focus guest editors' introduction

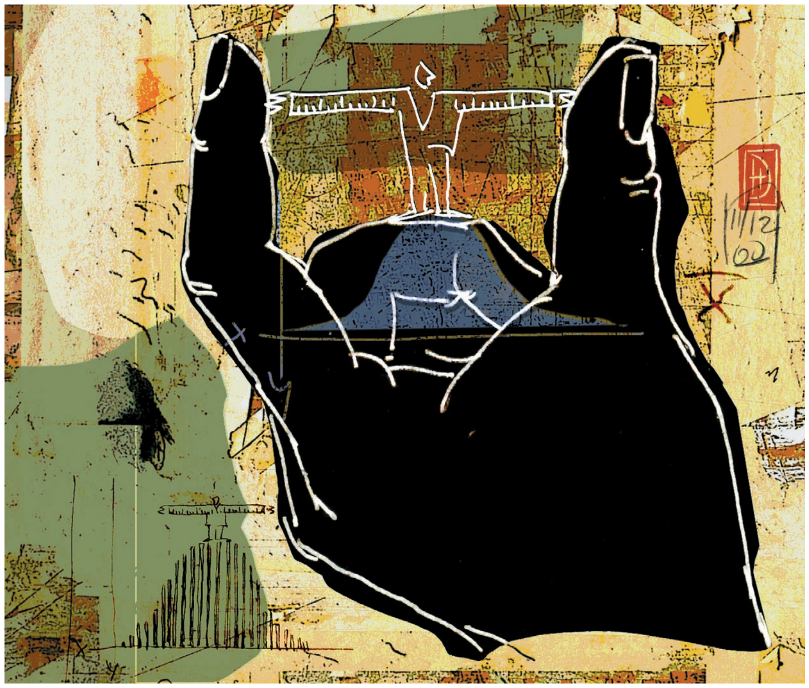
# Software Estimation Perspectives

**Barry W. Boehm,** *University of Southern California*
**Richard E. Fairley,** *Oregon Graduate Institute*

How much is 68 + 73? Engineer: "It's 141." Short and sweet.

Mathematician: "68 + 73 = 73 + 68 by the commutative law of addition." True, but not very helpful.

Accountant: "Normally it's 141, but what are you going to use it for?"



Often, the accountant's answer is the most helpful of all. You may have been looking for the answer to the following question: "The Inspections people estimate that they can remove 68% of my software defects; the Test people estimate that they can remove 73%. How many will they remove together?" Clearly, 141% is not a good answer; there must be some defects that are being counted twice.

This story brings out two important points about software estimation:

1. It's best to understand the background of an estimate before you use it.
2. It's best to orient your estimation approach to the use that you're going to make of the estimate.

Grant Rule's essay on "Bees and the Art of Estimating" (see the sidebar) illustrates point 1 well. Jumping into an estimate of the

0740-7459/00/$10.00 © 2000 IEEE

# Bees and the Art of Estimating

**by Grant Rule**

Have you heard the one about the bees? Are there any apiarists amongst *IEEE Software*'s readership? Well never mind. Even if you're not a beekeeper and absolutely hate bread and honey, you can still participate in this simple experiment.

I've run this trial on numerous courses and at several conferences in the US, the Netherlands, Germany, Scandinavia, and the UK—and each time, I get pretty similar results.

Try it now. Take paper and pencil (or use your handheld, if you're a gadget freak) and write your estimate for the number of insects in an average hive of English honeybees. No cheating now!

Are you done? Oh, you don't know anything about bees? Well, this is an estimating task. Estimating is about predicting in the face of uncertainty and incomplete knowledge, so that's no bar. We estimate for unfamiliar software projects every day. You don't bother? Oh! How ever do you prioritize your set of possible tasks (that's a rhetorical question)? What do I mean by "average"? Now, that's a good question. Do I mean the "mean population of bees in a hive over time, taking a typical annual cycle into account"? Or do I mean "the mean, calculated from the total population of bees in England at a specific time, divided by the number of hives in England"? And anyway, what do I mean by "insect"? Should you include critters such as mites, pillbugs, and wasps that live on, in, or among the bees? Hang on! Bees are hive animals; arguably the queen and her workers and drones constitute a "single genetic individual," so should you just count the number of queens? And if the hive is on the point of swarming (May or June maybe), should you count the queens that leave to find a new hive?

## Habits Are Hard to Break

The point, of course, has nothing to do with bees. It is about the habits and practice of estimating.

When I've run this trial with a "live" audience, very few people have questioned the implicit assumptions (actually 10 from a total of 1,100 ± 10%). Subsequent discussion, and daily observation, leads me to suppose that the same is true when those people estimate for software projects. By the way, I forgot to tell you that all the "live" trials were with groups of software measurement or improvement enthusiasts. The absence of documented assumptions, or even any consideration of the assumptions, pretty much makes an estimate unrepeatable and probably worthless. Especially if you believe, as I do, that each software project is a "complex system with sensitive dependence on initial conditions" (but that's a topic for another time).

