Software Engineering Economics

BARRY W. BOEHM

Abstract—This paper summarizes the current state of the art and recent trends in software engineering economics. It provides an overview of economic analysis techniques and their applicability to software engineering and management. It surveys the field of software cost estimation, including the major estimation techniques available, the state of the art in algorithmic cost models, and the outstanding research issues in software cost estimation.

Index Terms—Computer programming costs, cost models, management decision aids, software cost estimation, software economics, software engineering, software management.

I. INTRODUCTION

Definitions

The dictionary defines "economics" as "a social science concerned chiefly with description and analysis of the production, distribution, and consumption of goods and services." Here is another definition of economics which I think is more helpful in explaining how economics relates to software engineering.

Economics is the study of how people make decisions in resource-limited situations.

This definition of economics fits the major branches of classical economics very well.

Macroeconomics is the study of how people make decisions in resource-limited situations on a national or global scale. It deals with the effects of decisions that national leaders make on such issues as tax rates, interest rates, foreign and trade policy.

Microeconomics is the study of how people make decisions in resource-limited situations on a more personal scale. It deals with the decisions that individuals and organizations make on such issues as how much insurance to buy, which word processor to buy, or what prices to charge for their products or services.

Economics and Software Engineering Management

If we look at the discipline of software engineering, we see that the microeconomics branch of economics deals more with the types of decisions we need to make as software engineers or managers.

Clearly, we deal with limited resources. There is never enough time or money to cover all the good features we would like to put into our software products. And even in these days of cheap hardware and virtual memory, our more significant software products must always operate within a world of limited computer power and main memory. If you have been in the software engineering field for any length of time, I am sure

you can think of a number of decision situations in which you had to determine some key software product feature as a function of some limiting critical resource.

Throughout the software life cycle,[1] there are many decision situations involving limited resources in which software engineering economics techniques provide useful assistance. To provide a feel for the nature of these economic decision issues, an example is given below for each of the major phases in the software life cycle.

- *Feasibility Phase:* How much should we invest in information system analyses (user questionnaires and interviews, current-system analysis, workload characterizations, simulations, scenarios, prototypes) in order that we converge on an appropriate definition and concept of operation for the system we plan to implement?

- *Plans and Requirements Phase:* How rigorously should we specify requirements? How much should we invest in requirements validation activities (automated completeness, consistency, and traceability checks, analytic models, simulations, prototypes) before proceeding to design and develop a software system?

- *Product Design Phase:* Should we organize the software to make it possible to use a complex piece of existing software which generally but not completely meets our requirements?

- *Programming Phase:* Given a choice between three data storage and retrieval schemes which are primarily execution time-efficient, storage-efficient, and easy-to-modify, respectively; which of these should we choose to implement?

- *Integration and Test Phase:* How much testing and formal verification should we perform on a product before releasing it to users?

- *Maintenance Phase:* Given an extensive list of suggested product improvements, which ones should we implement first?

- *Phaseout:* Given an aging, hard-to-modify software product, should we replace it with a new product, restructure it, or leave it alone?

Outline of This Paper

The economics field has evolved a number of techniques (cost-benefit analysis, present value analysis, risk analysis, etc.)

---

[1] Economic principles underlie the overall structure of the software life cycle, and its primary refinements of prototyping, incremental development, and advancemanship. The primary economic driver of the life-cycle structure is the significantly increasing cost of making a software change or fixing a software problem, as a function of the phase in which the change or fix is made. See [11, ch. 4].

MASTER KEY
TO SOFTWARE ENGINEERING ECONOMICS
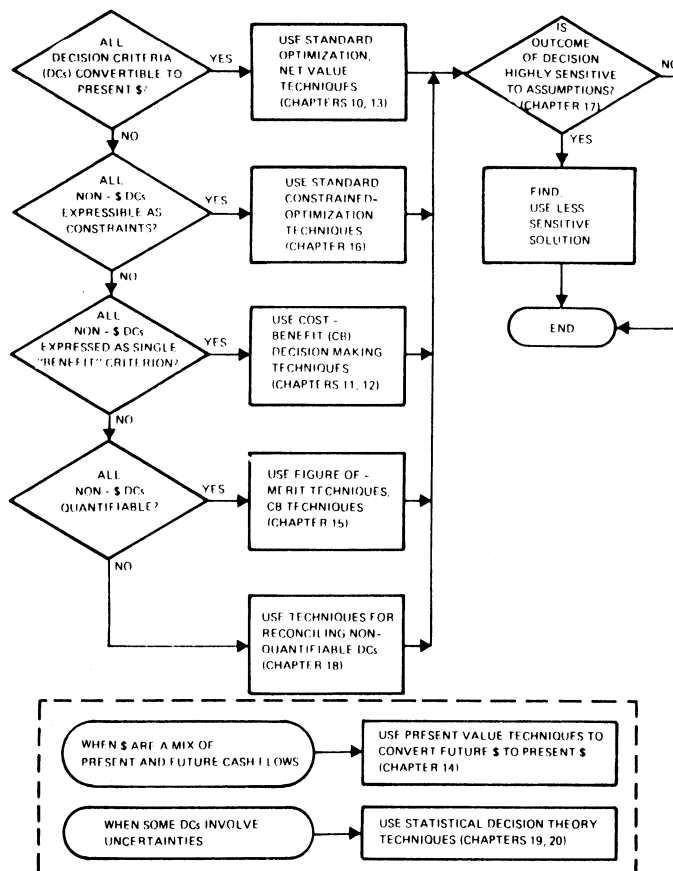## DECISION ANALYSIS TECHNIQUES

Fig. 1.   Master key to software engineering economics decision analysis techniques.

for dealing with decision issues such as the ones above. Section II of this paper provides an overview of these techniques and their applicability to software engineering.

One critical problem which underlies all applications of economic techniques to software engineering is the problem of estimating software costs. Section III contains three major sections which summarize this field:

III-A: Major Software Cost Estimation Techniques

III-B: Algorithmic Models for Software Cost Estimation

III-C: Outstanding Research Issues in Software Cost Estimation.

Section IV concludes by summarizing the major benefits of software engineering economics, and commenting on the major challenges awaiting the field.

II. SOFTWARE ENGINEERING ECONOMICS ANALYSIS TECHNIQUES

Overview of Relevant Techniques

The microeconomics field provides a number of techniques for dealing with software life-cycle decision issues such as the ones given in the previous section. Fig. 1 presents an overall master key to these techniques and when to use them.[2]

2 The chapter numbers in Fig. 1 refer to the chapters in [11], in which those techniques are discussed in further detail.

As indicated in Fig. 1, standard optimization techniques can be used when we can find a single quantity such as dollars (or pounds, yen, cruzeiros, etc.) to serve as a "universal solvent" into which all of our decision variables can be converted. Or, if the nondollar objectives can be expressed as constraints (system availability must be at least 98 percent; throughput must be at least 150 transactions per second), then standard constrained optimization techniques can be used. And if cash flows occur at different times, then present-value techniques can be used to normalize them to a common point in time.

More frequently, some of the resulting benefits from the software system are not expressible in dollars. In such situations, one alternative solution will not necessarily dominate another solution.

An example situation is shown in Fig. 2, which compares the cost and benefits (here, in terms of throughput in transactions per second) of two alternative approaches to developing an operating system for a transaction processing system.

- *Option A:* Accept an available operating system. This will require only $80K in software costs, but will achieve a peak performance of 120 transactions per second, using five $10K minicomputer processors, because of a high multiprocessor overhead factor.
- *Option B:* Build a new operating system. This system would be more efficient and would support a higher peak throughput, but would require $180K in software costs.

The cost-versus-performance curve for these two options are shown in Fig. 2. Here, neither option dominates the other, and various cost-benefit decision-making techniques (maximum profit margin, cost/benefit ratio, return on investments, etc.) must be used to choose between Options A and B.

In general, software engineering decision problems are even more complex than Fig. 2, as Options A and B will have several important criteria on which they differ (e.g., robustness, ease of tuning, ease of change, functional capability). If these criteria are quantifiable, then some type of figure of merit can be defined to support a comparative analysis of the preferability of one option over another. If some of the criteria are unquantifiable (user goodwill, programmer morale, etc.), then some techniques for comparing unquantifiable criteria need to be used. As indicated in Fig. 1, techniques for each of these situations are available, and discussed in [11].

Analyzing Risk, Uncertainty, and the Value of Information

In software engineering, our decision issues are generally even more complex than those discussed above. This is because the outcome of many of our options cannot be determined in advance. For example, building an operating system with a significantly lower multiprocessor overhead may be achievable, but on the other hand, it may not. In such circumstances, we are faced with a problem of *decision making under uncertainty,* with a considerable *risk* of an undesired outcome.
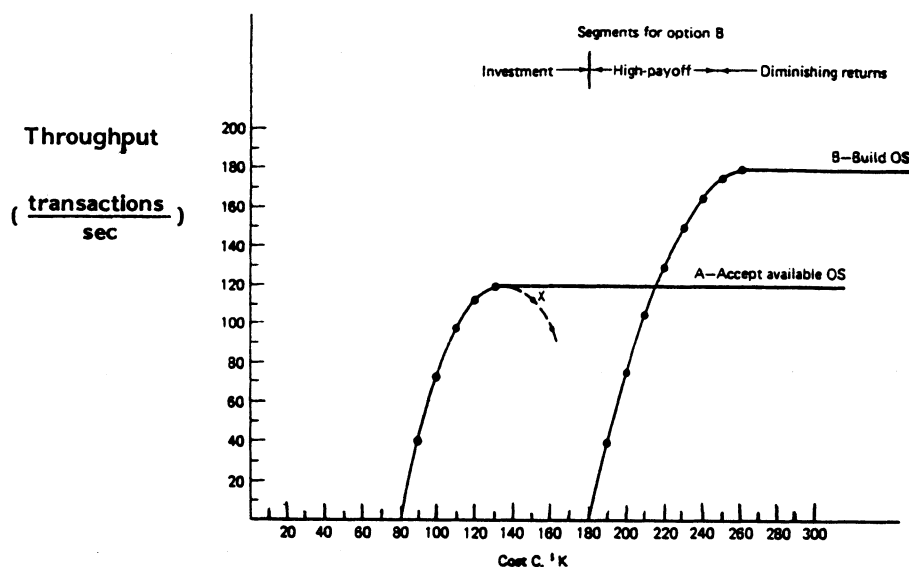
Fig. 2. Cost-effectiveness comparison, transaction processing system options.

