The Cost of Developing Large-Scale Software

RAY W. WOLVERTON, MEMBER, IEEE

Abstract—The work of software cost forecasting falls into two parts. First we make what we call structural forecasts, and then we calculate the absolute dollar-volume forecasts. Structural forecasts describe the technology and function of a software project, but not its size. We allocate resources (costs) over the project's life cycle from the structural forecasts. Judgment, technical knowledge, and econometric research should combine in making the structural forecasts. A methodology based on a $25 \times 7$ structural forecast matrix that has been used by TRW with good results over the past few years is presented in this paper. With the structural forecast in hand, we go on to calculate the absolute dollar-volume forecasts. The general logic followed in "absolute" cost estimating can be based on either a mental process or an explicit algorithm. A cost estimating algorithm is presented and five tradition methods of software cost forecasting are described: top-down estimating, similarities and differences estimating, ratio estimating, standards estimating, and bottom-up estimating. All forecasting methods suffer from the need for a valid cost data base for many estimating situations. Software information elements that experience has shown to be useful in establishing such a data base are given in the body of the paper. Major pricing pitfalls are identified. Two case studies are presented that illustrate the software cost forecasting methodology and historical results. Topics for further work and study are suggested.

Index Terms—Computer programmer code productivity rates, cost database used in cost estimation mechanism, cost of developing custom software, incentive fee structure influence on computer software development, principles of pricing computer software proposals, resource allocation in computer program development, software cost estimating methods and general logic, software cost estimation algorithm, software development and test life cycle—cost impact, systems approach to pricing large-scale software projects.

INTRODUCTION

Estimating the cost of a large-scale computer program has traditionally been a risky undertaking. Now that software has been with us for nearly two decades, it is reasonable to hope that we have learned something that would make our predictions less unreliable. At TRW we have been motivated to make a serious effort toward improving our software estimating methods for large-scale software systems, for a set of very compelling reasons.

1) Our customers have shown a growing unwillingness to accept cost and schedule overruns unless the penalties were increasingly borne by the software developer.

2) Partly as a consequence of 1), we have entered into software development contracts where both adherence to predicted costs and on-time delivery were incentivized.

That means we made more money if we could predict accurately the cost and the time it would take to do the job.

3) We found that we could improve our estimates only by improving our understanding of exactly what steps and processes were involved in software development, and this understanding enabled us to manage the effort better. The better management, in turn, improved our estimates.

This paper presents the essential results of our efforts to improve our software cost estimating techniques. It should be understood that the specific contracts that are used here were government contracts whose particular provisions shaped those efforts to some degree. Nevertheless, we feel that much of what we learned is of general interest and can be applied or adapted to nearly any software development program of any size.

We can start with a review of some of the ways in which software development differs from hardware development, and which have traditionally contributed to the problem of estimating software costs. One of these ways is the problem of managing the people. The nature of programmers is such that interesting work gets done at the expense of dull work, and documentation is dull work. Doing the job in a clever way tends to be a more important consideration than getting it done adequately, on time, and at reasonable cost. Programmers tend to be optimistic, not realistic, and their time estimates for task completion reflect this tendency. The software engineer, the mathematician, and the programmer have trouble communicating. Visible signs of programming progress are almost totally lacking.

Another set of difficulties arises from the nature of the product. There are virtually no objective standards or measures by which to evaluate the progress of computer program development. Software characteristics are often in conflict, requiring frequent tradeoffs among such factors as core storage requirement versus tight code, fast execution time, external storage to minimize I/O time, maintainability by the eventual user, flexibility or adaptability to new needs, accuracy and reliability, and self-check and fail-safe modes of operation. The applications software developer finds himself in a dynamically evolving and constantly changing environment.

Software planning has problems stemming from the natural desire to get going, which means taking shortcuts. What are the steps that should precede the start of coding, and how does the knowledgeable manager allocate his limited resources? Planning also is made difficult by the
proposals, software that level system to assure that all problems, deficiencies, change
user software and demonstration, acceptance management control is established reflecting costs.
resources over the cycle (and the detail system.
and, established problem solving, the software development manager will spend much of his
time solving problems that need not have arisen. Software is controlled through control of documentation within a structured software development process.

There are several ways of analyzing the software development process, but the one we have chosen leads to the definition of seven phases, or steps, each ending in a discrete event or document (see Fig. 1). The steps are basically sequential, but in practice there are iterations between adjacent steps and even between any of the steps. The seven steps are as follows.

Step 1—Performance and Design Requirements: This document establishes the requirements for performance, design, test, and qualification of the software at the system level.

Step 2—Implementation Concept and Test Plan: This document provides, at an early date, the preliminary design concepts that meet the requirements, and identifies the preferred approaches, design tradeoffs, and alternatives, and leaves no high-risk technology area unresolved.

Step 3—Interface and Data Requirements Specification: This document defines the interfaces between subsystems and major elements within subsystems, including presentation of table and file structure, data origin, destination, and update characteristics.

Step 4—Detailed Design Specification: This document provides, in precise detail, the complete design and acceptance specifications for the software at the routine level, which includes detailed flow charts, equations, and logic. It is the source material from which the program is coded.

Step 5—Coding and Debugging: Upon design review and approval of the Step 4 documentation package, which may be staggered in time and organized into volumes dealing with a single computer program module, the coding of the software can begin. This step is devoted to coding and debugging (and amending documentation), using debugging aids and desk check of the first compilations. The phase terminates for a given module when it has passed the development-level verification tests imposed by the designer, and he turns the module over for assembly onto the master tape for the beginning of independent system validation testing.

Step 6—System Validation Testing: As the software
modules are released by the design group for assembly onto the master tape, the buildup of the software package into successively more complex interfaces and capabilities can commence. When two or more modules are assembled, the first stages of "system testing" can get underway according to the test plan started at Step 2 and upgraded in level of detail at Step 4 to include test procedures and quantitative acceptance criteria. The phase terminates when all modules have been successfully tested against the predetermined acceptance criteria at the system level.

Step 7—Certification and Acceptance Demonstration: This final step is devoted to formal acceptance of the software system by subjecting it to previously defined acceptance and test specification procedures, which are executed in an operational environment and host computer configuration as possible. The phase terminates with successful "operational demonstration," witnessed by the customer and his quality assurance team. The software packages are then ready for installation, integration, and checkout in the operational environment.

The final step in the software development life cycle is the forerunner of the operations and maintenance phase, strictly speaking, Step 8. This phase is concerned with installation, integration, and checkout in the full operational configuration. The software contractor assists in any way required to integrate the software products of the development, or acquisition, phase into an operational software package that meets the original (and updated) performance and design requirements, the product of Step 1 and subsequent updates. The activities of this phase will also include training, rehearsal support, software problem reporting, fault isolation and correction, formal problem closure and documentation update, and finally, first mission operations support. Because this is not part of the software development effort, we cost it differently—usually as a level of effort.

PRICING PITFALLS

Pricing is the complete process by which we arrive at a price that we quote to the customer. Cost estimating is a major part of that process, although it frequently has to be done several times before a final price is arrived at. The general principles involved in pricing large R & D efforts of any kind have been defined by Beveridge [1], and apply to large software developments as well. Beveridge's discussion centers around the preparation of a proposal in response to a government Request for Proposal (RFP), but his common-sense principles are
applicable to any complex pricing exercise where many individuals are involved.

There is a general tendency on the part of designers to gold-plate their individual parts of any system, but in the case of software the tendency is both stronger and more difficult to control than in the case of hardware. A major task of management is to make certain that the design being proposed (and priced) does meet the customer's need, and that each individual component design can be traced to a specific need, while at the same time it does not provide more (more speed, more accuracy, less need for core, etc.) than the customer needs or wants. Anything being offered a customer beyond what he has asked for should be clearly identified (and priced) as an option. That is what is meant by the "fixed-price attitude": what is the absolute minimum I can do to satisfy the needs stated in the RFP?

