Software Estimation: An Overview*

Richard D. Stutzke
Science Applications International Corporation

Introduction

Our world increasingly relies on software. There is a seemingly insatiable demand for more functionality, interfaces that are easier to use, faster response, and fewer defects. Developers must strive to achieve these objectives while simultaneously reducing development costs and cycle time. Above all, senior management wants software delivered on schedule and within cost, a rarity in past software development projects. Software process improvement (SPI), as advocated by the Software Engineering Institute (SEI), helps to achieve these objectives. Project planning and tracking are identified as two key process areas in the SEI's Capability Maturity Model (CMM®).

Software cost and schedule estimation supports the planning and tracking of software projects. The formal study of software estimating technology did not begin until the 1960s, although Peter Norden did some earlier work on models of research and development [1]. Estimation received renewed attention during the 1990s to cope with new ways of building software and to provide more accurate and dependable estimates of costs and schedules.

This article discusses the problems encountered in software estimation, surveys previous work in the field, and describes current work.

Description of the Problem

The estimator must estimate the effort (person-hours) and duration (calendar-days) of a project to enable managers to assess important quantities such as product costs, return on investment, time to market, and quality. The estimation process is difficult for several reasons:

- Conflicting project goals,
- Lack of a detailed product description,
- Wide variations in developer productivity,
- Difficulty of modifying existing code, and
- Emergence of new development processes, methods, and tools.

*This paper originally appeared in the May 1996 issue of the journal *CrossTalk*. An updated version appeared in the 5th edition of this book. This article updates and significantly expands the previous version. It contains more detail on some of the parametric models and additional information about some new models. The entire article has been reformatted to improve readability.

The first reason is that **projects often must satisfy conflicting goals**. Projects to develop (or maintain) software must provide specified functionality within specified performance criteria, within a specified cost and/or schedule, and with some desired level of quality (typically defined as the absence of defects). Software engineering processes can be chosen to meet any one of these project goals. Usually, however, more than one goal must be satisfied by a particular project. These multiple constraints complicate the estimating process.

The second reason is that **estimates are required before the product is well defined**. Software functionality is very difficult to define, especially in the early stages of a project. This makes it difficult to estimate the amount of software to be produced (the product size). The basis for the first good cost estimate is not usually available for totally new systems until the top-level design has been defined. (This design is an instance of the product's "software architecture.") This level of design is only defined at the preliminary design review (PDR) in U.S. Department of Defense (DoD) contracts (and sometimes not even then, which leads to undesirable consequences). This milestone is reached after approximately 20 percent of the total effort and 40 percent of the total duration have been expended by the project staff. At this point in a project, typical accuracies for the estimated effort and duration are within 25 percent of the final project actuals. In general, the accuracy of estimates increases as a project proceeds because more information becomes available, for example, product structure, product size, and team productivity. Figure 1 illustrates this and is adapted from Boehm [2]. (Commercial software projects behave similarly.) To reduce costs (as well as to improve quality and reduce development times), some projects have begun to employ predefined domain-specific software architectures (DSSAs). The development costs for such projects can be estimated more accurately than for projects that build a totally new product because more information about the product is available earlier. (The Advanced Research Projects Agency and the SEI, among others, are sponsoring work on DSSAs.) In general, however, the estimator must apply considerable skill to estimate product size, project cost, and project schedule early in a project.

For modifications of existing code, more data are available earlier and so more accurate estimates of product size are potentially possible than for totally new products. (But many factors affect the effort needed to modify existing code. See below.) This is important because approximately half of all software maintenance work is really a response to changes in the original requirements or in the system's external environment (mission, interfaces to other systems, etc.) and involves modification of existing code. (The choice of the software architecture significantly influences modifiability and hence maintainability. Architecture-based reuse is another motivation for the work on DSSAs.)

The third reason is that estimates of effort, cost, and schedule **depend on programmer productivity**. Software development is knowledge intensive. The main cost component is the labor expended by the development team. To first order, the effort is proportional to the amount of software to be developed (the product size), divided by the average productivity of the team. The productivity of different individuals can vary by a factor of 10 or more. As software intensive systems become larger and more complex, a single individual cannot comprehend all of the details and cannot master all of the needed skills. This means that teams of individuals must work together to design and build large software products and systems. The productivity of a team depends not only on the knowledge and skills of its members, but also on how the members communicate and interact. This is difficult to predict for newly formed teams, or for existing teams facing new application domains, product architectures, methods, or tools.

The **need to modify existing code** is the fourth reason that software cost estimating is difficult. It is hard to identify and quantify the factors affecting the effort needed to incorporate existing code in a product. The code must be located, understood, modified, integrated, and tested. The associated effort depends on how the code is structured, the programmer's knowledge of the code, and so forth. Modifying code is a subset of a broader activity called software reuse, which faces these same problems. (Reuse includes additional labor,