The main economic analysis techniques available to support us in resolving such problems are the following.

1) Techniques for decision making under complete uncertainty, such as the maximax rule, the maximin rule, and the Laplace rule [38]. These techniques are generally inadequate for practical software engineering decisions.

2) Expected-value techniques, in which we estimate the probabilities of occurrence of each outcome (successful or unsuccessful development of the new operating system) and complete the expected payoff of each option:

$$EV = \text{Prob(success)} * \text{Payoff(successful OS)}$$
$$+ \text{Prob(failure)} * \text{Payoff(unsuccessful OS)}.$$

These techniques are better than decision making under complete uncertainty, but they still involve a great deal of risk if the Prob(failure) is considerably higher than our estimate of it.

3) Techniques in which we reduce uncertainty by *buying information*. For example, *prototyping* is a way of buying information to reduce our uncertainty about the likely success or failure of a multiprocessor operating system; by developing a rapid prototype of its high-risk elements, we can get a clearer picture of our likelihood of successfully developing the full operating system.

In general, prototyping and other options for buying information[3] are most valuable aids for software engineering decisions. However, they always raise the following question: "how much information-buying is enough?"

In principle, this question can be answered via statistical decision theory techniques involving the use of Bayes' Law, which allows us to calculate the expected payoff from a software project as a function of our level of investment in a prototype

or other information-buying option. (Some examples of the use of Bayes' Law to estimate the appropriate level of investment in a prototype are given in [11, ch. 20].)

In practice, the use of Bayes' Law involves the estimation of a number of conditional probabilities which are not easy to estimate accurately. However, the Bayes' Law approach can be translated into a number of *value-of-information guidelines*, or conditions under which it makes good sense to decide on investing in more information before committing ourselves to a particular course of action.

Condition 1: There exist attractive alternatives whose payoff varies greatly, depending on some critical states of nature. If not, we can commit ourselves to one of the attractive alternatives with no risk of significant loss.

Condition 2: The critical states of nature have an appreciable probability of occurring. If not, we can again commit ourselves without major risk. For situations with extremely high variations in payoff, the appreciable probability level is lower than in situations with smaller variations in payoff.

Condition 3: The investigations have a high probability of accurately identifying the occurrence of the critical states of nature. If not, the investigations will not do much to reduce our risk of loss due to making the wrong decision.

Condition 4: The required cost and schedule of the investigations do not overly curtail their net value. It does us little good to obtain results which cost more than they can save us, or which arrive too late to help us make a decision.

Condition 5: There exist significant side benefits derived from performing the investigations. Again, we may be able to justify an investigation solely on the basis of its value in training, team-building, customer relations, or design validation.

Some Pitfalls Avoided by Using the Value-of-Information Approach

The guideline conditions provided by the value-of-information approach provide us with a perspective which helps us avoid some serious software engineering pitfalls. The pitfalls

---

[3] Other examples of options for buying information to support software engineering decisions include feasibility studies, user surveys, simulation, testing, and mathematical program verification techniques.

below are expressed in terms of some frequently expressed but faulty pieces of software engineering advice.

Pitfall 1: Always use a simulation to investigate the feasibility of complex realtime software. Simulations are often extremely valuable in such situations. However, there have been a good many simulations developed which were largely an expensive waste of effort, frequently under conditions that would have been picked up by the guidelines above. Some have been relatively useless because, once they were built, nobody could tell whether a given set of inputs was realistic or not (picked up by Condition 3). Some have been taken so long to develop that they produced their first results the week after the proposal was sent out, or after the key design review was completed (picked up by Condition 4).

Pitfall 2: Always build the software twice. The guidelines indicate that the prototype (or build-it-twice) approach is often valuable, but not in all situations. Some prototypes have been built of software whose aspects were all straightforward and familiar, in which case nothing much was learned by building them (picked up by Conditions 1 and 2).

Pitfall 3: Build the software purely top-down. When interpreted too literally, the top-down approach does not concern itself with the design of low level modules until the higher levels have been fully developed. If an adverse state of nature makes such a low level module (automatically forecast sales volume, automatically discriminate one type of aircraft from another) impossible to develop, the subsequent redesign will generally require the expensive rework of much of the higher level design and code. Conditions 1 and 2 warn us to temper our top-down approach with a thorough top-to-bottom software risk analysis during the requirements and product design phases.

Pitfall 4: Every piece of code should be proved correct. Correctness proving is still an expensive way to get information on the fault-freedom of software, although it strongly satisfies Condition 3 by giving a very high assurance of a program's correctness. Conditions 1 and 2 recommend that proof techniques be used in situations where the operational cost of a software fault is very large, that is, loss of life, compromised national security, major financial losses. But if the operational cost of a software fault is small, the added information on fault-freedom provided by the proof will not be worth the investment (Condition 4).

Pitfall 5: Nominal-case testing is sufficient. This pitfall is just the opposite of Pitfall 4. If the operational cost of potential software faults is large, it is highly imprudent not to perform off-nominal testing.

Summary: The Economic Value of Information

Let us step back a bit from these guidelines and pitfalls. Put simply, we are saying that, as software engineers:

"It is often worth paying for information because it helps us make better decisions."

If we look at the statement in a broader context, we can see that it is the primary reason why the software engineering field exists. It is what practically all of our software customers say when they decide to acquire one of our products: that it is worth paying for a management information system, a weather forecasting system, an air traffic control system, an inventory control system, etc., because it helps them make better decisions.

Usually, software engineers are *producers* of management information to be consumed by other people, but during the software life cycle we must also be *consumers* of management information to support our own decisions. As we come to appreciate the factors which make it attractive for us to pay for processed information which helps *us* make better decisions as software engineers, we will get a better appreciation for what our customers and users are looking for in the information processing systems we develop for *them*.

III. SOFTWARE COST ESTIMATION

Introduction

All of the software engineering economics decision analysis techniques discussed above are only as good as the input data we can provide for them. For software decisions, the most critical and difficult of these inputs to provide are estimates of the cost of a proposed software project. In this section, we will summarize:

1) the major software cost estimation techniques available, and their relative strengths and difficulties;
2) algorithmic models for software cost estimation;
3) outstanding research issues in software cost estimation.

A. Major Software Cost Estimation Techniques

Table I summarizes the relative strengths and difficulties of the major software cost estimation methods in use today.

1) *Algorithmic Models:* These methods provide one or more algorithms which produce a software cost estimate as a function of a number of variables which are considered to be the major cost drivers.

2) *Expert Judgment:* This method involves consulting one or more experts, perhaps with the aid of an expert-consensus mechanism such as the Delphi technique.

3) *Analogy:* This method involves reasoning by analogy with one or more completed projects to relate their actual costs to an estimate of the cost of a similar new project.

4) *Parkinson:* A Parkinson principle ("work expands to fill the available volume") is invoked to equate the cost estimate to the available resources.

5) *Price-to-Win:* Here, the cost estimate is equated to the price believed necessary to win the job (or the schedule believed necessary to be first in the market with a new product, etc.).

6) *Top-Down:* An overall cost estimate for the project is derived from global properties of the software product. The total cost is then split up among the various components.

7) *Bottom-Up:* Each component of the software job is separately estimated, and the results aggregated to produce an estimate for the overall job.

The main conclusions that we can draw from Table I are the following.

• None of the alternatives is better than the others from all aspects.

• The Parkinson and price-to-win methods are unacceptable and do not produce satisfactory cost estimates.

TABLE I
STRENGTHS AND WEAKNESSES OF SOFTWARE
COST-ESTIMATION METHODS

| Method	Strengths	Weaknesses
Algorithmic model	• Objective, repeatable, analyzable formula • Efficient, good for sensitivity analysis • Objectively calibrated to experience	• Subjective inputs • Assessment of exceptional circumstances • Calibrated to past, not future
Expert judgment	• Assessment of representativeness, interactions, exceptional circumstances	• No better than participants • Biases, incomplete recall
Analogy	• Based on representative experience	• Representativeness of experience
Parkinson Price to win	• Correlates with some experience • Often gets the contract	• Reinforces poor practice • Generally produces large overruns
Top-down	• System level focus • Efficient	• Less detailed basis • Less stable
Bottom-up	• More detailed basis • More stable • Fosters individual commitment	• May overlook system level costs • Requires more effort
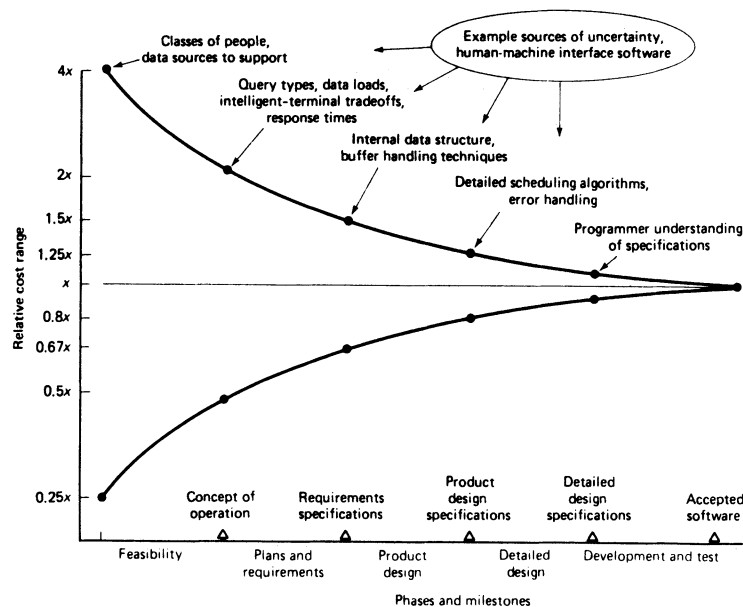
Fig. 3.   Software cost estimation accuracy versus phase.

• The strengths and weaknesses of the other techniques are complementary (particularly the algorithmic models versus expert judgment and top-down versus bottom-up).