SOFTWARE COST ESTIMATING METHODS

The software industry is young, growing, and marked by rapid change in technology and application. It is not surprising, then, that the ability to estimate costs is still relatively undeveloped. Even though many organizations are trying to devise more scientific and objective means for estimating costs of large software systems, the present state of the art is largely judgmental. We will, however, discuss certain important exceptions shortly. A review of other firms and agencies confirm that we are facing a problem common to all [2]-[5].

Estimating the cost of producing computer software relies heavily on the judgment of experienced performers. The software analyst, or estimator, normally breaks the total job into elements that are estimated separately and then summarized into an estimate for the total job. The estimating analysis and synthesis may appear as a mental process or may involve an explicit algorithm [6].

In either case, an empirical data base is used as an objective reference, and the estimator uses his judgment to account for differences. Pieces of information used in the comparisons and adjustments for differences include: a) analysis of initial requirements; b) allocation of requirements to software modules; c) estimates of number of object instructions per module; d) complexity and technological risk; e) user environment and characteristics of the customer; f) computer of choice and interchangeability among other user sites; g) higher order language of choice or criteria for use of assembly language; h) type of software to be developed; i) whether software is to be delivered to an operational user; j) technical experience on that type of job; k) capabilities of the member of the technical staff who probably will do the work; l) type of fee and its incentive structure; m) length of development time; n) single-model multiple-release versus multiple-model single-release development concept; o) performance record of other large-scale systems (AWAC, SAGE, etc.) in the number of instructions and development man-

months; and p) management factors to do with productivity rates, error rates, work environment, availability of computer time, and many other variables.

Most estimators use a logical sequence in establishing their estimates. The logic uses exchange coefficients between some measurable parameter and its cost and adjustment factors derived from experience. One of the pitfalls is that estimating ratios should be directly traceable to recorded cost facts (not hearsay), and even the factual data can contain varying and unstated allowances for risk (we may have assumed a 40-h work week, our records show we charged that rate, but our records do not show that most personnel worked 45-50 h/week or more). Our long-term objective at TRW has been to create a systematic software development estimation system that can significantly reduce statistical variances between estimated costs and actual costs, and is not only accurate in that sense but also simple, fast, and convenient. We will briefly look at estimation methods in general, and two in particular that have been developed and reduced to practice at TRW.

General Logic Followed in Estimating

Traditional cost estimating procedures start with fixing the size of each activity, its start date, and duration. When necessary, adjustments are made to account for the caliber of performer personnel to be assigned, risk, complexity, uncertainties in requirements, and so on. Finally, the amount and type of manpower (man-months per month) and computing resources (hours per month) are converted to dollar costs by applying bid rates. Other direct charges (documentation costs, travel, etc.) are added, and summaries are made through the pricing system. Traditional methods can be classified as one or more of the techniques described below.

1) Top-Down Estimating: The estimator relies on the total cost or the cost of large portions of previous projects that have been completed to estimate the cost of all or large portions of the project to be estimated. History coupled with informed opinion (or intuition) is used to allocate costs between packages. Among its many pitfalls is the substantial risk of overlooking special or difficult technical problems that may be buried in the project tasks, and the lack of details needed for cost justification.

2) Similarities and Differences Estimating: The estimator breaks down the jobs to be accomplished to a level of detail where the similarities to and differences from previous projects are most evident. Work units that cannot be compared are estimated separately by some other method.

3) Ratio Estimating: The estimator relies on sensitivity coefficients or exchange ratios that are invariant (within limits) to the details of design. The software analyst estimates the size of a module by its number of object instructions, classifies it by type, and evaluates its relative complexity. An appropriate cost matrix is constructed
from a cost data base in terms of cost per instruction, for that type of software, at that relative complexity level. Other ratios, empirically derived, can be used in the total estimation process, for instance, computer usage rate based on central processing unit (CPU) time per instruction, peripheral usage to CPU usage, engineers per secretary, and so forth. The method is simple, fast, convenient, and useful in the proposal environment and beyond. It suffers, as do all methods, from the need for a valid cost data base for many estimating situations (business versus scientific, real-time versus nonreal-time, operational versus nonoperational).

4) Standards Estimating: The estimator relies on standards of performance that have been systematically developed. These standards then become stable reference points from which new tasks can be calibrated. Many mature industries, such as manufacturing and construction, use this method routinely. The method is accurate only when the same operations have been performed repeatedly and good records are available. The pitfall is that custom software development is not “performed repeatedly.”

5) Bottom-Up Estimating: This is the technique most commonly used in estimating government research and development contracts. The total job is broken down into relatively small work packages and work units. The work breakdown is continued until it is reasonably clear what steps and talents are involved in doing each task. Each task is then estimated and the costs are pyramided to form the total project cost. An advantage of this technique is that the job of estimating can be distributed to the people who will do the work. A difficulty is the lack of immediate perspective of the most important parameter of all: the total cost of the project. In doing detailed estimates, the estimator is not sensitive to the reasonableness of the total cost of the software package. Therefore, top-down estimation is used as a check on the bottom-up method.

The estimation process in nearly always a combination of two or more of the basic classifications given above. We will briefly discuss two methods that have been used in estimating the cost of developing large-scale software at TRW: a method we used in June 1969 which we will call “smoothing and extrapolation,” and the current method we used beginning in October 1970, which we will call “a software cost estimation algorithm.”

A Systems Approach to Software Cost Estimation

To be as free as possible to deal with the cost estimating method, we will use hypothetical or normalized numbers where numbers are necessary. Assume the software development and test life cycle consists of the seven steps previously defined, except that the starting point at contract go-ahead is immediately on completion of Step 2. We need to understand this assumption clearly, since it establishes the remaining-milestone events which are included in the cost which was negotiated with the customer. In other words, the proposal was at a level of detail such that it was, by itself, the implementation concept and test plan, Step 2. The RFP package, together with the bidder’s briefing, was very detailed, technically complete, and was, by itself, the performance and design requirements, Step 1.

Our reference project for the most recent past successful software project was identified, and its case history was laid out in terms of the actual events, costs, start dates, and durations. The normalized contract cost as a function of the normalized period of performance is shown in Fig. 2. We will call this case history A. The cost distribution by development activity, that is, the actions that characterize the intervals between discrete milestone events, is shown in Fig. 3. Using case history A we can identify similarities and differences with respect to our cost estimating for our new project. The basic cost method, smoothing and extrapolation, is a combination of earlier methods plus some new ideas for the new project. We will call the new project case history B.

A major development cycle difference was that case A has a baseline design specification deliverable under contract, which accounted for about 12 percent of the total cost as shown by “analysis,” and also had the implementation concept and test plan deliverable under contract, which accounted for another 18 percent as shown by “design” (see Fig. 3). Even though the software development cycle was different, we had exact data on the manpower assigned to the balance of the milestone events, which are similar. We now are in a position to extrapolate from case A to case B. Furthermore, since case A had been produced on time and under cost, we were in a strong position for proving demonstrated cost performance on a previous project.

A major functional difference was that case B represented a unified communication, command, and control software system, whereas case A did not have the command function. Immediately, greater attention was given to the analysis and design of that new technology area and its interfaces. Even here, important similarities were identified in the history file because in case A the “similar” command function was the responsibility of another associate contractor. We concentrated on improving the information transfer by having full control of the interface, and by approaching the solution from an entirely new direction. We decided the technical risk was somewhat higher for case B because our cost data base was incomplete, and because the command algorithm design, coding, and testing were innovative. We developed the overall system block diagram, and allocated statement of work tasks to software functions. We were in a position for a first rough cut target cost.