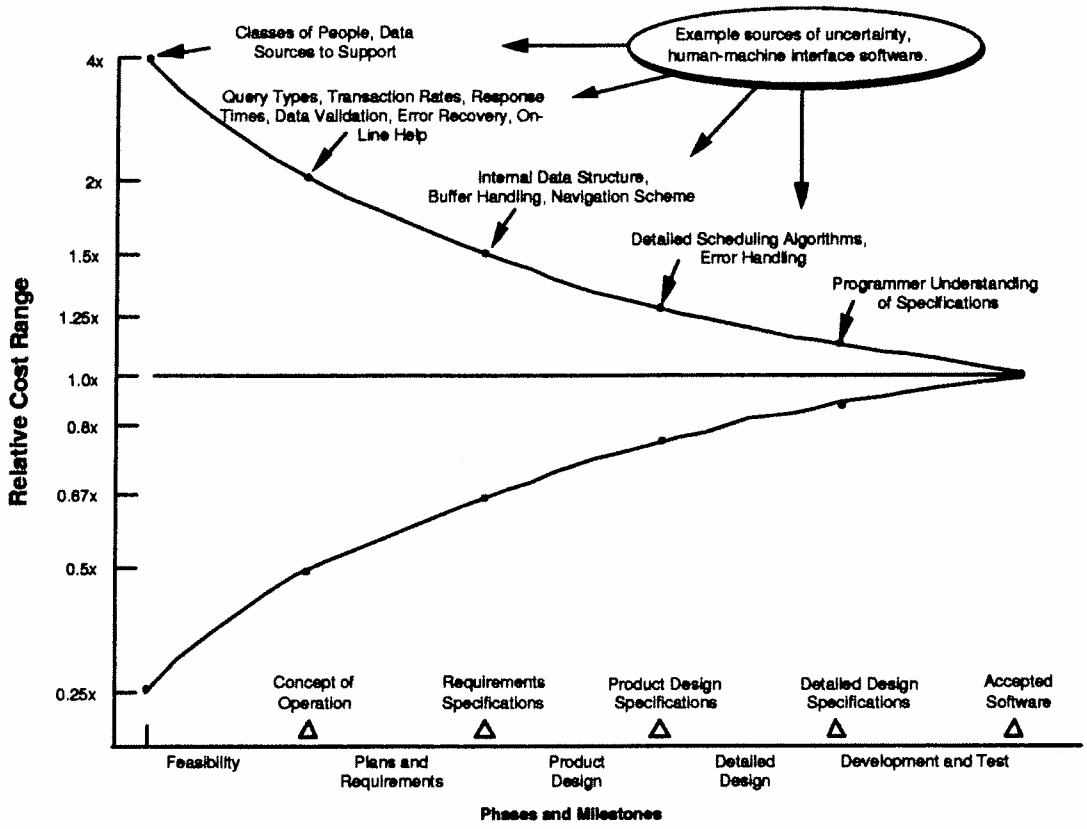

Figure 1: Software cost estimation accuracy versus phase. Reproduced with permission from [1].

plus costs for licenses, royalties, etc.)

Various cost estimation models have been developed to quantify the economic costs of building and reusing software components. For example, Richard Selby [3] analyzed costs at NASA's Software Engineering Laboratory and found that there is a large increase in programmer effort as soon as a programmer has to "look inside" the component. The amount of effort continues to increase as the amount of existing code modified increases. The exact amount of effort depends on the suitability of the software component for the intended application, its structure, and other factors. Figure 2 illustrates this, and is based on a model that I developed using data from Richard Selby [3], Rainer Gerlich and Ulrich Denskat [4], and Barry Boehm *et al.* [5]. (For details, see [6].) The figure shows the modification effort measured relative to the effort to develop totally new code, plotted versus the fraction of code modified. The two curves shown correspond to the best and worst cases for reuse based on the factors mentioned above. There are three noteworthy features of these curves. First, the effort is not zero, even if no code is modified. Second, the effort increases faster than linearly at first. Such nonlinear behavior is not handled in most existing software cost estimation models. (COCOMO II, described later, does handle this.) Third, the effort needed to modify all of the existing code is more than the effort to develop the same amount of code from scratch. (Effort is wasted to understand—and then discard—the existing code before the new code is written. This effort is never expended if the decision is made to develop totally new code from the start.) As shown in the figure, for the worst case, the economic break-even point occurs when only 23 percent of the code is modified; reuse is not cost effective above the break-even point. For the highest quality code, break-even occurs when 68 percent of the code must be modified.

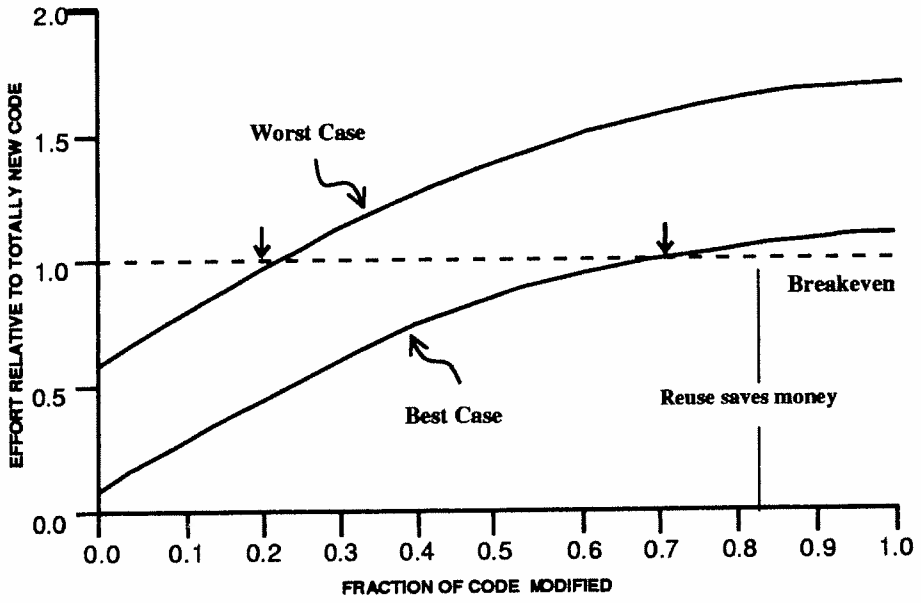

Figure 2: Reusing software is not always cost effective. Adapted from [6].

The fifth reason that estimation is difficult is because the way software is being built is changing. New development processes are emerging and new ways are needed to estimate the costs and schedules for the new processes. These processes endeavor to provide higher quality software (i.e., fewer defects), produce more modular and maintainable software, and/or deliver software (products and prototypes) to the end user faster. To meet these and the other objectives (stated previously), developers are using combinations of prebuilt code components, labor-saving tools, and nontraditional, even radical, development methods. For example, much programming is being put into the hands of the users themselves by providing macro definition capabilities in many products. These capabilities allow users to define sequences of frequently used commands[1]. A slightly more sophisticated approach is to allow domain experts to construct applications using special tools such as fourth-generation languages (4GLs) and application composition tools. Larger systems intended for specialized (one-of-a-kind) applications are often built using commercial off-the-shelf (COTS) products to provide functionality in areas that are understood well enough to be standardized. Examples are graphical user interfaces (GUI) and relational database management systems. The trend toward object-oriented languages supports this by making it easier to develop "plug-compatible" components. Thus, reuse of code is becoming an increasingly large factor in cost estimation. A good understanding of the cost factors associated with software reuse thus becomes even more important.

During the 1990s, several authors described "agile methods" that produce software using a series of rapid iterations. Each iteration starts by identifying the features to be added to the evolving product. The iterations last from 1 to 8 weeks. (Martin Fowler provides a good summary in [7].) The developers are also expected to evolve the method itself *during* the project. Planning and estimating for projects that use such development processes and methods prior to the start of the project are impossible, and controlling them is also difficult. This continues to be an active area of research.

[1]There is an analogy here to the situation shortly after telephones were introduced in the United States. Someone predicted that soon the majority of the United States' population would have to be employed as telephone operators in order to handle the workload of connecting calls. Instead, technology changed and now everyone serves as their own operator, using direct dialing to connect nearly all of their calls.