• Thus, in practice, we should use combinations of the above techniques, compare their results, and iterate on them where they differ.

Fundamental Limitations of Software Cost Estimation Techniques

Whatever the strengths of a software cost estimation technique, there is really no way we can expect the technique to compensate for our lack of definition or understanding of the software job to be done. Until a software specification is fully defined, it actually represents a range of software products, and a corresponding range of software development costs.

This fundamental limitation of software cost estimation technology is illustrated in Fig. 3, which shows the accuracy within which software cost estimates can be made, as a function of the software life-cycle phase (the horizontal axis), or of the level of knowledge we have of what the software is intended to do. This level of uncertainty is illustrated in Fig. 3

with respect to a human–machine interface component of the software.

When we first begin to evaluate alternative concepts for a new software application, the relative range of our software cost estimates is roughly a factor of four on either the high or low side.[4] This range stems from the wide range of uncertainty we have at this time about the actual nature of the product. For the human–machine interface component, for example, we do not know at this time what classes of people (clerks, computer specialists, middle managers, etc.) or what classes of data (raw or pre-edited, numerical or text, digital or analog) the system will have to support. Until we pin down such uncertainties, a factor of four in either direction is not surprising as a range of estimates.

The above uncertainties are indeed pinned down once we complete the feasibility phase and settle on a particular concept of operation. At this stage, the range of our estimates diminishes to a factor of two in either direction. This range is

---

[4] These ranges have been determined subjectively, and are intended to represent 80 percent confidence limits, that is, "within a factor of four on either side, 80 percent of the time."

reasonable because we still have not pinned down such issues as the specific types of user query to be supported, or the specific functions to be performed within the microprocessor in the intelligent terminal. These issues will be resolved by the time we have developed a software requirements specification, at which point, we will be able to estimate the software costs within a factor of 1.5 in either direction.

By the time we complete and validate a product design specification, we will have resolved such issues as the internal data structure of the software product and the specific techniques for handling the buffers between the terminal microprocessor and the central processors on one side, and between the microprocessor and the display driver on the other. At this point, our software estimate should be accurate to within a factor of 1.25, the discrepancies being caused by some remaining sources of uncertainty such as the specific algorithms to be used for task scheduling, error handling, abort processing, and the like. These will be resolved by the end of the detailed design phase, but there will still be a residual uncertainty about 10 percent based on how well the programmers really understand the specifications to which they are to code. (This factor also includes such consideration as personnel turnover uncertainties during the development and test phases.)

B. Algorithmic Models for Software Cost Estimation

Algorithmic Cost Models: Early Development

Since the earliest days of the software field, people have been trying to develop algorithmic models to estimate software costs. The earliest attempts were simple rules of thumb, such as:

- on a large project, each software performer will provide an average of one checked-out instruction per man-hour (or roughly 150 instructions per man-month);
- each software maintenance person can maintain four boxes of cards (a box of cards held 2000 cards, or roughly 2000 instructions in those days of few comment cards).

Somewhat later, some projects began collecting quantitative data on the effort involved in developing a software product, and its distribution across the software life cycle. One of the earliest of these analyses was documented in 1956 in [8]. It indicated that, for very large operational software products on the order of 100 000 delivered source instructions (100 KDSI), that the overall productivity was more like 64 DSI/man-month, that another 100 KDSI of support-software would be required; that about 15 000 pages of documentation would be produced and 3000 hours of computer time consumed; and that the distribution of effort would be as follows:

Program Specs:	10 percent
Coding Specs:	30 percent
Coding:	10 percent
Parameter Testing:	20 percent
Assembly Testing:	30 percent

with an additional 30 percent required to produce operational specs for the system. Unfortunately, such data did not become well known, and many subsequent software projects went through a painful process of rediscovering them.

During the late 1950's and early 1960's, relatively little progress was made in software cost estimation, while the frequency and magnitude of software cost overruns was becoming critical to many large systems employing computers. In 1964, the U.S. Air Force contracted with System Development Corporation for a landmark project in the software cost estimation field. This project collected 104 attributes of 169 software projects and treated them to extensive statistical analysis. One result was the 1965 SDC cost model [41] which was the best possible statistical 13-parameter linear estimation model for the sample data:

$$MM = -33.63$$
$$+9.15 \text{ (Lack of Requirements) (0-2)}$$
$$+10.73 \text{ (Stability of Design) (0-3)}$$
$$+0.51 \text{ (Percent Math Instructions)}$$
$$+0.46 \text{ (Percent Storage/Retrieval Instructions)}$$
$$+0.40 \text{ (Number of Subprograms)}$$
$$+7.28 \text{ (Programming Language) (0-1)}$$
$$-21.45 \text{ (Business Application) (0-1)}$$
$$+13.53 \text{ (Stand-Alone Program) (0.1)}$$
$$+12.35 \text{ (First Program on Computer) (0-1)}$$
$$+58.82 \text{ (Concurrent Hardware Development) (0-1)}$$
$$+30.61 \text{ (Random Access Device Used) (0-1)}$$
$$+29.55 \text{ (Difference Host, Target Hardware) (0-1)}$$
$$+0.54 \text{ (Number of Personnel Trips)}$$
$$-25.20 \text{ (Developed by Military Organization) (0-1)}.$$

The numbers in parentheses refer to ratings to be made by the estimator.

When applied to its database of 169 projects, this model produced a mean estimate of 40 MM and a standard deviation of 62 MM; not a very accurate predictor. Further, the application of the model is counterintuitive; a project with all zero ratings is estimated at minus 33 MM; changing language from a higher order language to assembly language adds 7 MM, independent of project size. The most conclusive result from the SDC study was that there were too many nonlinear aspects of software development for a linear cost-estimation model to work very well.

Still, the SDC effort provided a valuable base of information and insight for cost estimation and future models. Its cumulative distribution of productivity for 169 projects was a valuable aid for producing or checking cost estimates. The estimation rules of thumb for various phases and activities have been very helpful, and the data have been a major foundation for some subsequent cost models.

In the late 1960's and early 1970's, a number of cost models were developed which worked reasonably well for a certain restricted range of projects to which they were calibrated. Some of the more notable examples of such models are those described in [3], [54], [57].

The essence of the TRW Wolverton model [57] is shown in Fig. 4, which shows a number of curves of software cost per object instruction as a function of relative degree of difficulty
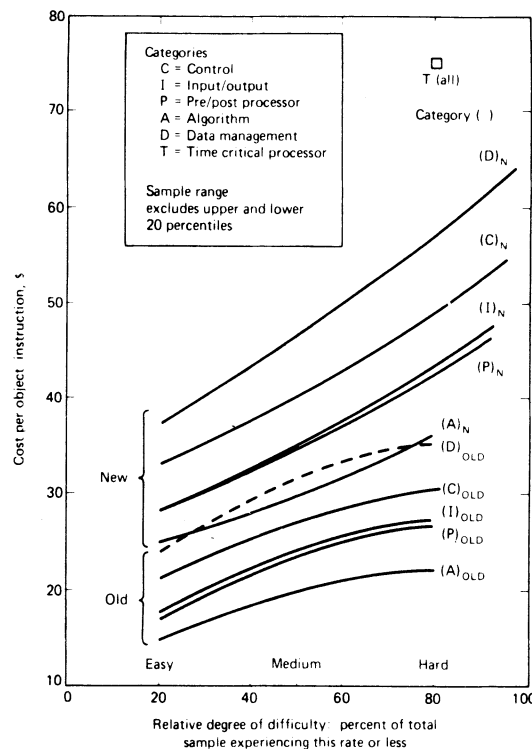
Fig. 4.   TRW Wolverton model: Cost per object instruction versus relative degree of difficulty.

(0 to 100), novelty of the application (new or old), and type of project. The best use of the model involves breaking the software into components and estimating their cost individually. This, a 1000 object-instruction module of new data management software of medium (50 percent) difficulty would be costed at $46/instruction, or $46 000.

This model is well-calibrated to a class of near-real-time government command and control projects, but is less accurate for some other classes of projects. In addition, the model provides a good breakdown of project effort by phase and activity.

In the late 1970's, several software cost estimation models were developed which established a significant advance in the state of the art. These included the Putnam SLIM Model [44], the Doty Model [27], the RCA PRICE S model [22], the COCOMO model [11], the IBM-FSD model [53], the Boeing model [9], and a series of models developed by GRC [15]. A summary of these models, and the earlier SDC and Wolverton models, is shown in Table II, in terms of the size, program, computer, personnel, and project attributes used by each model to determine software costs. The first four of these models are discussed below.

The Putnam SLIM Model [44], [45]

The Putnam SLIM Model is a commercially available (from Quantitative Software Management, Inc.) software product based on Putnam's analysis of the software life cycle in terms of the Rayleigh distribution of project personnel level versus time. The basic effort macro-estimation model used in SLIM is

$$S_s = C_K K^{1/3} t_d^{4/3}$$

where

S_s = number of delivered source instructions
K = life-cycle effort in man-years
t_d = development time in years
C_k = a "technology constant."

Values of C_k typically range between 610 and 57 314. The current version of SLIM allows one to calibrate C_k to past projects or to past projects or to estimate it as a function of a project's use of modern programming practices, hardware constraints, personnel experience, interactive development, and other factors. The required development effort, DE, is estimated as roughly 40 percent of the life-cycle effort for large systems. For smaller systems, the percentage varies as a function of system size.

The SLIM model includes a number of useful extensions to estimate such quantities as manpower distribution, cash flow, major-milestone schedules, reliability levels, computer time, and documentation costs.