To return to the bees. It seems that the population of worker honeybees in a hive fluctuates during the course of the seasons and with the weather and depending on the food supply. Taking any populated hive at random, there is a 95% probability that it will contain a population in the range of 20,000 to 60,000 workers and drones. Add one for the queen if you like.

The surprising fact, to me, is that during the course of my trials, only about eight people have ever given a range estimate (that is, less than 1% of participants). Most go for precise answers, with a high probability of being incorrect. I would forecast that 80–90% of those of you who made an attempt at an answer wrote down a single number (95% confidence). I'm assuming that *Software*'s readership, with an engineering background, is 10 to 20 times more likely to give a range estimate than a point estimate.

It seems to be a built-in human characteristic to prefer precision over accuracy. By this I mean that humans prefer a specific, single value that pretends to certainty (and which the estimator "knows" is probably wrong) over a range value that almost certainly includes the most probable value and hence is correct, but which retains some level of uncertainty. As there are very few certainties in this life, this proclivity is—just maybe—a contributing factor to the well-reported high percentage of failed software projects.

## Who's an Expert?

The most common basis used for estimating is so-called "expert opinion." That is, project managers make estimates by referring to their past project experiences. But few can produce any historic data when challenged, and typically there is very little rigor in the approach. No assumptions are documented, no baseline measurements made, and often no provision exists for feedback. Few organizations consistently hold post-implementation reviews, and the project manager who makes personal records of the accuracy of estimates seems a rarity.

Estimating is more craft than science. Good craft work requires lots of practice. As a degree of uncertainty can be guaranteed, the method used to deal with this uncertainty is crucial to ensure repeatability and accuracy and to build confidence in one's ability as an estimator. The specific calibrations, and even the details of the algorithms and tools used, are secondary. After all, once you can repeatably estimate the range in which an ultimate result will fall, even though the range is wide, it is only a matter of tweaking the calibrations to narrow the range and improve the precision of your estimating. That's a matter of comparing the estimate with the result, adjusting the calibrations, then doing it all again. Over and over. Just as athletes do when they practice.

The key is repetition and practice. It's very difficult to produce reliable estimates if you rarely exercise the craft, say once at the beginning of each project (maybe once in nine months?), or if the feedback loop between estimate and result is too long to facilitate recalibration. If you can't estimate in the small, then don't estimate in the large. It's too expensive.

Good estimators practice on inconsequential topics, so they develop the habits of estimating. They do so all the time, on various kinds of estimate. For example, How long will this journey take? What is the percentage of home-built versus imported cars in town? How many emails will I receive tomorrow? What effort is needed to produce each use case? How much effort will I "waste" on unplanned work tomorrow? Even when it is not their responsibility, good estimators challenge themselves to produce an estimate, for their own benefit if not for others. And then they improve over time.

Or, on the other hand, you might just think we are very sad people.

## About the Author

**Grant Rule**, a founder of the European consultancy Software Measurement Services Ltd., helps organizations improve their estimating, risk, and requirements processes through benchmarking and CMM-based services; PG_Rule@compuserve.com; www.software-measurement.com.

number of bees in a hive without understanding the counting rules can produce a pretty useless estimate. The same is true for software: Does an estimate of 100 person-months for "software development" include analysis and design, integration and test, deployment, management, or uncompensated overtime? If you use the estimate without knowing the answers, you can get yourself into serious trouble.

**You can often get both better and simpler estimates if you keep the use of your estimate in mind.**

Point 2 highlights the fact that estimates have a number of uses, and you can often get both better and simpler estimates if you keep the use of your estimate in mind. For example, suppose you are doing estimates for a make-or-buy analysis. The vendors' quotes for the "buy" option are clustering around $100K for satisfactory-looking products, and it is looking like the "make" option will be a good deal more expensive. In this case, you can make a simpler and even stronger estimate for the "make" option by using optimistic assumptions about its size, complexity, and staff capability. If even the resulting optimistic "make" estimate comes out at $130,000, you have both saved yourself a good deal of estimating effort and produced a stronger conclusion that the "buy" option is better than even the best-case "make" option.

## Using Your Estimates

Besides setting budgets and schedules and supporting make-or-buy analyses, software estimation techniques can have several additional decision support uses:

- supporting negotiations or tradeoff analyses among software cost, schedule, quality, performance, and functionality;
- providing the cost portion of a cost–benefit or return-on-investment analysis;
- supporting software cost and schedule risk analyses and risk management decisions; and
- supporting software quality or productivity improvement investment decisions.