The whole idea in this case was a) to price the total software system from the top down using the experience of the system concept group, b) to price each work
package from the bottom up using the experience of each of the five work package managers, and c) to conduct a mock negotiation between the system concept group and each work package group taken one at a time. Differences were derived, costs were traced back to source authority requirements in the customer's statement of work, and adjustments made between the cost, probable risk, and scope of work.

In both methods the basic unit of estimation was the "man-month" for a given task or element. The distribution of the work (man-months per month) was driven by two critical variables: initial operational capability (IOC) and full operational capability (FOC) software configurations were required by the RFP, and a new incentive fee structure was required by the RFP. Each will be briefly addressed by assessing its impact on the cost method.

The distribution, or manpower spread over the proposed period of performance, was initially determined by the system concept group using one new idea. That idea was first to assume that the total development manpower resource was 1000 man-months, and then to determine the shape of the manpower distribution over time that maximized the fee incentive with least risk, such that the area under the curve remained constant (namely, 1000 man-months). This curve would serve to generate the master schedule for IOC and FOC events. The amplitude of the curve would be the primary focus in the separate negotiations with the five work package managers.

The amplitude represented the number of man-months per month for any given time slice. The sum of the manpower estimates of each of the five work packages at any time point determined that month's manpower requirements, and the sum of those over all time determined the total IOC/FOC manpower requirements, hence total cost. Of the five initial estimates, four were judged too high in the internal first cut negotiations, and one was judged too low.

The shape of the first and second estimate by the system concept group is given in Fig. 4. It is a poor distribution with steep manning rates, high peaks, and sharp declines. Considerable smoothing was required to achieve a unimodal increasing (and decreasing) shape, at comfortable manning rates and reasonable levels and still meet the original goal we set with respect to incentive fee. The shape was drastically altered by introducing the concept of "staggered milestone events." That is, instead

---

**Fig. 2.** Typical software development cost experience (case history A).

**Fig. 3.** Cost distribution by activity during full period of performance (case history A). All activities include documentation and travel costs.
of one event 4) and one event 5) (steps as previously defined), we planned for a total of six each. We could do this because of modular construction of the software to the end that each module was ready when needed, and system level testing by the independent test and validation group could commence soon enough to meet all incentives, at least by plan. The smoothed shape of the third estimate, and the one that influenced the time

---

### Fig. 4

(a) Manpower estimate with single-release milestone approach, showing key deliverables and labor allocation of 1000 man-months. (b) Manpower estimate with single-release milestone approach, showing uneven labor spread of 1000 man-months.
placement of the incentive events, is shown in Fig. 5. Both sets of curves show only IOC activities for simplicity; as IOC man loading phased down, FOC phased up, in general.

A simplified description of the fee structure that influenced the shape of Fig. 5 is that cost, schedule, and performance were incentivized (from a maximum of 15 percent of final target cost to a minimum of 0) based on certain predetermined rules that would be applied at three discrete milestones: Step 4), the detailed design specification, Step 5), the updated detailed design specification with computer program listing, and Step 7), opera-
demonstration (OD). The plan called for three IOC and three FOC dates.

The effect of a two-step procurement, such as an IOC followed by a later FOC, is as follows [7]. The IOC milestone events and OD constitute the only way that any fee can be earned, in discrete steps as shown in Fig. 6. The earned fee percentages apply to the target cost for the total IOC/FOC task. The incentive fee structure provides for fee penalties for failure to meet the contract dates for satisfactory demonstration of milestone steps 4), 5), and the OD. These penalties are assessed as shown in Fig. 7, on a linearly graduated basis in units of whole days late, where “days late” mean calendar days from the date specified in the contract. The penalties apply separately to both IOC and FOC. That is, the maximum penalty for lateness in demonstrating the OD of the IOC is 15 percent; it is also 15 percent for failure to demonstrate the OD of the FOC. This insures priority to the IOC, but also insures responsible attention to the FOC.

To be effective, the basic incentive structure must be simple, even though it is necessary in the contract to address the major contingencies and allowable options. If the basic incentive structure is not simple it will not readily be grasped by the many people at all levels of the contractor’s plant whose work affects the chance of success. If they do not understand it, they will not do anything differently because of the incentive structure. If they do not, the incentive contract will have failed to achieve its fundamental purpose: the people who work on all aspects of the entire undertaking must be conscious of the incentive and must do their work with more care and quality because of it.

The entire incentive approach presumes that the contractor will take specific internal implementing action. It should include some tangible internal management actions that place an additional incentive on the work quality. A clear explanation of the essential features of the incentive structure should be given to all who work in any manner on the software, keyed to the contribution each one can make to affect the fee that can be realized by his company.
The normalized contract cost as a function of the normalized period of performance is shown in Fig. 8 for case history B. The normalized cost distribution by development activity over the software life cycle, for both IOC and FOC, is shown in Fig. 9. In estimating the cost of large software systems, we treat “operations and maintenance” (O&M) as a level of effort commencing immediately after selloff of the system in its operational environment. In the multimodel software development, O&M of the first model overlaps purely development activities of the second model. To avoid any misinterpretation of the development costs, the O&M amounted to 15 percent of the total contract value. Comparison of case history A and B in size and cost (man-months), excluding O&M is given below:

<table>
<thead>
<tr>
<th></th>
<th>Number of Object Instructions</th>
<th>Man-Months</th>
<th>Rate (I/M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case History A</td>
<td>102 400</td>
<td>570</td>
<td>180</td>
</tr>
<tr>
<td>Case History B IOC</td>
<td>160 000a</td>
<td>945</td>
<td>170</td>
</tr>
<tr>
<td>Case History B FOC</td>
<td>215 000b</td>
<td>630b</td>
<td>340</td>
</tr>
</tbody>
</table>

* IOC activities included design and production of a prototype software package of an additional 20,000-object instructions. A slightly modified version was made during FOC.  
* Learning curve phenomenon has been verified by empirical data and controlled tests in other industries [8], but not systematically for the software industry.

A Software Cost Estimation Algorithm

TRW has developed a cost estimation algorithm based on the assumption that costs vary proportionally with the
number of instructions. For each identified routine, the
procedure combines an estimate of the number of object
instructions, category, relative degree of difficulty, and
historic data in dollars per instruction from the cost data
base to give a trial estimate of the total cost. The design
group estimates the first three software parameters. The
system concept group provides the appropriate cost data
base used by all designers, as well as the allocation of
resources to each phase of the software development
cycle, the schedule for each milestone event, and the labor
mix. The procedure then spreads the total cost over the
period of performance according to these input param-
eters, gives the man-months per month by labor category,
the cost associated with each development activity, and
the computer usage by month for checkout and test
activities. The output may be considered a “trial estimate”
in the sense that the proposal team must be satisfied with
the resulting estimate when tested against any conven-
tional method of cost estimating.

The first step in the method is to categorize the software
routines that are being considered in the preliminary
design. The software categories have been selected based
on experience, and are those functionally different kinds
of software entities for which a significant cost per in-
struction (CPI) is expected. Categories that have stood
the test of usage in several proposal and preliminary
design activities are: a) control routine, which controls
execution flow and is non-time-critical, C; b) input/output
routine, which transfers data into or out of the computer,
I; c) pre- or postalgorithm processor, which manipulates
data for subsequent processing or output, P; d) algorithm,
which performs logical or mathematical operations,
A; e) data management routine, which manages data
transfer within the computer, D; and f) time-critical
processor, which is a highly optimized machine dependent
code, T.

The next step is size and complexity estimates by
routine, or subprogram, by the designer. To balance cost
and risk, the designer may decide to use software elements
that are available to him from a software library and need
only some degree of modification or adaptation. At the
other extreme, a new technique may be required and be
estimated as a high technological risk. To account for
the degree of difficulty of a given kind of routine, the
designer estimates a risk or complexity factor. This is
the most crucial step in the estimating process, for it
establishes the cost of the routine with all direct and in-
direct charges amortized against it. The other steps
basically determine how the total cost will be spread over
the development cycle.