The most controversial aspect of the SLIM model is its tradeoff relationship between development effort K and between development time t_d. For a software product of a given size, the SLIM software equation above gives

$$K = \frac{\text{constant}}{t_d^4}.$$

For example, this relationship says that one can cut the cost of a software project in half, simply by increasing its development time by 19 percent (e.g., from 10 months to 12 months). Fig. 5 shows how the SLIM tradeoff relationship com-

TABLE II
FACTORS USED IN VARIOUS COST MODELS

| GROUP	FACTOR	SDC, 1965	TRW, 1972	PUTNAM, SLIM	DOTY	RCA, PRICE S	IBM	BOEING, 1977	GRC, 1979	COCOMO	SOFCOST	DSN	JENSEN
SIZE ATTRIBUTES	SOURCE INSTRUCTIONS			X	X		X	X		X	X	X	X
	OBJECT INSTRUCTIONS	X	X		X	X							
	NUMBER OF ROUTINES	X				X					X		
	NUMBER OF DATA ITEMS						X			X	X		
	NUMBER OF OUTPUT FORMATS								X			X	
	DOCUMENTATION				X		X				X		X
	NUMBER OF PERSONNEL			X			X	X			X		X
PROGRAM ATTRIBUTES	TYPE	X	X	X	X	X	X	X			X		
	COMPLEXITY		X	X		X	X			X	X	X	X
	LANGUAGE	X		X				X	X		X	X	
	REUSE			X		X		X	X	X	X	X	X
	REQUIRED RELIABILITY			X		X				X	X		X
	DISPLAY REQUIREMENTS				X						X		X
COMPUTER ATTRIBUTES	TIME CONSTRAINT		X	X	X	X	X	X	X	X	X	X	X
	STORAGE CONSTRAINT			X	X	X	X		X	X	X	X	X
	HARDWARE CONFIGURATION	X				X							
	CONCURRENT HARDWARE DEVELOPMENT	X			X	X	X			X		X	X
	INTERFACING EQUIPMENT, S/W										X	X	
PERSONNEL ATTRIBUTES	PERSONNEL CAPABILITY			X		X	X			X	X	X	X
	PERSONNEL CONTINUITY						X					X	
	HARDWARE EXPERIENCE	X		X	X	X	X		X	X	X	X	X
	APPLICATIONS EXPERIENCE		X	X		X	X	X	X	X	X	X	X
	LANGUAGE EXPERIENCE			X		X	X		X	X	X	X	X
PROJECT ATTRIBUTES	TOOLS AND TECHNIQUES			X		X	X	X		X	X	X	X
	CUSTOMER INTERFACE	X					X				X	X	
	REQUIREMENTS DEFINITION	X			X		X				X	X	X
	REQUIREMENTS VOLATILITY	X			X	X	X		X	X	X	X	X
	SCHEDULE			X		X				X	X	X	X
	SECURITY						X				X	X	
	COMPUTER ACCESS			X	X	X	X	X		X	X	X	X
	TRAVEL/REHOSTING/MULTI-SITE	X			X	X					X	X	X
	SUPPORT SOFTWARE MATURITY									X		X	
CALIBRATION FACTOR				X		X				X			
EFFORT EQUATION	$MM_{NOM} = C(DSI)^X$, X =		1.0		1.047		0.91	1.0		1.05 – 1.2		1.0	1.2
SCHEDULE EQUATION	$t_D = C(MM)^X$, X =						0.35			0.32 – 0.38		0.356	0.333
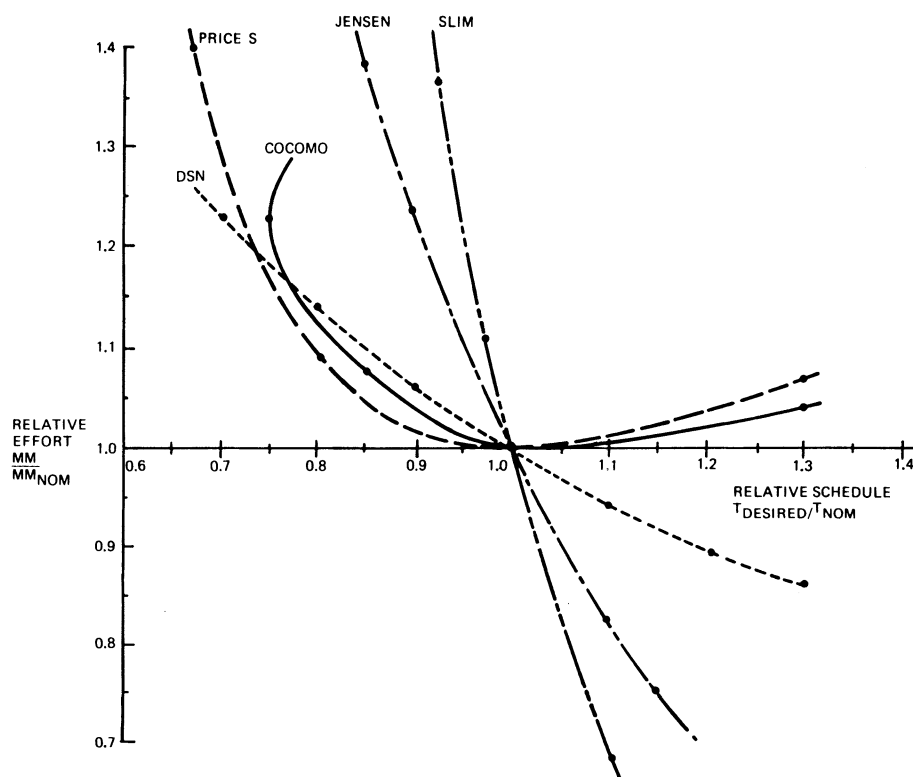
Fig. 5.   Comparative effort-schedule tradeoff relationships.

TABLE III
DOTY MODEL FOR SMALL PROGRAMS*

$$MM = 2.060\ I^{1.047} \prod_{j=1}^{j=14} f_j$$

Factor	f_j	Yes	No
Special display	f_1	1.11	1.00
Detailed definition of operational requirements	f_2	1.00	1.11
Change to operational requirements	f_3	1.05	1.00
Real-time operation	f_4	1.33	1.00
CPU memory constraint	f_5	1.43	1.00
CPU time constraint	f_6	1.33	1.00
First software developed on CPU	f_7	1.92	1.00
Concurrent development of ADP hardware	f_8	1.82	1.00
Timeshare versus batch processing, in development	f_9	0.83	1.00
Developer using computer at another facility	f_{10}	1.43	1.00
Development at operational site	f_{11}	1.39	1.00
Development computer different than target computer	f_{12}	1.25	1.00
Development at more than one site	f_{13}	1.25	1.00
Programmer access to computer	f_{14}	Limited	1.00
		Unlimited	0.90

* Less than 10,000 source instructions

pares with those of other models; see [11, ch. 27] for further discussion of this issue.

On balance, the SLIM approach has provided a number of useful insights into software cost estimation, such as the Rayleigh-curve distribution for one-shot software efforts, the explicit treatment of estimation risk and uncertainty, and the cube-root relationship defining the minimum development time achievable for a project requiring a given amount of effort.

The Doty Model [27]

This model is the result of an extensive data analysis activity, including many of the data points from the SDC sample. A number of models of similar form were developed for different application areas. As an example, the model for general application is

$$MM = 5.288\ (KDSI)^{1.047}, \quad \text{for KDSI} \geqslant 10$$

$$MM = 2.060\ (KDSI)^{1.047} \left(\prod_{j=1}^{14} f_j \right), \quad \text{for KDSI} < 10.$$

The effort multipliers f_i are shown in Table III. This model has a much more appropriate functional form than the SDC model, but it has some problems with stability, as it exhibits a discontinuity at KDSI = 10, and produces widely varying estimates via the f factors (answering "yes" to "first software developed on CPU" adds 92 percent to the estimated cost).

The RCA PRICE S Model [22]

PRICE S is a commercially available (from RCA, Inc.) macro cost-estimation model developed primarily for embedded system applications. It has improved steadily with experience; earlier versions with a widely varying subjective complexity factor have been replaced by versions in which a number of computer, personnel, and project attributes are used to modulate the complexity rating.

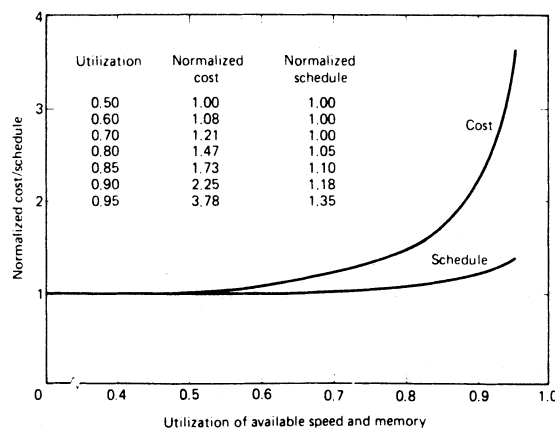| Utilization	Normalized cost	Normalized schedule
0.50	1.00	1.00
0.60	1.08	1.00
0.70	1.21	1.00
0.80	1.47	1.05
0.85	1.73	1.10
0.90	2.25	1.18
0.95	3.78	1.35

Fig. 6.   RCA PRICE S model: Effect of hardware constraints.

PRICE S has extended a number of cost-estimating relationships developed in the early 1970's such as the hardware constraint function shown in Fig. 6 [10]. It was primarily developed to handle military software projects, but now also includes rating levels to cover business applications.

PRICE S also provides a wide range of useful outputs on gross phase and activity distributions analyses, and monthly project cost-schedule-expected progress forecasts. Price S uses a two-parameter beta distribution rather than a Rayleigh curve to calculate development effort distribution versus calendar time.

PRICE S has recently added a software life-cycle support cost estimation capability called PRICE SL [34]. It involves the definition of three categories of support activities.

• *Growth:* The estimator specifies the amount of code to be added to the product. PRICE SL then uses its standard techniques to estimate the resulting life-cycle-effort distribution.

• *Enhancement:* PRICE SL estimates the fraction of the existing product which will be modified (the estimator may

provide his own fraction), and uses its standard techniques to estimate the resulting life-cycle effort distribution.

• *Maintenance:* The estimator provides a parameter indicating the quality level of the developed code. PRICE SL uses this to estimate the effort required to eliminate remaining errors.

The COnstructive COst MOdel (COCOMO) [11]

The primary motivation for the COCOMO model has been to help people understand the cost consequences of the decisions they will make in commissioning, developing, and supporting a software product. Besides providing a software cost estimation capability, COCOMO therefore provides a great deal of material which explains exactly what costs the model is estimating, and why it comes up with the estimates it does. Further, it provides capabilities for sensitivity analysis and tradeoff analysis of many of the common software engineering decision issues.

