Step 3) and was iterated with the affiliates at two follow-up workshops. These workshops also determined the cost drivers' relative significance and motivated the model developers to drop and combine some of the drivers. The remainder of Step 4 involved development, review, and iteration of the cost driver rating scales and model data definitions, including definition of the relationships between Cocots estimates and Cocomo II estimates. These enabled the execution of the Delphi process with affiliate and USC-CSSE experts (Step 5), and the beginning of Step 6 with the initial collection of two affiliate pilot project data points (which identified further data definition clarifications needed).

The two affiliate data points appeared compatible with the model. However, at this point, we completed six well-instrumented COTS-based applications as part of our series of campus e-services team project applications. Four of these didn't fit the model well. In analyzing the data and interviewing the developers, we found that these projects were spending much effort in COTS assessment and tailoring, none of which generated glue code.

This caused Abts to revise his hypothesis and to go back to Step 3 to perform behavioral analyses and develop forms for estimating COTS assessment and tailoring. The resulting revised model, with significant support from the US Federal Aviation Administration, the US Office of Naval Research, the US Air Force Electronic Systems Center, and the USC-CSSE Affiliates, achieved a reasonably accurate set of estimates across a 20-project set of project data points.[1]

The data collection and analysis also provided us with further insights on the critical success factors and processes of COTS-based application development. Using the principle of "process happens where the effort happens," we were able to develop, apply, and validate a set of composable process elements for COTS-based applications development.[2]

Step 7 involved integrating inputs from experts and statistical data analysis to produce a calibrated model. We did this using the Bayesian approach, which allows the combination of expert opinion from Step 5 and empirical data from Step 6. Multiple-regression analysis of project data points produced outcome-influenced values. For Cocomo II, 161 data points produced mostly statistically significant parameter values. The Bayesian approach favors experts when they agree or historical data where results are significant. The Bayesian version of Cocomo II performed considerably better than the pure data-regression version in estimating the 161 projects in the Cocomo II database. This shows that including expert judgment produced a more robust model by avoiding too much chasing of noisy data or outliers.

The eight-step methodology has also been used on such Cocomo II extensions as the Constructive Quality Model (Coqualmo), the Constructive Systems Engineering Cost Model (Cosysmo),[3] and an elaboration of the Cocomo II Tools effort multiplier. Further uses of this methodology are underway for Cocomo II extensions addressing computer security cost increases and software-intensive systems-of-systems integration. The process has enabled successful responses to the challenges of developing resource estimation models for new software development styles and objectives.

References

1. C. Abts, "A Cost Estimation Model for COTS Integration," PhD dissertation, Industrial and Systems Eng. Dept., Univ. of Southern Calif., 2004.
2. Y. Yang et al., "Value-Based Processes for COTS-Based Applications," *IEEE Software*, vol. 22, no. 4, 2005, pp. 54–62.
3. R. Valerdi, *Systems Engineering Cost Estimation with Cosysmo*, John Wiley & Sons, 2008.

- *Appropriate functional forms for estimation models*—determining which parameters contribute in additive, multiplicative, exponential, and asymptotic ways.
- *Statistically significant model calibration*—obtaining critical masses of carefully defined, multiparameter project data that produce robust, statistically significant parameter values.
- *Bayesian combination of expert judgment and statistical data analysis*—providing the ability to bootstrap model usage and accumulation of critical masses of project data.
- *Model reinvention to accommodate new development paradigms*—developing anchor-point milestones enabling not only principled estimation but also controllable concurrent engineering for spiral and evolutionary software processes.[15] This achievement includes models for rapid development, reuse, product lines, and COTS integration.
- *New sizing parameters*, including function points, object-oriented metrics, and specification-based sizing parameters.
- *Methodologies for development, calibration, and evolution of new models*, including the multistep process for exploration, analysis, definition, calibration, and refinement of parametric estimation models and techniques for determining viable reduced-parameter models in special domains.

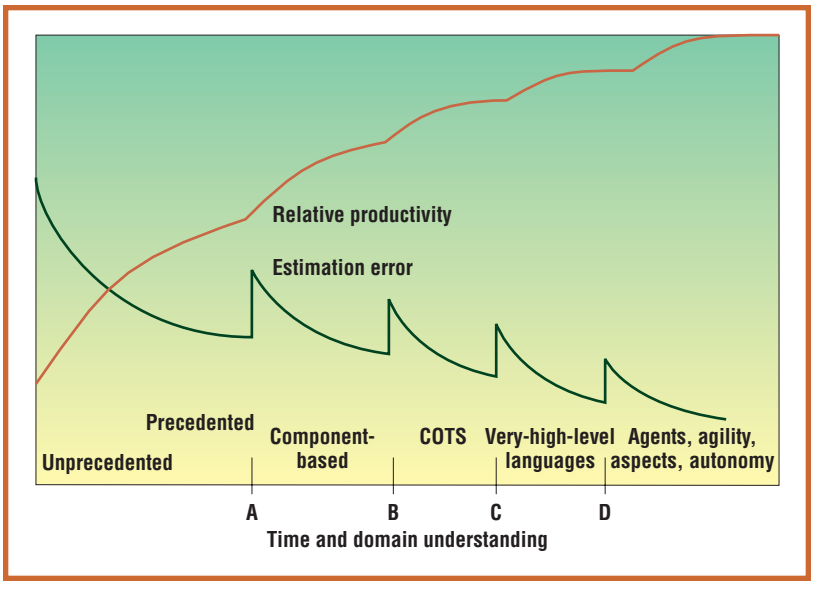

Figure 3. Software estimation—the receding horizon. At point A, increased domain understanding led to the ability to develop and reuse software components. Points B, C, and D indicate other points where software development was essentially reinvented.