Two consultants have different views of how software
parameters should be estimated, in general. Brandon says
a single individual should establish a complexity rating
scale (A,B,C,D,E,F) and make a “standard estimate”
for each job based on: a) complexity rating of each job;
b) machine used; c) language used; and d) estimated
number of instructions. Brandon uses equations fitted to
historical data to get standards, and then measures
performance against the standard [9].

Lecht believes the estimator should interview the
member of the technical staff who will do the job, and
negotiate personal agreement on effort. Historical cost
data reinforce the estimator's judgment where similar
jobs can be found. The estimate is based on: a) similarity
with previous modules; b) person doing the job; c)
machine used; d) language used; and e) estimated
number of instructions. Lecht does not believe meaningful
performance standards can be set for software [10].

The simplest technique for narrowing down the many
subjective choices early in the estimation process is to ask
is the routine new or old, and is it easy, medium, or hard?
A complexity rating coefficient can be applied continu-
ously from 1 to 20 as a multiplier, if preferred. The
important consideration is allowing sufficient degree of
freedom to accommodate a learning experience and in-
dividual differences in performance in the effort to obtain
a realistic cost. For our present purposes of early pre-
liminary design, we will allow the designer four choices
to account for six levels of difficulty for each routine. They
are:

<table>
<thead>
<tr>
<th>Easy</th>
<th>Medium</th>
<th>Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old</td>
<td>OE</td>
<td>OM</td>
</tr>
<tr>
<td>New</td>
<td>NE</td>
<td>NM</td>
</tr>
</tbody>
</table>

The only parameter that changes as a function of degree
of difficulty (OE through NH) is cost per instruction.
Override control is available to the designer. Sample data
sets will be given in the section on Software Cost Data
Base.

The next step is to identify the various development and
test phases for producing the software from the conceptual
stage to delivery of the operational software to the ultimate
user. For our present purposes, the seven steps given
in the section entitled The Software Development Cycle
will be adopted, but they could be any other steps tailored
to the needs of the particular application. For each phase
an estimate is required for the fraction of the total amount
to be allocated to it. In the case of manpower allocation
for distribution of 1000 man-months over the development
cycle, it was functionally allocated as 424 man-months
to design and analysis, 210 man-months to code and debug,
and 366 man-months to check out and test (see Fig. 5).
This distribution of 42–21–37 percent at this level of
detail has been borne out in practice by several other
researchers, as will be demonstrated shortly.

The fourth step is to define the activities in each de-
velopment phase by means of an activity array and associ-
ated cost matrix. The activities as a function of develop-
ment phase is a 25 × 7 matrix; that is, 25 activities are
identified for each of 7 phases. In turn, subsets of the
activities may be summed to "super activity" levels, and the makeup may vary from phase to phase. A typical activity array and cost matrix will be presented in the section entitled Resource Allocation.

The final step in setting up the initial conditions for the cost estimation algorithm is to provide schedule data based on the customer's statement of work, or other management considerations. Schedule data are input as months from go-ahead for each of the milestone periods. Burden rates are input for projected overhead rates, general and administrative, and so forth. Labor mix is defined and unburdened bid rates for the labor grades desired for the phase are defined. Selective cost cuts are under management control by the override capability. Other direct charges are input, which for software is typically travel as a percentage of direct labor costs, such as 3 percent and documentation at 10 percent.

Computer usage data for a machine in the CDC 6500 class with time-shared central processor unit based on sampled data from a programming department by month have been [11]

<table>
<thead>
<tr>
<th>Number of MTS</th>
<th>Total Number of Processor Units Used</th>
<th>Number of Processor Hours per Man-Month</th>
<th>Peripheral Hours per Man-Month</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>55.6</td>
<td>2.20</td>
<td>7.6</td>
</tr>
<tr>
<td>35</td>
<td>31.0</td>
<td>1.27</td>
<td>4.4</td>
</tr>
<tr>
<td>33</td>
<td>30.0</td>
<td>1.30</td>
<td>4.5</td>
</tr>
<tr>
<td>33</td>
<td>46.5</td>
<td>2.02</td>
<td>7.0</td>
</tr>
</tbody>
</table>

Allowing for slightly less power in a 370/155 computer, for instance, a figure of 3 h/month/programmer man-month would be reasonable. At an average productivity of 1 object instruction/h, this is equivalent to 156 instructions/man-month, or 1.2 min/instruction. The data in the third column are based on 1.42 processor units corresponding to 1 h of computing time. A computer hours matrix in terms of usage rate per instruction by phase is given in the section entitled Resource Allocation based on the above data.

The summary output from the cost estimation algorithm, SPREAD, includes: a) cost per routine based on either historic burden rates or proposed burden rates; b) average cost per instruction, number of instructions, and category; c) total number of development computer hours per routine; d) simple graphic display of schedule and events; e) cost breakdown by development phase per routine in total dollars and percent; f) cost breakdown by activity, by routine, and summed over all routines in the software, such as management, review, documentation, specifications, design, coding, and testing; and finally, g) manloading and cost summary by segment showing for each month the labor breakdown for senior staff, staff, technical, clerical; also, computer hours by month, other direct charges, cost by month, and cumulative cost.

The outputs from the cost estimation algorithm are considered a "trial set" for the cost estimation group. The trial set is used in combination with all other sources of data to test that cost position against the project objectives. The approved trial set, which contains the best judgmental and quantitative measures of cost per software element and per activity, becomes the input to the official pricing computer run. Necessary translation of the data set to exactly match the customer's work breakdown structure (WBS) or other appropriate cost elements is made for the final pricing run. The official cost figures are produced by cost guidelines, approved rates, and procedures that have been established by the in-house pricing group and approved by the government auditor. The results of the software cost estimation algorithm are retained as cost backup data and for possible use in later cost justification. The official cost figures from the pricing computer run show manloading and costs collected and aggregated against the customer's WBS (see Fig. 10).

This cost estimation approach has the following advantages. a) The amount of data required to obtain a cost estimate is minimal. b) The software designer immediately sees the cost consequences of his preliminary design. c) To the extent that the routine is directly traceable to a requirement, the requirement is directly traceable to a total cost. d) Comparisons of alternate schedules on resource allocation can be made rapidly. e) A cost justification is produced that can stand the test of government audit.

SOFTWARE COST DATA BASE

Sensitivity coefficients or exchange ratios used by the cost estimation algorithm reside in the data base. The objective in cost estimating is to realize greater accuracy and precision while anticipating risk and its impact on profit. The greater the risk, for a given confidence level, the greater the allowance on the part of the estimator for this uncertainty. The more uncertain the estimate, the less competitive is the cost proposal. Further, an estimate of cost backed by relevant cost experience facts gives confidence to negotiator, management, and the customer. Cost justification is demanded by the customer, and is usually provided in the form of lower level back-up data. Valid data based on proven experience are required for any cost estimation process. Producing good operational software is dependent on the many activities of good people properly directed and motivated. The process is difficult to measure and apply widely beyond the technology center for which the performance measures were derived. There is no universal model. Applying "average productivity rates" without knowledge of the development steps that were used in deriving the rates is wrong. Averages based on large samples are useful in comparison of a local variable against a global variable, if the estimator is reasonably certain he is comparing truly equiv-
alent measures. Metzelaar has studied the problem of industry-wide productivity rates in terms of program size and complexity, as shown in Fig. 11 [12].

The activities that productively occupy the analyst’s time, along with the plant overhead and other direct charges, largely determine the price to the customer. It is in the company’s best interest to agree on measurable activities that should be available in the software cost data base for cost justification and estimation of new work. Brief definitions of activities that have been identified in this regard are given below [2].