COCOMO is actually a hierarchy of three increasingly detailed models which range from a single macro-estimation scaling model as a function of product size to a micro-estimation model with a three-level work breakdown structure and a set of phase-sensitive multipliers for each cost driver attribute. To provide a reasonably concise example of a current state of the art cost estimation model, the intermediate level of COCOMO is described below.

Intermediate COCOMO estimates the cost of a proposed software product in the following way.

1) A nominal development effort is estimated as a function of the product's size in delivered source instructions in thousands (KDSI) and the project's development mode.

2) A set of effort multipliers are determined from the product's ratings on a set of 15 cost driver attributes.

3) The estimated development effort is obtained by multiplying the nominal effort estimate by all of the product's effort multipliers.

4) Additional factors can be used to determine dollar costs, development schedules, phase and activity distributions, computer costs, annual maintenance costs, and other elements from the development effort estimate.

Step 1—Nominal Effort Estimation: First, Table IV is used to determine the project's development mode. Organic-mode projects typically come from stable, familiar, forgiving, relatively unconstrained environments, and were found in the COCOMO data analysis of 63 projects have a different scaling equation from the more ambitious, unfamiliar, unforgiving, tightly constrained embedded mode. The resulting scaling equations for each mode are given in Table V; these are used to determine the nominal development effort for the project in man-months as a function of the project's size in KDSI and the project's development mode.

For example, suppose we are estimating the cost to develop the microprocessor-based communications processing software for a highly ambitious new electronic funds transfer network with high reliability, performance, development schedule, and interface requirements. From Table IV, we determine that these characteristics best fit the profile of an embedded-mode project.

We next estimate the size of the product as 10 000 delivered

TABLE IV
COCOMO SOFTWARE DEVELOPMENT MODES

Feature	Mode		
	Organic	Semidetached	Embedded
Organizational understanding of product objectives	Thorough	Considerable	General
Experience in working with related software systems	Extensive	Considerable	Moderate
Need for software conformance with pre-established requirements	Basic	Considerable	Full
Need for software conformance with external interface specifications	Basic	Considerable	Full
Concurrent development of associated new hardware and operational procedures	Some	Moderate	Extensive
Need for innovative data processing architectures, algorithms	Minimal	Some	Considerable
Premium on early completion	Low	Medium	High
Product size range	<50 KDSI	<300 KDSI	All sizes
Examples	Batch data reduction	Most transaction processing systems	Large, complex transaction processing systems
	Scientific models	New OS, DBMS	Ambitious, very large OS
	Business models	Ambitious inventory, production control	Avionics
	Familiar OS, compiler	Simple command-control	Ambitious command-control
	Simple inventory, production control		

TABLE V
COCOMO NOMINAL EFFORT AND SCHEDULE EQUATIONS

DEVELOPMENT MODE	NOMINAL EFFORT	SCHEDULE
Organic	$(MM)_{NOM} = 3.2(KDSI)^{1.05}$	$TDEV = 2.5(MM_{DEV})^{0.38}$
Semidetached	$(MM)_{NOM} = 3.0(KDSI)^{1.12}$	$TDEV = 2.5(MM_{DEV})^{0.35}$
Embedded	$(MM)_{NOM} = 2.8(KDSI)^{1.20}$	$TDEV = 2.5(MM_{DEV})^{0.32}$

(KDSI = thousands of delivered source instructions)

source instructions, or 10 KDSI. From Table V, we then determine that the nominal development effort for this Embedded-mode project is

$$2.8(10)^{1.20} = 44 \text{ man-months (MM)}.$$

Step 2—Determine Effort Multipliers: Each of the 15 cost driver attributes in COCOMO has a rating scale and a set of effort multipliers which indicate by how much the nominal effort estimate must be multiplied to account for the project's having to work at its rating level for the attribute.

These cost driver attributes and their corresponding effort multipliers are shown in Table VI. The summary rating scales for each cost driver attribute are shown in Table VII, except for the complexity rating scale which is shown in Table VIII (expanded rating scales for the other attributes are provided in [11]).

The results of applying these tables to our microprocessor communications software example are shown in Table IX. The effect of a software fault in the electronic fund transfer system could be a serious financial loss; therefore, the project's RELY rating from Table VII is High. Then, from Table VI, the effort multiplier for achieving a High level of required reliability is 1.15, or 15 percent more effort than it would take to develop the software to a nominal level of required reliability.

TABLE VI
INTERMEDIATE COCOMO SOFTWARE DEVELOPMENT EFFORT
MULTIPLIERS

	Ratings					
Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
RELY Required software reliability	.75	.88	1.00	1.15	1.40	
DATA Data base size		.94	1.00	1.08	1.16	
CPLX Product complexity	.70	.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME Execution time constraint			1.00	1.11	1.30	1.66
STOR Main storage constraint			1.00	1.06	1.21	1.56
VIRT Virtual machine volatility*		.87	1.00	1.15	1.30	
TURN Computer turnaround time		.87	1.00	1.07	1.15	
Personnel Attributes						
ACAP Analyst capability	1.46	1.19	1.00	.86	.71	
AEXP Applications experience	1.29	1.13	1.00	.91	.82	
PCAP Programmer capability	1.42	1.17	1.00	.86	.70	
VEXP Virtual machine experience*	1.21	1.10	1.00	.90		
LEXP Programming language experience	1.14	1.07	1.00	.95		
Project Attributes						
MODP Use of modern programming practices	1.24	1.10	1.00	.91	.82	
TOOL Use of software tools	1.24	1.10	1.00	.91	.83	
SCED Required development schedule	1.23	1.08	1.00	1.04	1.10	

* For a given software product, the underlying virtual machine is the complex of hardware and software (OS, DBMS, etc.) it calls on to accomplish its tasks.

TABLE VII
COCOMO SOFTWARE COST DRIVER RATINGS

	Ratings					
Cost Driver	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
RELY	Effect: slight inconvenience	Low, easily recoverable losses	Moderate, recoverable losses	High financial loss	Risk to human life	
DATA		$\frac{DB\ bytes}{Prog.\ DSI} < 10$	$10 \le \frac{D}{P} < 100$	$100 \le \frac{D}{P} < 1000$	$\frac{D}{P} \ge 1000$	
CPLX	See Table 8					
Computer attributes						
TIME			$\le 50\%$ use of available execution time	70%	85%	95%
STOR			$\le 50\%$ use of available storage	70%	85%	95%
VIRT		Major change every 12 months Minor: 1 month	Major: 6 months Minor: 2 weeks	Major: 2 months Minor: 1 week	Major: 2 weeks Minor: 2 days	
TURN		Interactive	Average turnaround <4 hours	4-12 hours	>12 hours	
Personnel attributes						
ACAP	15th percentile*	35th percentile	55th percentile	75th percentile	90th percentile	
AEXP	≤ 4 months experience	1 year	3 years	6 years	12 years	
PCAP	15th percentile*	35th percentile	55th percentile	75th percentile	90th percentile	
VEXP	≤ 1 month experience	4 months	1 year	3 years		
LEXP	≤ 1 month experience	4 months	1 year	3 years		
Project attributes						
MODP	No use	Beginning use	Some use	General use	Routine use	
TOOL	Basic microprocessor tools	Basic mini tools	Basic midi/maxi tools	Strong maxi programming, test tools	Add requirements, design, management, documentation tools	
SCED	75% of nominal	85%	100%	130%	160%	

* Team rating criteria: analysis (programming) ability, efficiency, ability to communicate and cooperate.

TABLE VIII
COCOMO MODULE COMPLEXITY RATINGS VERSUS TYPE OF
MODULE

Rating	Control Operations	Computational Operations	Device-dependent Operations	Data Management Operations
Very low	Straightline code with a few non-nested SP^{a} operators: DOs, CASEs, IFTHENELSEs. Simple predicates	Evaluation of simple expressions: e.g., A = B + C * (D − E)	Simple read, write statements with simple formats	Simple arrays in main memory
Low	Straightforward nesting of SP operators. Mostly simple predicates	Evaluation of moderate-level expressions, e.g., D = SQRT (B**2–4.*A*C)	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET/PUT level. No cognizance of overlap	Single file subsetting with no data structure changes, no edits, no intermediate files
Nominal	Mostly simple nesting. Some inter-module control. Decision tables	Use of standard math and statistical routines. Basic matrix/vector operations	I/O processing includes device selection, status checking and error processing	Multi-file input and single file output. Simple structural changes, simple edits
High	Highly nested SP operators with many compound predicates. Queue and stack control. Considerable intermodule control.	Basic numerical analysis: multivariate interpolation, ordinary differential equations. Basic truncation, roundoff concerns	Operations at physical I/O level (physical storage address translations; seeks, reads, etc). Optimized I/O overlap	Special purpose subroutines activated by data stream contents. Complex data restructuring at record level
Very high	Reentrant and recursive coding. Fixed-priority interrupt handling	Difficult but structured N.A.: near-singular matrix equations, partial differential equations	Routines for interrupt diagnosis, servicing, masking. Communication line handling	A generalized, parameter-driven file structuring routine. File building, command processing, search optimization
Extra high	Multiple resource scheduling with dynamically changing priorities. Microcode-level control	Difficult and unstructured N.A.: highly accurate analysis of noisy, stochastic data	Device timing-dependent coding, micro-programmed operations	Highly coupled, dynamic relational structures. Natural language data management

a SP = structured programming

TABLE IX
COCOMO COST DRIVER RATINGS: MICROPROCESSOR
COMMUNICATIONS SOFTWARE

Cost Driver	Situation	Rating	Effort Multiplier
RELY	Serious financial consequences of software faults	High	1.15
DATA	20,000 bytes	Low	0.94
CPLX	Communications processing	Very High	1.30
TIME	Will use 70% of available time	High	1.11
STOR	45K of 64K store (70%)	High	1.06
VIRT	Based on commercial microprocessor hardware	Nominal	1.00
TURN	Two-hour average turnaround time	Nominal	1.00
ACAP	Good senior analysts	High	0.86
AEXP	Three years	Nominal	1.00
PCAP	Good senior programmers	High	0.86
VEXP	Six months	Low	1.10
LEXP	Twelve months	Nominal	1.00
MODP	Most techniques in use over one year	High	0.91
TOOL	At basic minicomputer tool level	Low	1.10
SCED	Nine months	Nominal	1.00
	Effort adjustment factor (product of effort multipliers)		1.35

The effort multipliers for the other cost driver attributes are obtained similarly, except for the Complexity attribute, which is obtained via Table VIII. Here, we first determine that communications processing is best classified under device-dependent operations (column 3 in Table VIII). From this column, we determine that communication line handling typically has a complexity rating of Very High; from Table VI, then, we determine that its corresponding effort multiplier is 1.30.