What is an Estimate?

As a minimum, the estimator must compute the effort (cost) and duration (schedule) for the project's process activities; identify associated costs such as equipment, travel, and staff training; and state the rationale behind the calculations (including the input values used, assumptions, etc.) Estimation is closely tied to the details of the product's requirements and design (primarily the software architecture), the activities of the chosen development process, and the resources (people, tools, and facilities) provided. The requirements, architecture, and reuse of existing code all affect the product size. The process activities and the project resources affect the productivity. Estimators must understand all of these factors in order to produce accurate estimates for project planning (i.e., these factors also affect estimates of product performance and quality.)

It is also highly desirable for the estimator to indicate the confidence in the reported values via ranges or standard deviations. The estimator should also try to state the assumptions and risks to highlight any areas of limited understanding relating to the requirements, product design, or development process.

Estimation Techniques

Estimation techniques use a mixture of analytic models, historical data, and expert judgment. These techniques differ based on the emphasis they place on each element, and the rules they use to combine the three elements. The paper that follows this one, "Software Engineering Economics" by Barry Boehm and Sunita Chulani, describes six categories of estimation techniques.

The next section gives a chronological summary of work in the field during the past four decades, with emphasis on analytic models (also called "parametric models"). The paper by Boehm and Chulani contains detailed descriptions of several popular software estimation models. Their paper also explains the important role that estimation plays in decisions involving limited resources.

Survey of Past Work

From simple beginnings in the 1960s, software estimation technology has expanded due to the work of many individuals.

The 1960s

In the 1960s, while at RCA, Frank Freiman developed the concept of parametric estimating, and this lead to the development of the PRICE model for hardware. This was the first generally available computerized estimating tool. It was extended to handle software in the 1970s.

The 1970s

The decade of the 1970s was a very active period. During this decade, the need to accurately predict the costs and schedules for software development became increasingly important and so began to receive more attention. Larger and larger systems were being built and many past projects had been financial disasters. Frederick Brooks, while at IBM, described many of these problems in his book *The Mythical Man-Month* [8]. His book provides an entertaining but realistic account of the problems as perceived at that time.

During the 1970s, high-order languages, such as FORTRAN, ALGOL, JOVIAL and Pascal, were coming into increasingly wider use but did not support reuse. Also, programming tools (other than compilers

for the languages and simple text editors) were in very limited use. For these two reasons, systems were essentially built by hand from scratch. The cost models of this period thus emphasized new development.

During this period, many authors analyzed project data using statistical techniques in an attempt to identify the major factors contributing to software development costs. Significant factors were identified using correlation techniques and were then incorporated in models using regression techniques. [Regression is a statistical method for predicting values of one or more dependent variables from a collection of independent (predictor) variables. Basically, the model's coefficients are chosen to produce the "best possible" fit to actual, validated project data.] Unfortunately, in practice there neither is enough data to define a model completely, nor are there any accepted "laws of software physics" that could constrain the form of the model's basic equations. As noted by Conte *et al.* [9], researchers must thus resort to so-called "composite models" created using a combination of analytic equations, statistical data fitting, and expert judgment. Experts differ on which independent variables are significant and on the form of estimating equations, giving rise to diverse models. The coefficients of a particular model are determined using actual project data and regression techniques. Most estimating models are in fact composite models. Such models are one form of cost estimating relation (CER).

The prototypical composite model is the COnstructive COst MOdel (COCOMO) developed by Barry W. Boehm in the late 1970s and described in his classic book *Software Engineering Economics* [2]. Various implementations of COCOMO continue to be widely used throughout the world. COCOMO provides formulas for estimating the total development effort and time. The "nominal" effort depends on the amount of software to be produced, measured in delivered source instructions (essentially physical source lines of code without comments). The nominal effort is adjusted to account for the effects of 15 cost drivers, which describe attributes of the product, computer, personnel, and project. (The project attributes include the effects of modern methods and tools.) Estimators use five- or six-point Likert scales to rate each cost driver attribute [10]. These ratings are converted into quantitative values using tables. The quantitative values are multiplied to give the total effort adjustment factor (EAF). (The range of effort adjustment, based on the ratio of the highest possible to lowest possible values of EAF, is 817.) The adjusted effort is then used to compute the development schedule. Finally, additional tables are used to allocate the effort and development time by phase and by activity.1

COCOMO distinguishes three development modes that essentially correspond to the formality of the development process. For example, the *embedded mode* is used to estimate the development of large, complex systems such as those built to military specifications. Different coefficients are used in the effort and schedule equations for each mode. COCOMO also has three levels of detail. Basic COCOMO excludes the effects of the cost driver attributes. Intermediate COCOMO uses the EAF as just described. Detailed COCOMO uses quantitative cost driver values that are phase dependent. Intermediate COCOMO is the version most commonly used. It predicts effort and schedule within $\pm$ 20 percent of the final actual values approximately 70 percent of the time. COCOMO, like other composite models, also provides ways to handle the reuse, modification, and maintenance of existing code.

The PRICE-Software Cost Model (PRICE-S), another software cost and schedule estimation model, was initially developed from 1975 through 1977 at RCA by Frank Freiman and Robert Park based on data from over 400 projects. Parametric models in the PRICE family were the first generally available computerized cost estimation models. William Rapp programmed the models, among them PRICE-S, to run on mainframe computers at RCA, making them available via a time-sharing (dial-in) service.

PRICE-S is described in [11] and [12]. The model operates as follows. The "central equation" computes the nominal effort to perform the tasks in an ideal world. The PRICE-S central equation computes effort based on the "volume" of the software to be produced. Volume is computed based on the amount of code to be produced [measured in source lines of code (SLOC) or *function points*], the programming language

used, and the complexity of the application. The nominal effort is then computed by raising the volume to a power that is a function of the organization's productivity factor (PROFAC)[2]. PROFAC is determined based on language type, application complexity, and platform type (described below) using a table. (The value of PROFAC can also be determined by calibration using the organization's own historical data; this is preferable.) The nominal effort is then adjusted to account for the effects of the various cost drivers. Approximately a dozen cost and schedule drivers are used in the PRICE-S model. These cost drivers adjust for nonnominal factors such as interface complexity, personnel skill, tools, utilization of available target computer resources, and so forth. The platform type cost driver describes the formality in which the development must take place. For example, software intended for internal use is developed less formally than software for manned space vehicles. The adjusted effort is then used to compute the nominal ("reference") schedule. Additional adjustments are made to obtain the final estimated schedule, as well as the final estimated effort (because compressing schedule and overlapping phases affect costs). The PRICE-S calculations also handle the reuse/reengineering of code. Studies made in 1979, 1981, and 1984 found that PRICE-S predicts costs within 8 to 15 percent of project actuals.