1) Learning and Orientation (L): Those actions necessary to gain understanding of the hardware and software to be applied in the project, and to learn the operating requirements, specifications, and constraints to be satisfied by the software package being produced.

2) Analysis and Design (A): All actions necessary to generate technical solutions, which are expected to satisfy requirements and specifications within the imposed constraints. It includes determining and evaluating technical approaches, determining and evaluating the effects of specific requirements and constraints on potential technical solutions, selecting the best solution, and describing that solution so it can be coded. This activity includes the rough documentation that is normally produced in generating a technical solution, but not the efforts of finalizing the documentation package.

3) Coding (C): The translation of the detailed steps of the technical solution into machine-readable language; the ordered list, in computer code or pseudocode, of the successive computer operations for solving a specific problem. It does not include actual input to the computer.
(except for time-share inputs), compiling, nor testing the adequacy or validity of the instructions.

4) Test and Checkout (T): All actions necessary to assure that the instructions coded by the programmer cause the machine to do what the programmer intended it to do. It covers preparing the test data, compiling the program, running the test data, reviewing the compiler and/or machine output, identifying errors, correcting the errors, and making changes to the program that improve its reliability and efficiency. It does not include verifying that the software product satisfies the requirements and specifications stated by the customer or sponsor.

5) Verify/Quality (V): Those actions necessary to assure that the software product satisfies the requirements and specifications provided by the customer or sponsor. It includes preparation of qualifying test data, verification runs, evaluation of computer run results, and adjustments to the program to correct any deficiencies.

6) System Integration and Test (I): All actions necessary to assure that the subsystem or module as programmed will interface with other subsystems or modules to provide a satisfactory system. It includes preparing data for evaluating the joint functioning of two or more subsystems or modules, making the evaluation runs, evaluating the joint operation, and making necessary modifications to the software in accordance with prevailing configuration control procedures.

7) Documentation (D): Those actions that require technical editing, technical typing, art work, and reproduction of deliverable documents to the customer or sponsor.

An example of the cost per instruction for the significant categories of software previously defined versus degree of difficulty is given in Fig. 12. The cost figures, in principle, include all seven development steps described in the section on software development and test management. Activities that are defined immediately preceding are inherently included in the cost figures as are all other direct charges up to and including general and administrative costs.

In late 1971, TRW compiled in-house data on a wide variety of software development characteristics, which in turn were independently analyzed by Lulejian and Associates under contract to the U.S. Air Force [13, 14]. The variables of particular interest for our present purposes were analyzed by Lulejian and Associates for a package of 88 routines developed by TRW. A brief interpretation of four of the analyses that bear on the cost of developing software is presented next.

Consider Fig. 13. This is a plot of cost versus number of instructions for the 88 routines. Two conclusions can be drawn, either by direct observation or by regression analysis. a) There is a general trend in the data—larger routines cost more. b) A linear fit to the data is not very good. If one attempts to predict costs from routine size alone, one can expect errors of about 50 percent in the predicted cost. The prediction is better than nothing, but not much better. Fig. 14 shows the same data plotted another way—cost per 1000 instructions versus number of instructions. If the cost of the routine can be estimated by a fixed cost per 1000 instructions, one would expect this data to show constant cost. It appears to be constant, with a large error, especially for small routines. Again, one concludes that cost can be predicted from number of instructions provided that one is willing to accept fairly large errors.

Cost depends on a number of other factors, and one might hope to get better cost estimates by including difficulty, programmer experience, etc. A considerable number of fits were tried. The answers, unfortunately, were negative. Including other variables did not result in an equation that gives any significantly better fit to the data. Fig. 15 shows an example of one of the variables—
experience of programmer. A glance at the curve suggests that a knowledge of this variable does not help one predict cost. The regression analysis results support this suggestion. Table I shows the results of a multivariate linear regression analysis. Adding the remaining variables gives a better fit, but not much better.

The conclusion is that there are no simple universal rules for costing software accurately. It is necessary to understand the nature of the individual program and the individual routines within the program.

RESOURCE ALLOCATION

Various researchers have separately discovered by empirical methods a rule of thumb, namely, that analysis and design account for 40 percent, coding and debugging account for 20 percent, and checkout and test account for 40 percent of the total resource (cost) spent for software development, the 40–20–40 rule. If the cost of “modeling” of physical systems is to be borne by the software development group, as contrasted to modeling analysis from raw data by technology centers within the company (such as thermal, attitude control and pointing, propulsion, electrical power, and so forth) an extra 30 percent typically would be required in addition to the software development by itself. In other words, the deriving of equations of motion or physical behavior of other physical systems from raw data is not included in the software development resource allocation. Similarly, the cost of computing time is not covered in the resource allocation. Experience has consistently shown that computer costs add an additional 20 to 25 percent of the total cost of the project; that is, a $5 million/year software development contract would cost the government an additional $1 million for GFE computing time to the developer. Boehm discusses his views on resource allocation in [15]; his findings are presented in Table II. The findings of an in-house survey by Metzelaar are presented in Table III. Independently derived resource allocation for the 1000 man-month costing example was given in Figs. 4 and 5.

The basic resource allocation data that are required for application of the cost estimation algorithm are shown in explicit form for purposes of illustration only in this paragraph. Table IV shows a simplified version of how “complexity” might be handled in a preliminary design. For example, if we are designing one routine in a large system that we estimate to have 1000 executable, or object, instructions (not Fortran IV source instructions), and that has as its primary function setting up the input

---

**Table I: Multivariate Regression Summary**

<table>
<thead>
<tr>
<th>EQUATIONS</th>
<th>GOODNESS OF FIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>COST = 0.25 + 0.00028 (NUMBER OF INSTRUCTIONS)</td>
<td>$r^2 = 0.43$</td>
</tr>
<tr>
<td>COST = 0.021 + 0.00018 (NUMBER OF INSTRUCTIONS)</td>
<td>$r^2 = 0.49$</td>
</tr>
<tr>
<td>+ 0.19 (DIFFICULTY)</td>
<td></td>
</tr>
<tr>
<td>COST = 0.005 + 0.00029 (NUMBER OF INSTRUCTIONS)</td>
<td>$r^2 = 0.60$</td>
</tr>
<tr>
<td>- 0.00111 (NUMBER OF LOGICAL INSTRUCTIONS)</td>
<td></td>
</tr>
<tr>
<td>- 0.00027 (NUMBER OF INPUT/OUTPUT INSTRUCTIONS)</td>
<td></td>
</tr>
<tr>
<td>+ 0.0088 (NUMBER OF INTERFACES)</td>
<td></td>
</tr>
<tr>
<td>+ 0.136 (DIFFICULTY)</td>
<td></td>
</tr>
<tr>
<td>+ 0.0034 (PERCENTILE RATING)</td>
<td></td>
</tr>
<tr>
<td>+ 0.0080 (YEARS OF EXPERIENCE)</td>
<td></td>
</tr>
<tr>
<td>+ 0.0259 (YEARS OF SCF EXPERIENCE)</td>
<td></td>
</tr>
<tr>
<td>+ 0.103 (SUPERVISORS RATING)</td>
<td></td>
</tr>
</tbody>
</table>

---

**Table II: Computer Program Development Breakdown**

<table>
<thead>
<tr>
<th></th>
<th>ANALYSIS AND DESIGN</th>
<th>CODING AND AUDITING</th>
<th>CHECKOUT AND TEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAGE</td>
<td>39%</td>
<td>14%</td>
<td>47%</td>
</tr>
<tr>
<td>NTDS</td>
<td>30%</td>
<td>20%</td>
<td>50%</td>
</tr>
<tr>
<td>GEMINI</td>
<td>36%</td>
<td>17%</td>
<td>47%</td>
</tr>
<tr>
<td>SATURN V</td>
<td>32%</td>
<td>24%</td>
<td>44%</td>
</tr>
</tbody>
</table>

---

**Fig. 14.** Routine unit cost versus number of instructions.

**Fig. 15.** Routine unit cost versus experience rating factor of programmer.
TABLE III

<table>
<thead>
<tr>
<th>ACTIVITY</th>
<th>INFORMATICS</th>
<th>RAYTHEON</th>
<th>TRW</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANALYSIS</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
</tr>
<tr>
<td>DESIGN</td>
<td>16</td>
<td>20</td>
<td>20</td>
<td>18.7</td>
</tr>
<tr>
<td>CODE</td>
<td>16</td>
<td>25</td>
<td>24</td>
<td>21.7</td>
</tr>
<tr>
<td>TEST</td>
<td>32</td>
<td>25</td>
<td>28</td>
<td>28.3</td>
</tr>
<tr>
<td>DOCUMENT</td>
<td>16</td>
<td>10</td>
<td>8</td>
<td>11.3</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

*M Substantial variation from project to project.