Step 3—Estimate Development Effort: We then compute the estimated development effort for the microprocessor communications software as the nominal development effort (44 MM) times the product of the effort multipliers for the 15 cost driver attributes in Table IX (1.35, in Table IX). The resulting estimated effort for the project is then

(44 MM) (1.35) = 59 MM.

Step 4—Estimate Related Project Factors: COCOMO has additional cost estimating relationships for computing the resulting dollar cost of the project and for the breakdown of cost and effort by life-cycle phase (requirements, design, etc.) and by type of project activity (programming, test planning, management, etc.). Further relationships support the estimation of the project's schedule and its phase distribution. For example, the recommended development schedule can be obtained from the estimated development man-months via the embedded-mode schedule equation in Table V:

$$T_{DEV} = 2.5(59)^{0.32} = 9 \text{ months.}$$

As mentioned above, COCOMO also supports the most common types of sensitivity analysis and tradeoff analysis involved in scoping a software project. For example, from Tables VI and VII, we can see that providing the software developers with an interactive computer access capability (Low turnaround time) reduces the TURN effort multiplier from 1.00 to 0.87, and thus reduces the estimated project effort from 59 MM to

(59 MM) (0.87) = 51 MM.

The COCOMO model has been validated with respect to a sample of 63 projects representing a wide variety of business, scientific, systems, real-time, and support software projects. For this sample, Intermediate COCOMO estimates come within 20 percent of the actuals about 68 percent of the time (see Fig. 7). Since the residuals roughly follow a normal distribution, this is equivalent to a standard deviation of roughly 20 percent of the project actuals. This level of accuracy is representative of the current state of the art in software cost models. One can do somewhat better with the aid of a calibration coefficient (also a COCOMO option), or within a limited applications context, but it is difficult to improve significantly on this level of accuracy while the accuracy of software data collection remains in the "±20 percent" range.

A Pascal version of COCOMO is available for a nominal distribution charge from the Wang Institute, under the name WI-COMO [18].

Recent Software Cost Estimation Models

Most of the recent software cost estimation models tend to follow the Doty and COCOMO models in having a nominal
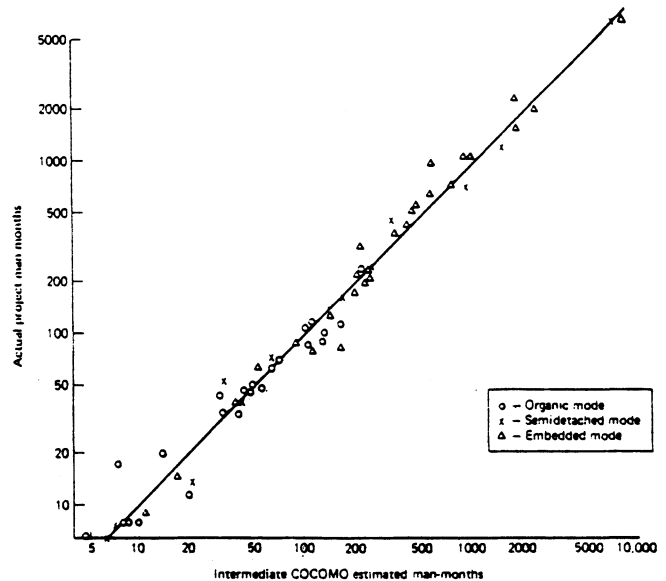

Fig. 7.   Intermediate COCOMO estimates versus project actuals.

scaling equation of the form $MM_{NOM} = c(KDSI)^x$ and a set of multiplicative effort adjustment factors determined by a number of cost driver attribute ratings. Some of them use the Rayleigh curve approach to estimate distribution across the software life-cycle, but most use a more conservative effort/schedule tradeoff relation than the SLIM model. These aspects have been summarized for the various models in Table II and Fig. 5.

The Bailey-Basili meta-model [4] derived the scaling equation

$$MM_{NOM} = 3.5 + 0.73 \text{ (KDSI)}^{1.16}$$

and used two additional cost driver attributes (methodology level and complexity) to model the development effort of 18 projects in the NASA-Goddard Software Engineering Laboratory to within a standard deviation of 15 percent. Its accuracy for other project situations has not been determined.

The Grumman SOFCOST Model [19] uses a similar but unpublished nominal effort scaling equation, modified by 30 multiplicative cost driver variables rated on a scale of 0 to 10. Table II includes a summary of these variables.

The Tausworthe Deep Space Network (DSN) model [50] uses a linear scaling equation $(MM_{NOM} = a(KDSI)^{1.0})$ and a similar set of cost driver attributes, also summarized in Table II. It also has a well-considered approach for determining the equivalent KDSI involved in adapting existing software within a new product. It uses the Rayleigh curve to determine the phase distribution of effort, but uses a considerably more conservative version of the SLIM effort-schedule tradeoff relationship (see Fig. 5).

The Jensen model [30], [31] is a commercially available model with a similar nominal scaling equation, and a set of cost driver attributes very similar to the Doty and COCOMO models (but with different effort multiplier ranges); see Table II. Some of the multiplier ranges in the Jensen model vary as functions of other factors; e.g., increasing access to computer resources widens the multiplier ranges on such cost drivers as personnel capability and use of software tools. It uses the Rayleigh curve for effort distribution, and a somewhat more conservative ef-

fort-schedule tradeoff relation than SLIM (see Fig. 5). As with the other commercial models, the Jensen model produces a number of useful outputs on resource expenditure rates, probability distributions on costs and schedules, etc.

C. Outstanding Research Issues in Software Cost Estimation

Although a good deal of progress has been made in software cost estimation, a great deal remains to be done. This section updates the state-of-the-art review published in [11], and summarizes the outstanding issues needing further research:

1) Software size estimation;
2) Software size and complexity metrics;
3) Software cost driver attributes and their effects;
4) Software cost model analysis and refinement;
5) Quantitative models of software project dynamics;
6) Quantitative models of software life-cycle evolution;
7) Software data collection.

1) Software Size Estimation: The biggest difficulty in using today's algorithmic software cost models is the problem of providing sound sizing estimates. Virtually every model requires an estimate of the number of source or object instructions to be developed, and this is an extremely difficult quantity to determine in advance. It would be most useful to have some formula for determining the size of a software product in terms of quantities known early in the software life cycle, such as the number and/or size of the files, input formats, reports, displays, requirements specification elements, or design specification elements.

Some useful steps in this direction are the function-point approach in [2] and the sizing estimation model of [29], both of which have given reasonably good results for small-to-medium sized business programs within a single data processing organization. Another more general approach is given by DeMarco in [17]. It has the advantage of basing its sizing estimates on the properties of specifications developed in conformance with DeMarco's paradigm models for software specifications and designs: number of functional primitives, data elements, input elements, output elements, states, transitions between states, relations, modules, data tokens, control tokens, etc. To date, however, there has been relatively little calibration of the formulas to project data. A recent IBM study [14] shows some correlation between the number of variables defined in a state-machine design representation and the product size in source instructions.

Although some useful results can be obtained on the software sizing problem, one should not expect too much. A wide range of functionality can be implemented beneath any given specification element or I/O element, leading to a wide range of sizes (recall the uncertainty ranges of this nature in Fig. 3). For example, two experiments, involving the use of several teams developing a software program to the same overall functional specification, yielded size ranges of factors of 3 to 5 between programs (see Table X).

The primary implication of this situation for practical software sizing and cost estimation is that *there is no royal road to software sizing*. This is no magic formula that will provide an easy and accurate substitute for the process of thinking through and fully understanding the nature of the software product to be developed. There are still a number of useful

TABLE X
SIZE RANGES OF SOFTWARE PRODUCTS PERFORMING SAME FUNCTION

Experiment	Product	No. of Teams	Size Range (source-instr.)
Weinberg & Schulman [55]	Simultaneous linear equations	6	33-165
Boehm, Gray, & Seewaldt [13]	Interactive cost model	7	1514-4606

things that one can do to improve the situation, including the following.

• Use techniques which explicitly recognize the ranges of variability in software sizing. The PERT estimation technique [56] is a good example.

• Understand the primary sources of bias in software sizing estimates. See [11, ch. 21].

• Develop and use a corporate memory on the nature and size of previous software products.

2) Software Size and Complexity Metrics: Delivered source instructions (DSI) can be faulted for being too low-level a metric for use in early sizing estimation. On the other hand, DSI can also be faulted for being too high-level a metric for precise software cost estimation. Various complexity metrics have been formulated to more accurately capture the relative information content of a program's instructions, such as the Halstead Software Science metrics [24], or to capture the relative control complexity of a program, such as the metrics formulated by McCabe in [39]. A number of variations of these metrics have been developed; a good recent survey of them is given in [26].

However, these metrics have yet to exhibit any practical superiority to DSI as a predictor of the relative effort required to develop software. Most recent studies [48], [32] show a reasonable correlation between these complexity metrics and development effort, but no better a correlation than that between DSI and development effort.

Further, the recent [25] analysis of the software science results indicates that many of the published software science "successes" were not as successful as they were previously considered. It indicates that much of the apparent agreement between software science formulas and project data was due to factors overlooked in the data analysis: inconsistent definitions and interpretations of software science quantities, unrealistic or inconsistent assumptions about the nature of the projects analyzed, overinterpretation of the significance of statistical measures such as the correlation coefficient, and lack of investigation of alternative explanations for the data. The software science use of psychological concepts such as the Stroud number have also been seriously questioned in [16].

The overall strengths and difficulties of software science are summarized in [47]. Despite the difficulties, some of the software science metrics have been useful in such areas as identifying error-prone modules. In general, there is a strong intuitive argument that more definitive complexity metrics will eventually serve as better bases for definitive software cost estimation than will DSI. Thus, the area continues to be an attractive one for further research.