For the latter, for example, the Cocomo II productivity ranges shown in Figure 7 of Bradford Clark's article provide the basis for analyzing various mixed strategies of investments in personnel capability improvement, process maturity, software reuse, software tools, and multisite software development support.

Another observation in Grant Rule's essay about estimation perspectives involves the dynamic nature of the software field. One perspective is that software projects will necessarily evolve during development and that up-front estimates cannot be precise. Several software cost and schedule estimation models now provide (optimistic, most likely, and pessimistic) estimation ranges rather than point estimates.

These ranges support entirely new process models better attuned to the dynamic nature of modern software, such as cost- or schedule-as-independent variable (CAIV or SAIV). At USC, we have evolved a highly successful SAIV approach for developing Web-based digital library systems on a necessarily-fixed schedule of 24 weeks. It works as follows:

- Manage the developers' and clients' expectations to recognize that not all features can be developed in 24 weeks, and have the clients prioritize their desired features.
- Using an estimation model providing optimistic–pessimistic schedule estimate ranges, converge on a core-capability set of top-priority features that even pessimistically is buildable in 24 weeks.
- Build the core capability, which usually will take less than 24 weeks, and use the remaining time to add the next-highest-priority features.

Even with a considerable dynamism and uncertainty in the nature of the desired product, this approach almost always produces a satisfactory result in a short, fixed development time.

As a final perspective, the dynamism of the software field means that the software estimation discipline needs to be continually reinventing itself. The articles in this special focus are good examples of this. Traditional software estimation models did not have to deal with graphical user interface builders, objects, process maturity, and Web-based systems. Traditional software estimation methods were either expert-based or model-based, and did not try to mix the two. The articles here show healthy new approaches to these phenomena and indications that the estimation field is rising to the challenge of continually reinventing itself.

## The Articles

Consistent with the theme of "Recent Developments in Software Estimation," this issue of *IEEE Software* presents six articles that report on promising estimation techniques, each of which can potentially improve the estimation process, the accuracy of the resulting estimates, and the productivity and quality achieved by software developers. The techniques reported are not "tried and true" because it is in the nature of recent developments in science and engineering that others must subject them to trial use before they become accepted practices. However, these authors present approaches that are worthy of consideration by estimators and by those who affect and are affected by estimates for software projects.

In "Improving Size Estimates Using Historical Data," James Bielak presents an analysis of a completed C++ project and shows that the number of GUI elements in a component and the number of GUI events handled by the component provide a rough estimate of the component's size in source lines of code. The number of methods in a component's interface and the number of components reused from the architecture can also be used to estimate size, but the estimate depends on a component's position within the overall architecture.

Analysis of defect data obtained from software inspections is often used to identify problem areas in software projects. Stefan Biffl, in "Using Inspection Data for Defect Estimation" presents the design and results of a large-scale experiment in which he investigated the accuracy of defect estimation models based on inspection data. He shows that the accuracy of subjective defect estimation models based on weighted averages of estimates by individual team members is superior to the accuracy of objective DEMs.

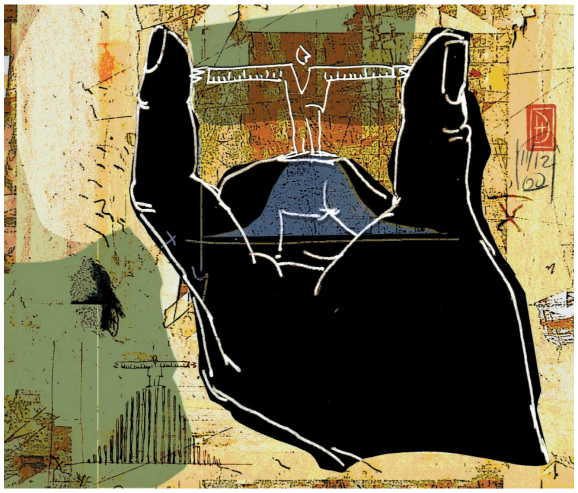
In "Enhancing the Cocomo Estimation Models," Joanne Hale, Allen Parrish, Randy Smith, and Brandon Dixon propose estimation adjustment factors based on the task assignments of project team members that can be used to improve the accuracy of existing cost estimation models. They show improvements in the predictive abilities of Cocomo I and Cocomo II when these factors are included.

"Empirically Guided Software Guesstimation" by Philip Johnson, Joseph Dane, Carleton Moore, and Robert Brewer reports on an experiment in which developer-generated "guesstimates" of software effort were more accurate than analytical estimates. However, they also found that access to a range of analytical estimation methods appeared to be useful to developers in generating their guesstimates and improving them over time.