TABLE IV

<table>
<thead>
<tr>
<th>DEGREE OF DIFFICULTY</th>
<th>CATEGORY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td>OE</td>
<td>$21.00</td>
</tr>
<tr>
<td>OM</td>
<td>27.00</td>
</tr>
<tr>
<td>OH</td>
<td>30.00</td>
</tr>
<tr>
<td>NE</td>
<td>33.00</td>
</tr>
<tr>
<td>NM</td>
<td>40.00</td>
</tr>
<tr>
<td>NH</td>
<td>49.00</td>
</tr>
</tbody>
</table>

Fig. 16. Typical allocation of resources in custom software development and test.

to the main algorithm, we first categorize it as a “preprocessor,” or category P. It is new and seems to be of medium difficulty (compared to others we have seen), therefore it is further categorized as new–medium, or NM. We enter on the worksheet for this routine its name, say PROG, its category P/NM, and a number of executable instructions equal to 1000. The consequence of this action is that PROG, at $34.00/instruction, will cost the customer (i.e., directly timeable to PROG) $34,000. This covers the cost of all activities, including analysis, design, documenting, coding, checkout, and test. These activities are spread into the development phases according to the typical allocation of resources shown in Fig. 16. Summing the first four phases prior to code and debug shows we allocate 46 percent of our total dollar there, coding takes 20 percent, and the two major phases after coding take the remaining 34 percent. This distribution has been tested against at least two other computer program development cycles, provides a good fit to present design techniques, and will be adopted as the nominal.

The resource distribution in the seven phases [steps 1–7] previously defined correspond to the seven phases A–G] will vary as a function of “category,” and may be interpreted as a perturbation about the nominal. In all cases H, the O&M phase, will be treated differently since it is not characterized by the same objectives or activities as the development phases. The percent variation of total resource allocation as a function of category is given in Table V.

After classifying the software package (at the routine
level) by category and degree of difficulty, and defining the development phases for producing the software, we define the "activities" to be performed during each development phase. The software statement of work is the source authority for defining activities against which costs will be collected at an appropriate level in the work breakdown structure. The 25 activities (or tasks) that comprise the 7 development phases, which in turn lead to preparation of the deliverables for our hypothetical project, are shown in Table VI.

The assignment of resources to each of the 25 activities for each of the 7 development phases is shown in Table VII. Historical data from the cost data base combined with engineering judgment gained from work on previous similar projects are used in making the assignment.

The remaining step is to assign a properly time-phased computer resource to be used for the development and test of the software. The rationale for such a computer hours matrix was given earlier. The numerical results are given in Table VIII, together with an explanatory note on interpreting the data array.

An example in the application of the entire process would be useful. Within the space of this paper it is not practical, however, preliminary design of a large command and control software system was carried out in late 1971 that provides two useful insights for our present purposes. The first is that the command and control portion of the software system consisted of two major subsystems, the first containing 69 subprograms which were considered quasi-real-time, and the second containing 29 subprograms which were considered real-time. In fact, some of the former contained real-time subprograms and the latter some nonreal-time subprograms. Thus, if a casual observer were to characterize the software as solely real-time (or nonreal-time), in hopes of deriving useful cost data base parameters, he would be certain to fail in precise interpretation of the performance data. The resource allocation in terms of burdens man-months and computer hours for the effort is given in Table IX at the module level, where routines make up modules, and modules make up the system. If this system had been carried out in a complete development cycle, a new data set would be entered into the cost data base and variances between actual and predicted resources would be available to aid in the next estimating effort.

A second insight that seems useful is the cost estimating relationships that emerge when the software elements are separated into meaningful groupings of, in this case, real-time, real-time plus its supporting environment (pre- and postprocessors programmed in higher order language, for example), and command and control elements that are not time-critical (see Fig. 17). The data are now highly correlated, and the differences in productivity rates between the groupings would be expected to range nearly 3 to 1. The observed differences should not be attributed to lack of correlation between dependent variables that are seen to be members of different sets.

TOPICS FOR FURTHER WORK

The topics that were originally identified for discussion will be summarized, together with key issues that influence cost estimating of large software systems and that need further work and study.

1) What is the typical code production rate per programmer man-month? A working standard may be inferred from the data given of 1 object instruction/man-hour, which is equivalent to 156 instructions/minute-month, or 1870 instructions/year-month, or 6.4 man-months/1000 object instructions for nontime-critical software.

If a burdened man-month varies between $4300 and $4900 in 1972 dollars for a typical programming department's labor mix, on the average this working standard translates to $27.50-$31.50 as the cost per instruction. Both case history A and B correlate strongly with these standards. They are characterized as large software jobs whose development is firmly structured and controlled. But what about the R&D work, not well structured and controlled, and which therefore does not lend itself to estimation by extrapolation from a historical cost data base?

The underlying question is whether we have identified all the right parameters to capture and quantify in our cost data base. Do we understand the causal relationships, and can they be isolated from their effects or symptoms? For example, is the SAGE-data point where it is in Fig. 11 because it is "extremely complex" or because the requirements were poorly defined?

2) How does code production rate vary with problem complexity? With more sophisticated physical systems being brought under software command and control, a
corresponding increase in technological risk and complexity causes an increase in a responsible cost estimate. Reliable estimates place this increase in cost right now from 3 to 3.5 times the average of our historical cost data base. The leading causes of these predicted increases need serious attention and study, and effective management methods to detect and deal with them. For example, we have already observed in two cases that the cost of producing real-time (or time-critical) software is three times more costly than non-real-time software (see Figs. 12 and 17). Should the development dollar be spent on highly optimized machine-dependent code by the applications programmer or on more efficient compilers and assemblers? Cost tradeoffs are needed between the crucial param-

![Table VI](https://www.ieee.org/.../images/tbl-vi.jpg)