3) Software Cost Driver Attributes and Their Effects: Most of the software cost models discussed above contain a selection of cost driver attributes and a set of coefficients, functions, or tables representing the effect of the attribute on software cost (see Table II). Chapters 24-28 of [11] contain summaries of the research to date on about 20 of the most significant cost driver attributes, plus statements of nearly 100 outstanding research issues in the area.

Since the publication of [11] in 1981, a few new results have appeared. Lawrence [35] provides an analysis of 278 business data processing programs which indicate a fairly uniform development rate in procedure lines of code per hour, some significant effects on programming rate due to batch turnaround time and level of experience, and relatively little effect due to use of interactive operation and modern programming practices (due, perhaps, to the relatively repetitive nature of the software jobs sampled). Okada and Azuma [42] analyzed 30 CAD/CAM programs and found some significant effects due to type of software, complexity, personnel skill level, and requirements volatility.

4) Software Cost Model Analysis and Refinement: The most useful comparative analysis of software cost models to date is the Thibodeau [52] study performed for the U.S. Air Force. This study compared the results of several models (the Wolverton, Doty, PRICE S, and SLIM models discussed earlier, plus models from the Boeing, SDC, Tecolote, and Aerospace corporations) with respect to 45 project data points from three sources.

Some generally useful comparative results were obtained, but the results were not definitive, as models were evaluated with respect to larger and smaller subsets of the data. Not too surprisingly, the best results were generally obtained using models with calibration coefficients against data sets with few points. In general, the study concluded that the models with calibration coefficients achieved better results, but that none of the models evaluated were sufficiently accurate to be used as a definitive Air Force software cost estimation model.

Some further comparative analyses are currently being conducted by various organizations, using the database of 63 software projects in [11], but to date none of these have been published.

In general, such evaluations play a useful role in model refinement. As certain models are found to be inaccurate in certain situations, efforts are made to determine the causes, and to refine the model to eliminate the sources of inaccuracy.

Relatively less activity has been devoted to the formulation, evaluation, and refinement of models to cover the effects of more advanced methods of software development (prototyping, incremental development, use of application generators, etc.) or to estimate other software-related life-cycle costs (conversion, maintenance, installation, training, etc.). An exception is the excellent work on software conversion cost estimation performed by the Federal Conversion Support Center [28]. An extensive model to estimate avionics software support costs using a weighted-multiplier technique has recently been developed [49]. Also, some initial experimental results have been obtained on the quantitative impact of prototyping in [13] and on the impact of very high level nonprocedural languages in [58]. In both studies, projects using prototyping and VHLL's were completed with significantly less effort.

5) Quantitative Models of Software Project Dynamics: Current software cost estimation models are limited in their ability to represent the internal dynamics of a software project, and to estimate how the project's phase distribution of effort and schedule will be affected by environmental or project management factors. For example, it would be valuable to have a model which would accurately predict the effort and schedule distribution effects of investing in more thorough design verification, of pursuing an incremental development strategy, of varying the staffing rate or experience mix, of reducing module size, etc.

Some current models assume a universal effort distribution, such as the Rayleigh curve [44] or the activity distributions in [57], which are assumed to hold for any type of project situation. Somewhat more realistic, but still limited are models with phase-sensitive effort multipliers such as PRICE S [22] and Detailed COCOMO [11].

Recently, some more realistic models of software project dynamics have begun to appear, although to date none of them have been calibrated to software project data. The Phister phase-by-phase model in [43] estimates the effort and schedule required to design, code, and test a software product as a function of such variables as the staffing level during each phase, the size of the average module to be developed, and such factors as interpersonal communications overhead rates and error detection rates. The Abdel Hamid–Madnick model [1], based on Forrester's System Dynamics world-view, estimates the time distribution of effort, schedule, and residual defects as a function of such factors as staffing rates, experience mix, training rates, personnel turnover, defect introduction rates, and initial estimation errors. Tausworthe [51] derives and calibrates alternative versions of the SLIM effort-schedule tradeoff relationship, using an intercommunication-overhead model of project dynamics. Some other recent models of software project dynamics are the Mitre SWAP model and the Duclos [21] total software life-cycle model.

6) Quantitative Models of Software Life-Cycle Evolution: Although most of the software effort is devoted to the software maintenance (or life-cycle support) phase, only a few significant results have been obtained to date in formulating quantitative models of the software life-cycle evolution process. Some basic studies by Belady and Lehman analyzed data on several projects and derived a set of fairly general "laws of program evolution" [7], [37]. For example, the first of these laws states:

> "A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system with a re-created version."

Some general quantitative support for these laws was obtained in several studies during the 1970's, and in more recent studies such as [33]. However, efforts to refine these general laws into a set of testable hypotheses have met with mixed results. For

example, the Lawrence [36] statistical analysis of the Belady-Lahman data showed that the data supported an even stronger form of the first law ("systems grow in size over their useful life"); that one of the laws could not be formulated precisely enough to be tested by the data; and that the other three laws did not lead to hypotheses that were supported by the data.

However, it is likely that variant hypotheses can be found that are supported by the data (for example, the operating system data supports some of the hypotheses better than does the applications data). Further research is needed to clarify this important area.

7) Software Data Collection: A fundamental limitation to significant progress in software cost estimation is the lack of unambiguous, widely-used standard definitions for software data. For example, if an organization reports its "software development man-months," do these include the effort devoted to requirements analysis, to training, to secretaries, to quality assurance, to technical writers, to uncompensated overtime? Depending on one's interpretations, one can easily cause variations of over 20 percent (and often over a factor of 2) in the meaning of reported "software development man-months" between organizations (and similarly for "delivered instructions," "complexity," "storage constraint," etc.) Given such uncertainties in the ground data, it is not surprising that software cost estimation models cannot do much better than "within 20 percent of the actuals, 70 percent of the time."

Some progress towards clear software data definitions has been made. The IBM FSD database used in [53] was carefully collected using thorough data definitions, but the detailed data and definitions are not generally available. The NASA-Goddard Software Engineering Laboratory database [5], [6], [40] and the COCOMO database [11] provide both clear data definitions and an associated project database which are available for general use (and reasonably compatible). The recent Mitre SARE report [59] provides a good set of data definitions.

But there is still no commitment across organizations to establish and use a set of clear and uniform software data definitions. Until this happens, our progress in developing more precise software cost estimation methods will be severely limited.

IV. SOFTWARE ENGINEERING ECONOMICS BENEFITS AND CHALLENGES

This final section summarizes the benefits to software engineering and software management provided by a software engineering economics perspective in general and by software cost estimation technology in particular. It concludes with some observations on the major challenges awaiting the field.

Benefits of a Software Engineering Economics Perspective

The major benefit of an economic perspective on software engineering is that it provides a balanced view of candidate software engineering solutions, and an evaluation framework which takes account not only of the programming aspects of a situation, but also of the human problems of providing the best possible information processing service within a resource-limited environment. Thus, for example, the software engineering economics approach does not say, "we should use

these structured structures because they are mathematically elegant" or "because they run like the wind" or "because they are part of the structured revolution." Instead, it says "we should use these structured structures because they provide people with more benefits in relation to their costs than do other approaches." And besides the framework, of course, it also provides the techniques which help us to arrive at this conclusion.

Benefits of Software Cost Estimation Technology

The major benefit of a good software cost estimation model is that it provides a clear and consistent universe of discourse within which to address a good many of the software engineering issues which arise throughout the software life cycle. It can help people get together to discuss such issues as the following.

- Which and how many features should we put into the software product?
- Which features should we put in first?
- How much hardware should we acquire to support the software product's development, operation, and maintenance?
- How much money and how much calendar time should we allow for software development?
- How much of the product should we adapt from existing software?
- How much should we invest in tools and training?

Further, a well-defined software cost estimation model can help avoid the frequent misinterpretations, underestimates, overexpectations, and outright buy-ins which still plague the software field. In a good cost-estimation model, there is no way of reducing the estimated software cost without changing some objectively verifiable property of the software project. This does not make it impossible to create an unachievable buy-in, but it significantly raises the threshold of credibility.

A related benefit of software cost estimation technology is that it provides a powerful set of insights on how a software organization can improve its productivity. Many of a software cost model's cost-driver attributes are management controllables: use of software tools and modern programming practices, personnel capability and experience, available computer speed, memory, and turnaround time, software reuse. The cost model helps us determine how to adjust these management controllables to increase productivity, and further provides an estimate of how much of a productivity increase we are likely to achieve with a given level of investment. For more information on this topic, see [11, ch. 33], [12] and the recent plan for the U.S. Department of Defense Software Initiative [20].

Finally, software cost estimation technology provides an absolutely essential foundation for software project planning and control. Unless a software project has clear definitions of its key milestones and realistic estimates of the time and money it will take to achieve them, there is no way that a project manager can tell whether his project is under control or not. A good set of cost and schedule estimates can provide realistic data for the PERT charts, work breakdown structures, manpower schedules, earned value increments, etc., necessary to establish management visibility and control.

Note that this opportunity to improve management visibility and control requires a complementary management com-

mitment to define and control the reporting of data on software progress and expenditures. The resulting data are therefore worth collecting simply for their management value in comparing plans versus achievements, but they can serve another valuable function as well: they provide a continuing stream of calibration data for evolving a more accurate and refined software cost estimation models.

Software Engineering Economics Challenges

The opportunity to improve software project management decision making through improved software cost estimation, planning, data collection, and control brings us back full-circle to the original objectives of software engineering economics: to provide a better quantitative understanding of how software people make decisions in resource-limited situations.

The more clearly we as software engineers can understand the quantitative and economic aspects of our decision situations, the more quickly we can progress from a pure seat-of-the-pants approach on software decisions to a more rational approach which puts all of the human and economic decision variables into clear perspective. Once these decision situations are more clearly illuminated, we can then study them in more detail to address the deeper challenge: achieving a quantitative understanding of how people work together in the software engineering process.

Given the rather scattered and imprecise data currently available in the software engineering field, it is remarkable how much progress has been made on the software cost estimation problem so far. But, there is not much further we can go until better data becomes available. The software field cannot hope to have its Kepler or its Newton until it has had its army of Tycho Brahes, carefully preparing the well-defined observational data from which a deeper set of scientific insights may be derived.