- *Contributions to value-based software engineering*—integration of resource investment levels and benefits estimation models into return-on-investment models.

These achievements have also impacted the management of software engineering. Practitioners have benefited from them in these areas:

- *The basis of project stakeholder negotiation and expectations management*. This includes the ability to avoid overcommitment to infeasible budgets and schedules.
- *The basis of project planning and control, and impact on processes*. Anchor-point milestones enable control of complex concurrent engineering processes. Schedule, cost, and quality as independent-variable processes enable meeting targets by prioritizing and adding or dropping marginal-priority features.
- *Improved project performance*. Phase and activity estimates provide a framework for better progress monitoring and control.
- *A framework for process improvement*. This includes improved planning realism; monitoring and control; models; and productivity, cycle time, quality, and business value.
- *Contributions to communities of interest*. Besides the core estimation community, these include the communities concerned with empirical methods, metrics, economics-driven or value-based software engineering, systems architecting, software processes, and project management.

Given that the software engineering field is continually reinventing itself, it is evident that software resource estimation is not a solved problem. Because we expect software engineering to continue changing, future challenges will introduce new opportunities for improved methods and tools. Here are the most significant challenges:

- *Integration of software- and systems-engineering estimation*. Challenges include compatible sizing parameters, schedule estimation, and compatible output estimates.
- *Sizing for new product forms*. These include requirements or architectural specifications, stories, and component-based development sizing.
- *Exploration of new model forms*. Candidates include case-based or analogy-based estimation; neural nets; system dynamics; and new sizing, complexity, reuse, or volatility parameters.
- *Maintaining compatibility across multiple classes of models*. Challenges include compatibility of inputs, outputs, and assumptions.
- *Total-cost-of-ownership estimation*. In addition to software development, this can include estimation of costs of installation, training, services, equipment, COTS licenses, facilities, operations, maintenance, and disposal.
- *Benefits and return-on-investment estimation*. This can include valuation of products, services, and less-quantifiable returns such as customer satisfaction, controllability, and staff morale.
- *Accommodating future software engineering trends*. These can include ultralarge software-intensive systems, ultrahigh dependability, increasingly rapid change, massively distributed and concurrent development, and the effects of computational plenty, autonomy, and biocomputing.

These trends contribute to the ever-receding horizon of perfectible resource estimation models but keep the model development and evolution community in a highly stimulating and challenge-driven state.

References

1. *Chaos Report*, Standish Group Int'l, 2007.
2. B.W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.
3. C.E. Walston and C.P. Felix, "A Method of Programming Measurement and Estimation," *IBM Systems J.*, vol. 16, no. 1, 1977, pp. 54–73.
4. A.J. Albrecht and J. Gaffney, "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Trans. Software Eng.*, vol. SE-9, no. 6, 1983, pp. 639–648.
5. J. Lane and R. Valerdi, "Synthesizing System-of-Systems Concepts for Use in Cost Estimation," *Systems Eng.*, Dec. 2007, pp. 297–308.

6. H. Rehesaar, "ISO/IEC Functional Size Measurement Standards," *Proc. GUFPI/IFPUG Conf. Software Measurement and Management*, Int'l Function Point Users Group, 1996, pp. 311–318.
7. B.W. Boehm et al., *Software Cost Estimation with Cocomo II*, Prentice Hall, 2000.
8. R. Selby, "Empirically Analyzing Software Reuse in a Production Environment," *Software Reuse: Emerging Technology*, W. Tracz, ed., IEEE CS Press, 1988, pp. 176–189.
9. G. Parikh and N. Zvegintzov, "The World of Software Maintenance," *Tutorial on Software Maintenance*, IEEE CS Press, 1993, pp. 1–3.
10. R. Gerlich and U. Denskat, "A Cost Estimation Model for Maintenance and High Reuse," *Proc. European Software Cost Modeling Meeting* (ESCOM 94), 1994.
11. W. Royce, *Software Project Management*, Addison-Wesley, 1998.
12. J. Marenzano, "System Architecture Review Findings," *Proc. 17th Int'l Conf. Software Eng. Architecture Workshop*, Carnegie Mellon Univ., 1995.
13. W. Hayes and D. Zubrow, *Moving on Up: Data and Experience Doing CMM-Based Process Improvement*, tech. report CMU/SEI-95-TR-008, Software Eng. Inst., Carnegie Mellon Univ., 1995.
14. R. Banker, H. Chang, and C. Kemerer, "Evidence on Economies of Scale in Software Development," *Information and Software Technology*, vol. 36, no. 5, 1994, pp. 275–282.
15. B.W. Boehm and W. Hansen, "The Spiral Model as a Tool for Evolutionary Acquisition," *CrossTalk*, vol. 14, no. 5, 2001, pp. 4–11.

About the Authors

Barry W. Boehm is the TRW Professor of Software Engineering in the University of Southern California's Computer Science and Industrial and Systems Engineering Departments. He's also the director of the university's Center for Systems and Software Engineering. Boehm received his PhD in mathematics from the University of California, Los Angeles. Contact him at boehm@usc.edu.

Ricardo Valerdi is a research associate at the Massachusetts Institute of Technology's Lean Advancement Initiative and the Systems Engineering Advancement Research Initiative. Valerdi received his PhD in industrial and systems engineering from the University of Southern California. Contact him at rvalerdi@mit.edu.