**TABLE VI**

**ACTIVITIES AS A FUNCTION OF SOFTWARE DEVELOPMENT PHASE**

<table>
<thead>
<tr>
<th>DEVELOPMENT PHASE</th>
<th>PHASE A</th>
<th>PHASE B</th>
<th>PHASE C</th>
<th>PHASE D</th>
<th>PHASE E</th>
<th>PHASE F</th>
<th>PHASE G</th>
<th>PHASE H</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVITY</td>
<td>PERFORMANCE REQUIREMENTS</td>
<td>IMPLEMENTATION CONCEPT AND TEST PLAN</td>
<td>INTERFACE AND DATA REQUIREMENTS SPECIFICATION</td>
<td>DETAILED DESIGN SPECIFICATION</td>
<td>CODING AND AUDITING</td>
<td>SYSTEM TESTING</td>
<td>CERTIFICATION AND ACCEPTANCE</td>
<td>OPERATIONS AND MAINTENANCE</td>
</tr>
<tr>
<td>PROGRAM MANAGEMENT</td>
<td>1</td>
<td>PROGRAM MANAGEMENT</td>
<td>PROGRAM MANAGEMENT</td>
<td>PROGRAM MANAGEMENT</td>
<td>PROGRAM MANAGEMENT</td>
<td>PROGRAM MANAGEMENT</td>
<td>PROGRAM CONTROL</td>
<td>PROGRAM CONTROL</td>
</tr>
<tr>
<td>REVIEW DOCUMENTS</td>
<td>4</td>
<td>DOCUMENT AND EDIT</td>
<td>DOCUMENT AND EDIT</td>
<td>DOCUMENT AND EDIT</td>
<td>DOCUMENT AND EDIT</td>
<td>DOCUMENT AND EDIT</td>
<td>DOCUMENT AND EDIT</td>
<td>DOCUMENT AND EDIT</td>
</tr>
<tr>
<td>REQUIREMENTS AND SPECIFICATIONS</td>
<td>6</td>
<td>REQUIREMENTS DEFINITION</td>
<td>POR MATERIAL</td>
<td>EVENT GENERATION INTERFACE</td>
<td>PRODUCT CONFIG DETAILED TECH DESCRIPTION (PART II)</td>
<td>TECHNICAL DESCRIPTION UPDATE (PART II)</td>
<td>PRODUCT CONFIG DETAILED TECH DESCRIPTION UPDATE (PART III)</td>
<td>SOFTWARE PROBLEM REPORTS (SPR)</td>
</tr>
<tr>
<td>DESIGN</td>
<td>10</td>
<td>TRADE STUDIES</td>
<td>TRADE STUDIES</td>
<td>TRADE STUDIES</td>
<td>TRADE STUDIES</td>
<td>TRADE STUDIES</td>
<td>TRADE STUDIES</td>
<td>TRADE STUDIES</td>
</tr>
<tr>
<td>CODING</td>
<td>15</td>
<td>PRODUCT AND CONFIGURATION CONTROL</td>
<td>PRODUCT AND CONFIGURATION CONTROL</td>
<td>PRODUCT AND CONFIGURATION CONTROL</td>
<td>PRODUCT AND CONFIGURATION CONTROL</td>
<td>PRODUCT AND CONFIGURATION CONTROL</td>
<td>PRODUCT AND CONFIGURATION CONTROL</td>
<td>PRODUCT AND CONFIGURATION CONTROL</td>
</tr>
<tr>
<td>TESTING, CONFIGURATION CONTROL AND OOA</td>
<td>18</td>
<td>TEST REQUIREMENTS</td>
<td>TEST PLANS</td>
<td>INTERFACES</td>
<td>TEST PROCEDURES</td>
<td>DEVELOPMENT TESTING PLANNING</td>
<td>SYSTEM TEST PLANNING</td>
<td>ACCEPTANCE AND TEST PLANNING</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>24</td>
<td>OPERATIONAL CONCEPT</td>
<td>OPERATIONAL TIMELINE</td>
<td>OPERATIONAL CONCEPT UPDATE</td>
<td>USER'S MANUAL (PRELIMINARY)</td>
<td>OPERATIONAL TIMELINE UPDATE</td>
<td>USER'S MANUAL UPDATE</td>
<td>INTEGRATION SUPPORT</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>25</td>
<td>TRAINING PLAN</td>
<td>TRAINING PLAN UPDATE</td>
<td>TRAINING</td>
<td>TRAINING</td>
<td>TRAINING</td>
<td>TRAINING</td>
<td>TRAINING</td>
</tr>
</tbody>
</table>
eters that define the problem complexity and the cost benefits. A marginal utility theory is needed for the economies of software development. Parameters that are crucial in defining problem complexity include degree and time phasing of simulation, extent of prototype code, parallel development of different formulations of key algorithms, new computer hardware/software, assembly language coding, multisite operations and interchangeability, growth requirements, critical timing and throughput, hardware/software interfaces, fault-tolerant computing, core occupancy constraints, reliability, and safety [16]-[18].

8) How does this rate vary as a function of computer availability? Accessibility? Configuration? The President’s Blue Ribbon Defense Panel reported in June 1970 in its staff report on automatic data processing, that because of the government’s method of selection and procurement of computer systems, multiple computers in the same geographical area can result in costs that are as much as five times larger than would be necessary if a few large computers were used in a shared operating mode. The underlying problem here is that of data privacy and protection (security) for enabling the use of remote terminals by government agencies for both classified and unclassified work. An independent evaluation by E. C.

TABLE VII
Cost Matrix Data Showing Allocation of Resources as a Function of Activity by Phase (Category P)

<table>
<thead>
<tr>
<th>ACTIVITY</th>
<th>PHASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (8)</td>
<td>B (19)</td>
</tr>
<tr>
<td>C (3)</td>
<td>D (14)</td>
</tr>
<tr>
<td>E (23)</td>
<td>F (21)</td>
</tr>
<tr>
<td>G (12)</td>
<td>H (0)</td>
</tr>
<tr>
<td>1 10 6 8 6 7 5 10 2</td>
<td></td>
</tr>
<tr>
<td>2 8 3 3 3 3 3 5 5</td>
<td></td>
</tr>
<tr>
<td>3 6 4 6 6 3 8 5 8</td>
<td></td>
</tr>
<tr>
<td>4 13 8 5 6 5 5 4 2</td>
<td></td>
</tr>
<tr>
<td>5 5 2 2 3 3 2 2 1 1</td>
<td></td>
</tr>
<tr>
<td>6 22 8 7 12 3 7 5 8</td>
<td></td>
</tr>
<tr>
<td>7 10 8 7 2 2 6 6 8</td>
<td></td>
</tr>
<tr>
<td>8 7 7 7 5 5 5 5 5 5</td>
<td></td>
</tr>
<tr>
<td>9 6 6 6 6 6 6 6 6 6</td>
<td></td>
</tr>
<tr>
<td>10 17 10 10 8 10 10 10 8 10</td>
<td></td>
</tr>
<tr>
<td>11 2 10 9 7 6 6 6 6 6</td>
<td></td>
</tr>
<tr>
<td>12 2 5 13 13 13 13 13 13 13</td>
<td></td>
</tr>
<tr>
<td>13 4 7 10 3 5 5 5 5 5</td>
<td></td>
</tr>
<tr>
<td>14 4 3 3 3 3 3 3 3 3</td>
<td></td>
</tr>
<tr>
<td>16 4 4 5 4 4 10 10 10 10</td>
<td></td>
</tr>
<tr>
<td>17 3 6 5 6 5 8 10 10 10</td>
<td></td>
</tr>
<tr>
<td>18 5 6 8 5 2 5 10 10 10</td>
<td></td>
</tr>
<tr>
<td>19 10 15 14 5 14 14 14 14 14</td>
<td></td>
</tr>
<tr>
<td>20 10 10 10 10 10 10 10 10 10</td>
<td></td>
</tr>
<tr>
<td>21 2 4 5 5 7 9 9 9 9</td>
<td></td>
</tr>
<tr>
<td>22 5 6 8 4 4 4 4 4 4</td>
<td></td>
</tr>
<tr>
<td>23 23 23 23 23 23 23 23 23 23</td>
<td></td>
</tr>
<tr>
<td>24 4 3 2 3 5 3 3 3 3</td>
<td></td>
</tr>
<tr>
<td>25 2 1 2 3 5 3 3 3 3</td>
<td></td>
</tr>
</tbody>
</table>

TABLE VIII
Computer Hours Matrix Showing Computer Usage Allocation as a Function of Phase

<table>
<thead>
<tr>
<th>PHASE</th>
<th>CATEGORY</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>I</td>
<td>P</td>
<td>A</td>
<td>D</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0 0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0 0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0 0.0017</td>
<td>0.0017</td>
<td>0.0017</td>
<td>0.0017</td>
<td>0.0017</td>
<td>0.0017</td>
<td>0.0017</td>
<td>0.0017</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0.0007</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
</tr>
<tr>
<td>E</td>
<td>0.008</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
</tr>
<tr>
<td>F</td>
<td>0.005</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>G</td>
<td>0.005</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>H</td>
<td>0 0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: The column sum of category C, control routine, is 217 X 10⁻⁴ h instruction, or 1.4 min/instruction. The total resource required is then distributed over phases D-G, as shown above, e.g., 0.42 min/instruction for phase E, coding and auditing.
of a dedicated computer installation costs three to five times as much to operate as if it were a time-shared installation with proper remote terminals. If private industry can enjoy these benefits of cost effectiveness by use of remote terminals (three to five times less expense), it would seem that some, if not a large part, of government installations could enjoy the same benefits if the problems of data privacy and protection were solved to everybody’s satisfaction.