A shortcoming of these 1970s models is that the independent variables were often "result measures" such as the size in lines of code. Such values are readily measured, but only *after* the project has been completed. It is very difficult to predict the values of such variables before the start of the project[3]. This means that many of the models, although based on statistical analyses of actual result data, were hard to use in practice because the values of the independent variables were hard to determine before the project team had analyzed the requirements and had prepared a fairly detailed design. Another shortcoming of such models is that they assume that software will be developed using the same process as was used previously. This assumption is becoming increasingly unrealistic since new processes, methods, and tools continue to be developed.

At the end of the 1970s, Allan Albrecht and John Gaffney of IBM developed Function Point Analysis (FPA) for estimating the size and development effort for management information systems [13, 14]. Components of a system are classified into five types according to specific rules: inputs, outputs, queries, logical internal files, and external interface files. Each type has an assigned weight based on characteristics of the component. These weights are proportional to the development effort needed to construct components of that type. The estimator counts the number of components of each type, multiplies these counts by the corresponding weight, sums these products, and multiplies the sum by a factor to account for global system characteristics. The result is the "size" of the system measured in "function points." The estimator then uses the team's productivity (in function points per person-month) to compute the development effort.

Later in the 1970s, two authors endeavored to define models based on theoretical grounds. Lawrence H. Putnam [15] based his Software Lifecycle Model (SLIM) on the Norden-Rayleigh curve plus empirical data from 50 U.S. Army projects. Putnam's data indicate that the development staffing rises smoothly and drops sharply during acceptance testing. The shape fits the first part of the Norden-Rayleigh curve. Putnam uses this fact to relate the area under the curve (which corresponds to the development effort) to the curve's parameters. Putnam's other empirical results are expressed as two equations describing relations between

[2]The names of the phases and activities are unfortunately nearly identical: analysis, design, code, test, etc. The phases refer to the time when activities are performed. The problem is that activities span phases. For example, most of the effort for the requirements analysis activity is expended during the analysis phase, but some effort is also expended later to revise the requirements to reflect new knowledge obtained during the subsequent design and coding phases. This close correspondence of the phases and activities reflects the "waterfall" development of the 1970s. Other estimating tools do this also. Unfortunately, the close similarity of the names can sometimes cause users to misinterpret the numbers computed by such models.

[3]Actually, effort is computed using separate equations for three phases: design, implementation (coding), and testing. The actual functions used to compute the volume, the exponent, and the adjustments for some of the cost drivers are proprietary and are not published.

the development effort and the schedule. The first equation, called the *software equation*, states that development effort is proportional to the cube of the size and inversely proportional to the fourth power of the development time. The second equation, the *manpower buildup equation*, states that the effort is proportional to the cube of the development time. Solving these two coupled equations gives the basic predictive equations used in the commercial tool SLIM®. (SLIM is a registered trademark of Quantitative Software Management, Inc.) The solution represents the minimum development time (maximum development effort) for the project. The SLIM tool uses other management constraints (e.g., staffing caps and desired product reliability level) to define a bounded region of possible solutions representing minimum cost, minimum time, and so forth. Some authors, such as Conte *et al.* [9], have criticized SLIM's modeling of the effort/schedule trade-off. Putnam's (and SLIM's) software equation implies that effort scales inversely as the fourth power of the development time, leading to severe cost increases for compressed schedules. Actually, Putnam's model limits the range of applicability of the fourth-power model relation using other constraints. Generally, development time computed by the Putnam model can range between some minimum development time and a time approximately 30 percent greater than this minimum time.

The second author attempting to define a model based on theoretical grounds was Maurice H. Halstead [16]. Halstead defined software size in terms of the number of operators and operands defined in the program and proposed relations to estimate the development time and effort. To obtain this size information before the start of a project is of course nearly impossible because a good understanding of the detailed design is not available until later. Subsequent work by S. D. Conte *et al.* [9, p. 300] has shown that Halstead's relations are based on limited data and Halstead's model is no longer used for estimation purposes. (Don Coleman *et al.* [17] have recently reported some success in using it to predict the "maintainability" of software.)

Comparison of the Major Composite Models

It is instructive to compare the basic equations of the 1970s-era composite models. First, look at the development effort. SLIM has development effort proportional to the size raised to the 1.29 power. Both COCOMO (embedded mode) and Jensen's Software Estimation Model (SEM), described in the next section, have development effort proportional to the size raised to the 1.20 power. The PRICE-S model is more complex, and the full equations have not been published. One complication is that PRICE-S computes the effort for three phases and then sums these values. The other models, in contrast, compute a total core effort and then allocate it to the phases. We can say, however, that the effort estimated by PRICE-S appears to increase approximately linearly with the volume (which is equivalent to the size used by the other models) for a given value of the productivity factor (PROFAC). The slope of the line decreases with increasing values of PROFAC.

Most of these models have the nominal development time ("schedule") proportional to the cube root of the estimated development effort, which includes all of the adjustments for nonnominal conditions. This is exactly true for the SLIM model and the Jensen model described in the next section. (For details, see Kitchenham, published as Chapter 9 of [18].) COCOMO's embedded mode uses an exponent ranging from 0.32 to 0.38. It is not as easy to make a clear statement for PRICE-S because its schedule equations are complex and not all details are published. PRICE-S computes the nominal schedule as proportional to the volume raised to the 0.37 power. If we assume that the effort is proportional to the volume raised to a power of 1.1 or so, then the PRICE-S model would have schedule proportional to effort raised to approximately the 0.34 power, and so PRICE-S would appear to be consistent with the other models. (This is rather approximate of course since the PRICE-S volume consists of size (in SLOC) multiplied by factors reflecting complexity and language. The other models use size directly.)

The COCOMO, SLIM, and SEM models all have effort proportional to the size raised to the 1.2 or 1.3

power, and development time proportional to the cube root of the effort. PRICE-S has effort approximately proportional to the size raised 1.1 power, and development time is also approximately proportional to the cube root of the effort. The basic equations of these 1970s-era composite models are thus similar even though the models were developed independently. These similarities suggest that there may be some common "laws of software estimating," at least for the case of new software development. Because of these similarities, these models have comparable accuracies, generally predicting effort within 10 to 20 percent of project actuals for some appreciable fraction of the projects analyzed.