In the article "Web Development: Estimating Quick-to-Market Software," Donald Reifer proposes a cost model for Web development projects that combines a size measure based on Halstead's Volume measure (using number of operators and operands) and a function-point-like table of complexity weights. Size estimates are used in Cocomo-like equations to produce estimates of effort and duration. He proposes eight cost drivers for the effort adjustment factor. As Reifer points out, there are still a large number of open issues to be resolved; however, his article shows an approach to developing estimation models for this important and growing domain of software engineering.

The final article, "Effects of Process Maturity on Software Development" by Brad Clark, presents the results of his analysis of 161 software projects (the USC Cocomo II database). His results indicate that, for the projects analyzed, a one-level change in process maturity resulted in a 4% to 11% reduction in project effort. Larger projects realize the larger gains. Clark applied the analysis across the five maturity levels of the Software CMM. He speculates that the percentage reduction in effort is not uniform across all levels. He could not determine this because his data did not contain a sufficient level of detail to permit analysis of improvement between levels. This work does, however, summarize a

**The ability to accurately scope projects is an essential element of an engineering discipline.**



solid analytical analysis of what till now has been largely anecdotal evidence that process maturity results in decreased project effort.

Software engineering is concerned with building software-intensive systems and products within the constraints of time, resources, technology, quality, and business considerations. The ability to scope projects accurately is an essential element of an engineering discipline. Estimation models, procedures, and techniques are essential components of the software engineering discipline. As the field changes, the techniques of estimation must, of necessity, change. The articles in this issue present new developments in software estimation that show the way to accommodating the needs of the always-changing world of software engineering.

## About the Editors

**Barry W. Boehm** is TRW Professor of Software Engineering and director of the Center for Software Engineering at the University of Southern California. He has served as director of the US Department of Defense DARPA Information Science and Technology Office, director of the DDR&E Software and Computer Technology Office, chief scientist of TRW's Defense Systems Group, and head of the Rand Corp.'s Information Sciences Department. His current research focuses on integrating a software system's process models, product models, property models, and success models via an approach called MBASE (Model-Based Architecting and Software Engineering). His contributions to the field include the Constructive Cost Model (Cocomo), the spiral model of the software process, and the Theory W (win–win) approach to software management and requirements determination.

Boehm received his BA from Harvard and his MS and PhD from UCLA, all in mathematics. He received an honorary ScD in computer science from the University of Massachusetts. He is an AIAA Fellow, an ACM Fellow, an INCOSE Fellow, an IEEE Fellow, and a member of the National Academy of Engineering. Contact him at USC Center for Software Engineering, Los Angeles, CA 90089-0781; boehm@sunset.usc.edu.

**Richard E. (Dick) Fairley** is a professor of computer science and director of the software engineering program at the Oregon Graduate Institute. He also teaches in the Oregon Master of Software Engineering degree program, offered collaboratively by four Oregon universities. His research interests include software estimation, project management, software process modeling, risk management, real-time systems, and software engineering education. Prior to this, he held tenured appointments at three universities, including dean of computer science at Colorado Technical University, and founded and ran a consulting company.

Fairley received his PhD in computer science from the University of California at Los Angeles; he also has a BS and MS in electrical engineering. Contact him at the Oregon Graduate Institute, Dept. of Computer Science and Engineering, 20000 N.W. Walker Rd., Beaverton, OR 97006; d.fairley@computer.org.

**MIT**

**SOFTWARE ENGINEERING FACULTY POSITION**

The Aeronautics and Astronautics Department [illegible]
a leader in the design and development of complex [illegible]
transportation, information telecommunications [illegible]
ty opening in the Aerospace Information Systems [illegible]
department seeks candidates for positions [illegible]
2001 or January 2002. The successful [illegible]
and relevant SW engineering research [illegible]
specification and analysis, assurance [illegible]
interaction, software design for [illegible]
and other quality attributes, [illegible]
cation issues like scheduling and [illegible]
with Computer Science is possible.

MIT encourages women and underrepresented [illegible]
Send two copies of cover letter (which includes [illegible]
est) and of your cv with the names and addresses [illegible]
als who will provide letters of recommendation, by [illegible]
2001, to: Professor Edward F. Crawley, Head, MIT [illegible]
Aeronautics and Astronautics, 33-207, 77 Massachusetts
Avenue, Cambridge, MA 02139-4307; or by electronic [illegible]
[illegible] (MS Word or plain text).
[illegible]

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
An Equal Opportunity/Affirmative Action Employer
Non-Smoking Environment
[illegible]

SMU | ENGINEERING

**Take a leadership role in Software Engineering.**

**Master of Science in Software Engineering**

A SMU graduate degree will give you a competitive advantage in the new millennium.

Core curriculum developed in consultation with Software Engineering Institute.

Emphasis on software and system design principles and practical problems of bringing products to market.

***Available nationwide through videotape distance education.***

**For information call**
**214.768.3232**
**www.seas.smu.edu**

SMU will not discriminate on the basis of race, color, religion, national origin, sex, age, disability, or veteran status.