REFERENCES

[1] T. K. Abdel-Hamid and S. E. Madnick, "A model of software project management dynamics," in *Proc. IEEE COMPSAC 82*, Nov. 1982, pp. 539–554.
[2] A. J. Albrecht, "Measuring Application Development Productivity," in *SHARE-GUIDE*, 1979, pp. 83–92.
[3] J. D. Aron, "Estimating resources for large programming systems." NATO Sci. Committee, Rome, Italy, Oct. 1969.
[4] J. J. Bailey and V. R. Basili, "A meta-model for software development resource expenditures," in *Proc. 5th Int. Conf. Software Eng.*, IEEE/ACM/NBS, Mar. 1981, pp. 107–116.
[5] V. R. Basili, "Tutorial on models and metrics for software and engineering," IEEE Cat. EHO-167-7, Oct. 1980.
[6] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," Univ. Maryland Technol. Rep. TR-1235, Dec. 1982.
[7] L. A. Belady and M. M. Lehman, "Characteristics of large systems," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: MIT Press, 1979.
[8] H. D. Benington, "Production of large computer programs," in *Proc. ONR Symp. Advanced Programming Methods for Digital Computers*, June 1956, pp. 15–27.
[9] R. K. D. Black, R. P. Curnow, R. Katz, and M. D. Gray, "BCS software production data," Boeing Comput. Services, Inc., Final Tech. Rep., RADC-TR-77-116, NTIS AD-A039852, Mar. 1977.
[10] B. W. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, pp. 48–59, May 1973.
[11] ——, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[12] B. W. Boehm, J. F. Elwell, A. B. Pyster, E. D. Stuckle, and R. D. Williams, "The TRW software productivity system," in *Proc. IEEE 6th Int. Conf. Software Eng.*, Sept. 1982.

[13] B. W. Boehm, T. E. Gray, and T. Seewaldt, "Prototyping vs. specifying: A multi-project experiment," *IEEE Trans. Software Eng.*, to be published.
[14] R. N. Britcher and J. E. Gaffney, "Estimates of software size from state machine designs," in *Proc. NASA-Goddard Software Eng. Workshop*, Dec. 1982.
[15] W. M. Carriere and R. Thibodeau, "Development of a logistics software cost estimating technique for foreign military sales," General Res. Corp., Rep. CR-3-839, June 1979.
[16] N. S. Coulter, "Software science and cognitive psychology," *IEEE Trans. Software Eng.*, pp. 166–171, Mar. 1983.
[17] T. DeMarco, *Controlling Software Projects*. New York: Yourdon, 1982.
[18] M. Demshki, D. Ligett, B. Linn, G. McCluskey, and R. Miller, "Wang Institute cost model (WICOMO) tool user's manual," Wang Inst. Graduate Studies, Tyngsboro, MA, June 1982.
[19] H. F. Dircks, "SOFCOST: Grumman's software cost eliminating model," in *IEEE NAECON 1981*, May 1981.
[20] L. E. Druffel, "Strategy for DoD software initiative," RADC/DACS, Griffiss AFB, NY, Oct. 1982.
[21] L. C. Duclos, "Simulation model for the life-cycle of a software product: A quality assurance approach," Ph.D. dissertation, Dep. Industrial and Syst. Eng., Univ. Southern California, Dec. 1982.
[22] F. R. Freiman and R. D. Park, "PRICE software model—Version 3: An overview," in *Proc. IEEE-PINY Workshop on Quantitative Software Models*, IEEE Cat. TH0067-9, Oct. 1979, pp. 32–41.
[23] R. Goldberg and H. Lorin, *The Economics of Information Processing*. New York: Wiley, 1982.
[24] M. H. Halstead, *Elements of Software Science*. New York: Elsevier, 1977.
[25] P. G. Hamer and G. D. Frewin, "M. H. Halstead's software science—A critical examination," in *Proc. IEEE 6th Int. Conf. Software Eng.*, Sept. 1982, pp. 197–205.
[26] W. Harrison, K. Magel, R. Kluczney, and A. DeKock, "Applying software complexity metrics to program maintenance," *Computer*, pp. 65–79, Sept. 1982.
[27] J. R. Herd, J. N. Postak, W. E. Russell, and K. R. Stewart, "Software cost estimation study—Study results," Doty Associates, Inc., Rockville, MD, Final Tech. Rep. RADC-TR-77-220, vol. 1 (of two), June 1977.
[28] C. Houtz and T. Buschbach, "Review and analysis of conversion cost-estimating techniques," GSA Federal Conversion Support Center, Falls Church, VA, Rep. GSA/FCSC-81/001, Mar. 1981.
[29] M. Itakura and A. Takayanagi, " A model for estimating program size and its evaluation," in *Proc. IEEE 6th Software Eng.*, Sept. 1982, pp. 104–109.
[30] R. W. Jensen, "An improved macrolevel software development resource estimation model," in *Proc. 5th ISPA Conf.*, Apr. 1983, pp. 88–92.
[31] R. W. Jensen and S. Lucas, "Sensitivity analysis of the Jensen software model," in *Proc. 5th ISPA Conf.*, Apr. 1983, pp. 384–389.
[32] B. A. Kitchenham, "Measures of programming complexity," *ICL Tech. J.*, pp. 298–316, May 1981.
[33] ——, "Systems evolution dynamics of VME/B," *ICL Tech. J.*, pp. 43–57, May 1982.
[34] W. W. Kuhn, "A software lifecycle case study using the PRICE model," in *Proc. IEEE NAECON*, May 1982.
[35] M. J. Lawrence, "Programming methodology, organizational environment, and programming productivity," *J. Syst. Software*, pp. 257–270, Sept. 1981.
[36] ——, "An examination of evolution dynamics," in *Proc. IEEE 6th Int. Conf. Software Eng.*, Sept. 1982, pp. 188–196.
[37] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proc. IEEE*, pp. 1060–1076, Sept. 1980.
[38] R. D. Luce and H. Raiffa, *Games and Decisions*. New York: Wiley, 1957.
[39] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, pp. 308–320, Dec. 1976.
[40] F. E. McGarry, "Measuring software development technology: What have we learned in six years," in *Proc. NASA-Goddard Software Eng. Workshop*, Dec. 1982.
[41] E. A. Nelson, "Management handbook for the estimation of computer programming costs," Syst. Develop. Corp., AD-A648750, Oct. 31, 1966.
[42] M. Okada and M. Azuma, "Software development estimation study—A model from CAD/CAM system development experiences," in *Proc. IEEE COMPSAC 82*, Nov. 1982, pp. 555–564.

[43] M. Phister, Jr., "A model of the software development process," *J. Syst. Software*, pp. 237–256, Sept. 1981.

[44] L. H. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *IEEE Trans. Software Eng.*, pp. 345–361, July 1978.

[45] L. H. Putnam and A. Fitzsimmons, "Estimating software costs," *Datamation*, pp. 189–198, Sept. 1979; continued in *Datamation*, pp. 171–178, Oct. 1979 and pp. 137–140, Nov. 1979.

[46] L.H. Putnam, "The real economics of software development," in *The Economics of Information Processing*, R. Goldberg and H. Lorin. New York: Wiley, 1982.

[47] V. Y. Shen, S. D. Conte, and H. E. Dunsmore, "Software science revisited: A critical analysis of the theory and its empirical support," *IEEE Trans. Software Eng.*, pp. 155–165, Mar. 1983.

[48] T. Sunohara, A. Takano, K. Uehara, and T. Ohkawa, "Program complexity measure for software development management," in *Proc. IEEE 5th Int. Conf. Software Eng.*, Mar. 1981, pp. 100–106.

[49] SYSCON Corp., "Avionics software support cost model," USAF Avionics Lab., AFWAL-TR-1173, Feb. 1, 1983.

[50] R. C. Tausworthe, "Deep space network software cost estimation model," Jet Propulsion Lab., Pasadena, CA, 1981.

[51] ——, "Staffing implications of software productivity models," in *Proc. 7th Annu. Software Eng. Workshop*, NASA/Goddard, Greenbelt, MD, Dec. 1982.

[52] R. Thibodeau, "An evaluation of software cost estimating models," General Res. Corp., Rep. T10-2670, Apr. 1981.

[53] C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, no. 1, pp. 54–73, 1977.

[54] G. F. Weinwurm, Ed., *On the Management of Computer Programming*. New York: Auerbach, 1970.

[55] G. M. Weinberg and E. L. Schulman, "Goals and performance in computer programming," *Human Factors*, vol. 16, no. 1, pp. 70–77, 1974.

[56] J. D. Wiest and F. K. Levy, *A Management Guide to PERT/CPM*. Englewood Cliffs, NJ: Prentice-Hall, 1977.

[57] R. W. Wolverton, "The cost of developing large-scale software," *IEEE Trans. Comput.*, pp. 615–636, June 1974.

[58] E. Harel and E. R. McLean, "The effects of using a nonprocedural computer language on programmer productivity," UCLA Inform. Sci. Working Paper 3-83, Nov. 1982.

[59] R. L. Dumas, "Final report: Software acquisition resource expenditure (SARE) data collection methodology," MITRE Corp., MTR 9031, Sept. 1983.

Barry W. Boehm received the B.A. degree in mathematics from Harvard University, Cambridge, MA, in 1957 and the M.A. and Ph.D. degrees from the University of California, Los Angeles, in 1961 and 1964, respectively.

From 1978 to 1979 he was a Visiting Professor of Computer Science at the University of Southern California. He is currently a Visiting Professor at the University of California, Los Angeles, and Chief Engineer of TRW's Software Information Systems Division. He was previously Head of the Information Sciences Department at The Rand Corporation, and Director of the 1971 Air Force CCIP-85 study. His responsibilities at TRW include direction of TRW's internal software R&D program, of contract software technology projects, of the TRW software development policy and standards program, of the TRW Software Cost Methodology Program, and the TRW Software Productivity Program. His most recent book is *Software Engineering Economics*, by Prentice-Hall.

Dr. Boehm is a member of the IEEE Computer Society and the Association for Computing Machinery, and an Associate Fellow of the American Institute of Aeronautics and Astronautics.