The 1980s

During the 1980s work continued to improve and consolidate the best models. As personal computers (PCs) started to come into general use, many models were programmed. Several firms began selling computerized estimating tools. Following the publication of the COCOMO equations in 1981, several tools that implemented COCOMO appeared during the latter half of the 1980s. (For proprietary models, the tool and the model are one and the same. The model exists only as implemented by the tool.)

The DoD introduced the Ada programming language in 1983 [American National Standards Institute (ANSI) and DOD-STD-1815A-1983] to reduce the costs of developing large systems. Certain features of Ada significantly impact development and maintenance costs and so Barry Boehm and Walker Royce defined a revised model called Ada COCOMO [19]. This model also addressed the fact that systems were being built incrementally in an effort to handle the inevitable changes in requirements.

Robert C. Tausworthe [20] extended the work of Boehm, Herd, Putnam, Walston and Felix, and Wolverton to develop a cost model for NASA's Jet Propulsion Laboratory. Tausworthe's model was further extended by Donald Reifer to produce the PC-based SOFTCOST-R model and a companion Ada estimating model called SOFTCOST-Ada. Both of these models are no longer sold commercially.

Randall W. Jensen [21] extended the work of Putnam by eliminating some of the undesirable behavior of Putnam's SLIM. Putnam's SLIM equation has development effort proportional to size (measured in SLOC) cubed divided by development time to the fourth power. Jensen asserted that development effort is proportional to the square of the size divided by the square of the development time. Both Jensen and Putnam apply the constraint that effort divided by the cube of the development time is less than some constant (which is chosen based on product and project parameters). Jensen's equations reduce to equations that are close to those of COCOMO's embedded mode but the effect of various cost drivers is handled quite differently. Daniel Galorath and coworkers continue to refine the Jensen model and market it as the Software Estimation Model (SEM), part of the System Evaluation and Estimation of Resources (SEER) toolset. (Jensen has recently proposed a new model that is described in the next section.)

In 1984, Albrecht [22] published a major revision to the FPA method. These revisions sharpened the rules for rating the complexity of the software. The original version of FPA had a single empirically derived weight for each type of component. The new method subdivided each type of component by complexity according to certain rules. Different weights are used for low-, medium-, and high-complexity components. This revised method is the basis for the current standard as defined by the International Function Point Users Group.

FPA was extended by Capers Jones [23] to include the effect of computationally complex algorithms on development costs. His Feature Point Method counts FPA's five types plus a sixth type called algorithms. His method eliminates the classification of the elements in terms of three levels of complexity; a single weight is used for each element type. Various PC-based tools implement these FPA-based methods, such as Function Point Workbench and Checkpoint from Software Productivity Research, and FPXpert and Estimacs from Computer Associates.

Charles Symons [24] proposed another revision of FPA to achieve the following major goals: reduce the subjectivity in dealing with files, make the size independent of whether the system was implemented as a single system or a set of related subsystems, and change the emphasis of function points away from gauging the value to users to predicting development effort. His method, called Mark II Function Points, bases the computation of size (functionality) on "logical transactions." Each processing activity done by the system is analyzed in terms of the number of data items input, referenced, and output. These are counted and weighted to compute the size in function points.

The 1990s

Because of the increasing diversity of software development processes, the 1990s saw renewed attention on developing improved cost models. In particular, Barry Boehm, now at the University of Southern California (USC), and his colleagues began to revise and extend the original COCOMO model in 1993. The new version of COCOMO, then called COCOMO 2.0, emerged in 1994 [5]. Since then the model has been renamed COCOMO II and has matured [25]. The updated model explicitly handles the availability of additional information in later stages of a project, the nonlinear costs of reusing software components, and the effects of several factors on the diseconomies of scale. (Some of these are the turnover rate of the staff, the geographic dispersion of the team, and the "maturity" of the development process as defined by the SEI.) The updated COCOMO II model has three versions: the application point model for initial estimates, the pre-architecture model, and the post-architecture model. (The application point model is described in the next paragraph.) The post-architecture model has 17 cost drivers and five scale factors. The scale factors are used to compute the exponents in the equations for nominal effort and schedule. (This eliminates discontinuities in the 1981 model related to the three development modes.) Besides adding some cost drivers, the multiplier values for some of the cost drivers have been revised. The dynamic range for the post-architecture model is 8,800. (The five scale factors contribute a factor of 4.3 to this value.) The pre-architecture model is a simplified version of the post-architecture model, obtained by combining cost drivers. It has 7 cost drivers and the same five scale factors. Both of these modules measure size in logical statements (LSLOC), instead of using physical source lines of code like COCOMO 1981. Both models also accept size in unadjusted function points (UFPs), converting to size in LSLOC using language-specific "backfire values" defined by Capers Jones. (Many existing parametric models do this as well.) The model also revises the handling of maintenance and adaptation (code reuse). A group of industrial and academic affiliates reviewed the revised COCOMO equations and provided data to calibrate the post-architecture model. The accuracy of COCOMO II.2000 is $\pm$ 30 percent of actuals 80 percent of the time for effort and for schedule. (For details, see Chapter 4 in [25].) Boehm and his coworkers expect to provide new calibrations biennially as more data are collected.

COCOMO II [25] defines an application point estimation model for use in small projects that construct products by combining existing software components. This method is based on a procedure in Appendix B.3 of [26] and productivity data for 19 projects presented in [27]. The total size of the software is estimated in application points. Then a nominal productivity is determined based on two factors: developer experience and capability, and integrated computer-aided software engineering (ICASE) tool maturity and capability. Dividing the size by the productivity gives the estimated effort. Application points are based on the following three elements: screens, reports, and components (assumed to be written in a third-generation language). The "objects" to be produced are identified and each is assigned a complexity rating of low, medium, or difficult. These ratings depend on the number of screen views, report sections, and referenced data tables, as well as on the source of the data tables (client or server). Weights are assigned to each pairing of object type and complexity. Adding all of the weighted instances gives the total application point count.

Multiplying by 1 minus the fraction of reuse expected or planned (i.e., 100 percent – reuse), decreases this count. This method combines simplified elements (similar to Gaffney's method described in the next paragraph) and complexity ratings of the elements (similar to FPA). The application point estimation model, while considered to be part of the COCOMO II model, has not yet been calibrated. It will no doubt evolve in the future.

In 1996, John Gaffney [28] reported that using only a subset of the elements of a function point count provides estimates of development effort that are as accurate as those produced using classical function points. His analysis showed that the development effort was highly correlated with just the counts of the inputs and outputs. (Gaffney actually evaluated six linear models and five nonlinear models.) Gaffney's "simplified function point estimation method" does not use the three levels of complexity (i.e., low, medium, high) for the elements that are part of the official IFPUG FPA method. (Capers Jones's Feature Points Method, as mentioned previously, also does not use the three levels of complexity.)

Scott Whitmire has proposed a way to extend FPA to handle scientific and real-time systems [29]. He acknowledges DeMarco's statement [30] that all software has three dimensions: data, function, and control. Whitmire asserts that classical FPA addresses only the data dimension of a software program. His three-dimensional (3D) function point method provides a way to quantify characteristics of the other two dimensions. The 3D function point index (i.e., the amount of functionality in the software) is computed in a way similar to that of classical FPA, with the addition of two new element types: transformations and transitions. The method is not yet rigorously validated. David Garmus has also described how to use function point counting in a real-time environment [31].

Randall Jensen has extended his original model [21] to explicitly handle the effects of management [32] on project costs and schedule. The new model, called SAGE (not an acronym for a longer term), considers factors such as the working environment (multiple development sites), team experience, and the degree of resource dedication. (The COCOMO 2.0 model also considers similar factors.) The SAGE model was first formulated in 1995 and handles new development. A new version that handles software maintenance was released in 1997.

The 2000s

The technology used to build systems continues to evolve rapidly. This technology includes hardware platforms, operating systems, data formats, transmission protocols, programming languages, methods, tools, and COTS components. The half-life of software engineering knowledge is typically less than 3 years. Many project teams are using nontraditional development processes such as Rapid Application Development (RAD), COTS integration, and agile methods to "grow" software systems. These processes will continue to evolve rapidly as we learn. Such constant and rapid changes mean that little relevant historical data will be available to help us estimate future software projects and to develop new cost estimation models. This may challenge SEI CMM® criteria relating to estimating, planning, and tracking that require the use of historical data and assume a stable process.

In an effort to keep up, researchers and model vendors are working to formulate, validate, and deploy new software estimation models. Much of this work is a continuation of work started in the 1990s. I cover it here because it is still in progress.

Allan Albrecht's function points, originally developed in the 1970s, are being extended to handle new software technologies and methods. Organizations such as the International Function Point Users Group (IFPUG), the Netherlands Software Metrics Users Association (NESMA), and the Federation of Software Measurement Associations (FESMA) are working to modernize the rules for function point counting. (See www.ifpug.org, www.nesma.nl, and www.fesma.org.) Lee Fischman has also proposed simplifying the

counting rules for standard function points [33]. His paper has references to recent work in this area.

Alain Abran and his collaborators [34] are working to update the measures of software size by extending function points to create what they call *full function points*. The two main organizations involved in this work are the Software Engineering Management Research Laboratory (Laboratoire de Recherche en Gestion des Logiciels) at the Université du Québec à Montréal (UQÀM) and the Common Software Measurement International Consortium (COSMIC). Full function points are based on a solid theoretical foundation and will be validated using actual data from modern projects (see [34] and [35]). Full function points (FFP) are now referred to as COSMIC-FFP. For the latest information, see www.lrgl.uqam.ca/cosmic-ffp or www.cosmicon.com.

Arlene Minkiewicz first proposed *predictive object points* (POPs) in 1997 [36]. The POP measure reflects three characteristics of object-oriented software: combined data and functions, object communication, and reuse via software. The POP measure is a function of four metrics: the number of top-level classes, the weighted methods per class, the average depth of the inheritance tree, and the average number of children per class. The function was validated using data from more than 20 projects (for details see [37]).

Some authors are attempting to link the products of analysis and design directly to the size measured in function points or some variant thereof. Specifically, they endeavor to tie the attributes of diagrams of the Unified Modeling Language (UML) to function points. This will make the counting of software size more objective. But this increased objectivity and precision comes at a price: We cannot count the size until after some analysis has been done. Some recent references are [35], [38], [39], and [40]. In particular, Fischman and McRitchie have also proposed basing an OO size measure on weighted methods per class [39]. Their Class-Method Points process has rules to identify and count only the "substantial" methods and classes. (Their paper also has references to OO metrics defined by other authors.)

Some recent work has been done to estimate the effort for Internet development projects using *user use case points* (UUCPs) [41]. UUCPs are counted in a manner similar to that of function points. The counting procedure has the following steps. First, the estimator identifies actors and assigns a weight to each (simple, average, or complex) and then sums these to obtain the total *unadjusted actor weights* (UAW). Second, the estimator similarly identifies and weights use cases, summing the weights to obtain the *unadjusted use case weights* (UUCW). Third, the estimator rates nine technical and environmental factors. Each is rated from 0 (not applicable) to 5 (essential for the project). It appears that they use different weights for each of the nine factors. The contribution of each factor is computed by multiplying the rating (0 to 5) times the weight. Adding the resulting values gives the technical complexity factor (TCF). The size in *adjusted use case points* (AUCP) is computed as $UUCP \times (0.6 + 0.1 \times TEF)$. (This formula is similar to the function point calculation except the two coefficients are 0.6 and 0.1, compared to 0.65 and 0.35 used for function point counting.) Using a productivity value determined from the organization's data, the estimator could compute the effort to implement the application. They tested this method using actual project data. The average error in the estimated effort was about 15 percent.

Geoffrey Sparks has adapted *use case points* (UCP) as defined by Geri Schneider and Jason Winters [42]. The method only counts the use cases, assigning weights based on complexity (simple, medium, or complex). The method has separate factors for technical complexity (TCF) and environmental complexity (ECF). The adjusted UCP equals $UUCP \times TCF \times ECF$. This multiplicative adjustment is similar to many other parametric models and is unlike the adjustment used in function point counting. (Contrast this to the formula for AUCP in the preceding paragraph.). Sparx Systems in Castlemaine Australia has implemented the method in a tool. For details see their Web site: www.sparxsystems.com.au.

UCPs have also been used to estimate testing effort [43]. Don Reifer has proposed "Web points" and a COCOMO-like model to estimate the effort and schedule for developing Web applications. (See the article entitled "Web Development: Estimating Quick-to-Market Software" in this chapter of this tutorial.)

Size is also difficult to define for prebuilt components. In many cases, the developer does not even have the source code, so measures such as SLOC are not feasible. In addition, only the portion of the component's interfaces and functionality that is actually used needs to be understood, integrated, and tested by the developer. The size needs to reflect only this portion for the purpose of estimating the developer's effort.

During the 1990s Boehm and his collaborators actually began to define a family of estimation models to estimate effort and schedule for different types of development processes. They have also done some work on a model to estimate software defect densities and reliability. As the importance of reliability increases, improved versions of such models will be needed. The family presently includes the following:

COCOMO II	–	COnstructive COst MOdel Version II
COCOTS	–	COnstructive COTS model
CORADMO	–	COnstructive Rapid Application Development MOdel
COSSEMO	–	COnstructive Staged Schedule & Effort MOdel
COPROMO	–	COnstructive PROductivity Improvement MOdel
COQUALMO	–	COnstructive QUALity Model

This family of models continues to evolve. In particular, the group plans to release new versions of COCOMO II incorporating the latest calibration data every 2 years or so. The versions are identified by their year of release, for example, COCOMO II.2000. For more information on these models see [25] and the COCOMO II Web site (http://sunset.usc.edu/COCOMOII/suite.html).

All model vendors are overhauling their proprietary models to cope with many of these same challenges. One trend is that the vendors are now packaging their estimation models with data collection capabilities. This supports integrated planning and tracking. It also provides the data needed to calibrate the estimation models to local development processes. (Many tools provide built-in tools to do this.) This enables organizations to retain the same basic toolset, yet adapt it to their changing development methods and processes.

Future Challenges

Technology and processes alone do not determine how businesses will develop software, license their products, etc. As the World Wide Web evolves, entirely new business models will become possible. For example, users may not buy software products, but instead rent them from service providers. (Costs would be based on actual usage.) Legal and security considerations may also affect the business models. Such influences will affect how software is built and sold. Estimating the trade-offs between development, operating, and maintenance costs will become more important and more difficult as new technologies and business models emerge. Estimators will require more knowledge of financial practices such as return on investment and discounted value. Barry Boehm covers such topics in Part III of his classic book [2].

Another factor that estimators must confront is the use of ephemeral teams. Projects need experts in multiple application and solution domains in order to build the large, complex systems. No single person can understand all of these domains and so development teams will be more interdisciplinary. Because all of these experts are not needed continuously, project teams (and possibly entire companies) may consist of a core of permanent experts and groups of temporary workers hired on a just-in-time basis. The permanent staff would include managers, project control, chief engineers, and so forth. The temporary workers would include analysts, designers, engineers, testers, support staff, and various domain experts. It will be challenging to assemble and manage such diverse, dynamic teams. Advances in telecommunications, networking, and support software ("groupware") can help such teams function even though they are geographically and

temporally dispersed. Such organizational structures affect estimators because they impact parameters such as the average staff capability, experience, and turnover.

There will also be a growing need for estimates of quality (defects), reliability, and availability, as well as the usual cost and schedule estimates. Developers and customers will become more interested in ensuring the safety and reliability of complex software systems such as those used for financial and medical applications. Estimating such characteristics is especially challenging for systems built using COTS components. For other perspectives on emerging trends, see recent articles in [45] and [45].

Developers of software estimation models continue to face the obstacle that no valid theoretical models of software development exist. There are no universal laws of "software physics" that can define constraints on and relationships between various independent variables characterizing the product and the project environment. Some approximate equations can be found, but these will surely change as computer technology advances. Consequently, software estimation will remain an experimental science for the foreseeable future. Estimators must rely on judgment and intuition to define heuristic rules and then validate these via analysis of actual project data. Once the significant factors have been isolated, simplified models can be defined and calibrated for use by the organization.

References

[1] Peter V. Norden, "Curve Fitting for a Model of Applied Research and Development Scheduling," *IBM Journal of Research and Development*, Vol. 2, No. 3, July 1958.

[2] Barry W. Boehm, *Software Engineering Economics*, Prentice-Hall, Upper Saddle River, NJ, 1981. Section 29.7 describes several models not discussed in this article, as well as the models developed by Herd, Putnam, Walston and Wolverton.

[3] Richard Selby, "Empirically Analyzing Reuse in a Production Environment," in *Software Reuse: Emerging Technology*, W. Tracz, editor, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 176–189.

[4] Rainer Gerlich and Ulrich Denskat, "A Cost Estimation Model for Maintenance and High Reuse," *Proceedings of the European Software Cost Modeling Conference (ESCOM 1994)*, Ivrea, Italy, May 1994.

[5] Barry W. Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby, "Cost Models for Future Software Lifecycle Processes: COCOMO 2.0," *Annals of Software Engineering*, Vol. 1, 1995, pp. 57–94. An earlier description was presented in the tutorial "COCOMO, Ada COCOMO and COCOMO 2.0" by Barry Boehm in the *Proceedings of the Ninth International COCOMO Estimation Meeting*, Los Angeles, CA, October 6-7, 1994.

[6] Richard D. Stutzke, "Deciding When It Is Cost Effective to Adapt Code," *Proceedings of the 7th European Software Control and Measurement Conference*, Wilmslow, England, May 15-17, 1996.

[7] Martin Fowler, "The New Methodology," March 2001, available at http://www.martinfowler.com/-newMethodology.html.

[8] Frederick P. Brooks, *The Mythical Man-Month*, Addison-Wesley, Reading, MA, 1975. An updated and expanded edition was published in 1995.

[9] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*, Benjamin Cummings, 1986.

[10] "A Technique for the Measurement of Attitudes," Rensis Likert, *Archives of Psychology*, Vol. 140, June 1932. Two brief discussions on Likert are available at http://www.oise.utoronto.ca/ lmclean/tomabs.html and http://www.vcu.edu/hasweb/psy/psy632/measure.htm.

[11] Robert E. Park, *The Central Equations of the PRICE-Software Cost Model*, PRICE-Systems, 1988.

[12] Arlene Minkiewicz and Anthony DeMarco, *The PRICE-Software Model*, Lockheed Martin PRICE-Systems, 1995.

[13] Allan J. Albrecht, "Measuring Application Development Productivity," *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, October 14–17, 1979.

[14] Allan J. Albrecht and John E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," *IEEE Transactions on Software Engineering*, Vol. 9, No. 2, November 1983.

[15] Lawrence H. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," *IEEE Transactions on Software Engineering*, Vol. SE-4, July 1978, pp. 345–361.

[16] Maurice H. Halstead, *Elements of Software Science*, Elsevier, New York, 1977.

[17] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman, "Using Metrics to Evaluate Software System Maintainability," *IEEE Computer*, Vol. 27, No. 8, August 1994, pp. 44–49.

[18] Norman E. Fenton, *Software Metrics: A Rigorous Approach*, Chapman and Hall, London, 1995. A revised edition, coauthored with Shari Lawrence Pfleeger, appeared in 1998.

[19] Barry W. Boehm and Walker Royce, "Ada COCOMO and the Ada Process Model," *Proceedings of the Third International COCOMO Users Meeting*, Software Engineering Institute, Pittsburgh, PA, November 1987, plus refinements presented at the Fourth International COCOMO Users Group Meeting held in November 1988.

[20] Robert C. Tausworthe, *Deep Space Network Estimation Model*, Jet Propulsion Report 81-7, 1981.

[21] Randall W. Jensen, "A Comparison of the Jensen and COCOMO Estimation Models," *Proceedings of the International Society of Parametric Analysts*, 1984, pp. 96–106.

[22] Allan J. Albrecht, *AD/M Productivity Measurement and Estimate Validation*, IBM Corporate Information Systems, IBM Corp., Purchase, NY, May 1984.

[23] Capers Jones, *The SPR Feature Point Method*, Software Productivity Research, Inc., 1986.

[24] Charles Symons, *Software Sizing and Estimating: Mark II Function Points (Function Point Analysis)*, John Wiley & Sons, New York, 1991, ISBN 0-471-92985-9.

[25] Barry W. Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald Reifer and Bert Steece, *Software Cost Estimation with COCOMO II*, Prentice-Hall, Upper Saddle River, NJ, 2000.

[26] R. Kauffman and R. Kumar, *Modeling Estimation Expertise in Object Based ICASE Environments*, Stern School of Business Report, New York University, January 1993.

[27] R. Banker, R. Kauffman and R. Kumar, "An Empirical Test of Object-Based Output Measurement Metrics on a Computer Aided Software Engineering (CASE) Environment," *Journal of Management Information Systems*, Vol. 8, No. 3, Winter 1991–1992, pp. 127–150.

[28] John E. Gaffney, Jr., "Software Cost Estimation Using Simplified Function Points," *Proceedings of the Eighth Annual Software Technology Conference*, Salt Lake City, UT, May 1996.

[29] Scott A. Whitmire, *3D Function Points: Scientific and Real-Time Extensions to Function Points*, Boeing Airplane Company Report BCS-G3252, 1992. It was published in the Proceedings of the 1992 Pacific Northwest Quality Conference. A more accessible reference by the same author is "An Introduction to 3D Function Points," *Software Development*, April 1995, p. 43.

[30] Tom DeMarco, *Controlling Software Projects*, Yourdon Press, New York, 1982.

[31] David Garmus, "Function Point Counting in a Real-Time Environment," *CrossTalk*, Vol. 9, No. 1, January 1996, pp. 11–14.

[32] Randall W. Jensen, "Management Impact on Software Cost and Schedule," *CrossTalk*, Vol. 9, No. 7, July 1996, pp. 6–10.

[33] Lee Fischman, "Evolving Function Points," *CrossTalk*, Vol. 14, No. 2, February 2001, pp. 24–27. *CrossTalk* is available at http://stsc.hill.af.mil. Recent information available at http://www.galorath.-com.

[34] Alain Abran and P. N. Robillard, "Function Point Analysis: An Empirical Study of Its Measurement Processes," *IEEE Transactions on Software Engineering*, Vol. 22, No. 12, December 1996, pp. 895–909.

[35] IWSM99, Proceedings of the 9th International Workshop on Software Measurement, Lac Supérieur, Québec, Canada, September 8–10, 1999. Accessible online at http://www.lrgl.uqam.ca/iwsm99/indes2.html. Stutzke, Labyad *et al.*, and Bévo et al. presented three papers dealing with UML-based size measures.

[36] Arlene Minkiewicz, "Predictive Object Points—Measuring the Size of OO Applications," *Proceedings of the 9th Software Technology Conference*, Salt Lake City, UT, April 27–May 2, 1997.

[37] Arlene Minkiewicz and John Staiger, "Estimating Object Oriented Software, An Automated Approach," *Proceedings of the 22nd Annual ISPA Conference*, Noordwijk, The Netherlands, May 8–10, 2000.

[38] Richard D. Stutzke, "Possible UML-Based Size Measures," *Proceedings of the 18th International Forum on COCOMO and Software Cost Modeling*, Los Angeles, CA, October 6–8, 1998.

[39] Lee Fischman and Karen McRitchie, "A Size Metric for UML," *Proceedings of the ISPA/SCEA Joint International Conference*, San Antonio, TX, June 8–11, 1999. Available at users.erols.com/scea/Conference99_pdfs/Conf99_53.pdf. Related articles can be found at www.galorath.com/presentations.html.

[40] Arlene Minkiewicz, *Measuring Object Oriented Software with Predictive Object Points*, PRICE-Systems Technical Report, 2000. Available at www.pricesystems.com/downloads/pdf/pops.pdf.

[41] Rakesh Agarwal, Santanu Banerjee, and Bhaskar Gosh, "Estimating Internet Based Projects: A Case Study," *Proceedings of the Quality Week 2001 Conference*, San Francisco, May 29–June 1, 2001, paper number 6W2. Accessible at www.soft.com/QualWeek/QW2001/papers/6W2.html.

[42] Geri Schneider and Jason Winters, *Applying Use Cases*, Addison-Wesley, Reading, MA, 1998, ISBN 0-201-30981-5.

[43] Suresh Nageswaran, "Test Effort Estimation Using Use Case Points (UCP)," *Proceedings of the Quality Week 2001 Conference*, San Francisco, May 29–June 1, 2001, paper number 4T2. Abstract available at www.soft.com/QualWeek/QW2001/papers/4T2.html. Paper available via www.cts-corp/cogcommunity/cogcomm_knowledge.htm.

[44] The April 2000 issue of *CrossTalk, the Journal of Defense Software Engineering*, presents articles on aspects of cost estimation. *CrossTalk*, Vol. 13, No. 4, April 2000, pp. 4–12, 14–17, and 20–24. Page 30 lists several web sites dealing with cost estimation. *CrossTalk* is available at www.stsc.hill.af.mil.

[45] The November/December 2000 issue of *IEEE Software* has several articles on software estimation. *IEEE Software*, Vol. 17, No. 6, November/December 2000, pp. 22–43, 45–49, and 51–70.