A management goal is to do the difficult parts of the software twice—once to produce prototype code early in the development cycle using perhaps 8–10 percent of the resources and an improved version with the technological risk eliminated for the operational version. One pitfall is that the risk is highest in procurements where a new computer and its operating system are being developed concurrently with advanced applications software, so just when computer time is needed the most, it is available the least. More planning to reduce the computer selection and procurement time is needed.

Furthermore, in some operational systems, the computer configuration available to the user is significantly different from the computer configuration available to the computer program associate contractor. Such a working environment requires an integrating contractor to maintain system-level configuration control. This may add 10 percent to the total cost and is viewed as an insurance policy. To eliminate such occurrences, standards for computer configurations are needed, and configuration control is needed by the associate contractor and, equally importantly, by the user.

4) **What are the pieces of information required to make a realistic prediction of software development cost?** We have identified the activities that occupy the analyst’s time in a 25 x 7 matrix (Table VI) and elsewhere. However it is one thing to identify significant activities for allocation of resources, and it is another thing to hold the right set of individuals accountable for the expenditure of those resources for those things over the life cycle of the software development, which spans 12–24 months or longer. The purpose of a cost estimating system is to reduce the variance between estimates of what a software development should cost and what it actually does cost. The cost history data file is a key element in an improved cost estimating system. Further work is needed in capturing and maintaining historical estimates, costs, and quantitative facts to provide more effective data for estimating future costs and sizes. Further work is needed in maintaining cost control over the actual performance period considering the following.

a) Recording and displaying of both individual and aggregate programmer activities, such as previously defined by $L, A, C, T, V, J$, and $D$, by easy-to-use automated project control and prediction tools.

b) Fast computation of project cost- and time-to-complete predictions by current and look-ahead activity statistics, including uncompensated overtime, as is now feasible through automated cost control tools.

c) Provision for storing, retrieving, and tracing of project statement of work deliverables to their cost estimate and cost estimate to actual cost, by activity, over the useful life of the software through cost control tools.

5) **How does code production rate vary as a function of**
programmer quality? Quality requirements on the final product? The point to be made here is that we have seen convincing evidence of the validity of the 40–20–40 rule. For a $5 million project, $2 million will be spent on check-out and test to the extent the rule applies. A management objective is to check out every logic path in the software to deliver an error-free program. This is a high-quality final product insofar as being error-free is a measure of quality. Achieving this objective may add significantly to the development cost but reduce even more significantly the operations and maintenance cost. With improved test tools and flow measurement tools, considerable improvement in efficiency of more branches checked per computer hour becomes realizable. From the analysis given for one software system, the quality of the programmer in years experience and supervisor's rating seemed to make no difference in the cost in terms of the goodness of fit, the adjusted index of determination (r²) given in Table I and Fig. 15. As R. J. Hatter says, some of the signs exhibit counter-intuitive behavior.

Technical performance standards and criteria are needed in the software development process. A wrong standard is better than no standard because at least it gives a sense of direction to the effort. A technical performance measurement system is needed for applying to the software development process, and which is understood and applied at all management and performer levels over the life cycle. Since software development is more process oriented than product oriented, such standards become measures of group productivity if not individual productivity. In some organizations it may be interpreted to be against company policy to measure and report on a productivity index of an individual member of the professional staff, such as code production rate, or error rates (as measured, for instance, by software problem reports). Producing quality custom software is a profit-motivated business, and more businesslike procedures are being applied as the product lines mature [19].

6) How does cost vary with completeness of problem formulation? Using the 40–20–40 rule again, but this time dealing with the “analysis and design” 40 percent (which for our example of Fig. 16 was 46 percent), we have evidence that analysis and design takes the lion's share of the software development dollar. A management goal that has proven effective in the past is to produce superior documentation which is reviewed with a knowledgeable technical staff in the customer's complex. Thorough and continuous involvement of the customer in the development process has been a reality of several large software developments. Nothing takes the place of competence and communication when it comes to understanding the customer's or sponsor's requirements. Translating total system requirements, which may not be well understood even by the customer, to the software system requirements is a crucial first step. In the requirements definition and preliminary design phase, there is typically a lack of data but lots of leverage (influence on the ultimate design outcome), in contrast to the operational phase where there are lots of data but little, if any, leverage. This is a significant area for cost- and performance-effective improvement.

7) What is the role of “design-to-cost” in the development of large-scale software? The DOD joint logistics commanders have recently entered into an agreement concerning the acquisition and ownership of major weapon systems based on the concept of “design-to-cost” [20]. The term “design-to-cost” is defined as a process utilizing unit cost goals as thresholds for managers and as design parameters for engineers. Cost parameters (goals) are to be established that translate into “design to” requirements. In the past, the order of priorities for major weapon system acquisition and ownership has been: a) performance, b) availability, and c) cost. Today, under the design-to-cost doctrine, the order is: a) cost, b) performance, and c) availability. The applicability of the design-to-cost concept to hardware is now beginning to emerge and is being reduced to practice. The relationship of the design-to-cost concept to software is not now understood.

In the 1975–1980 era, this reordering of priorities for major weapon systems, of which software is generally an integral part, is expected to have a sizeable influence on both hardware and software life cycle acquisition and ownership, particularly in the sense that the unit cost requirement will become a driving parameter in the design of large-scale software. System development will be continuously evaluated against the unit cost requirements with the same rigor as now applied to technical requirements. Practical tradeoffs must be made between system capability, cost, and schedule. Traceability of estimates and costing factors, including those for economic escalation, will have to be maintained.

ACKNOWLEDGMENT

The author wishes to thank Dr. R. H. Pennington, Chief Scientist of System Development Corporation, who invited the author to prepare this basic paper and to serve on a panel at the IEEE International Convention in New York, N. Y., March 20–23, 1972. The other panel members were T. E. Clinis of IBM, V. LaBolle of the Los Angeles Department of Water and Power, and Dr. R. E. Merwin of the Safeguard Systems Office, each one having a different approach to the problem of the cost of developing large-scale software [21]. The author also wishes to thank many close TRW colleagues for their help in getting the ideas and material together for this paper, particularly, D. J. Alley, C. A. Bosch, W. V. Buck, R. D. Kennedy, W. L. Hetrick, W. A. Krumreich, W. C. Lynam, P. N. Metzelaar, Dr. D. D. Morrison,
Dr. F. J. Mullin, Dr. E. C. Nelson, F. Putich, E. A. Rollin, and L. L. Wolfson. A special note of recognition is due Dr. W. W. Royce, who was the prime originator of TRW’s SPREAD cost estimation algorithm. The author must, in the final analysis, take full responsibility for this paper, including errors of fact, omission, or expressions of opinion.

The author wishes to thank R. J. Hatter of Lulejian Associates, Inc., for the release of Figs. 13-15 and Table I from the West Coast Study Group report, and Dr. B. W. Boehm of TRW, formerly of The Rand Corporation, for his resource allocation data, Table II [15], [22].

Guidance from the three referees and Dr. R. A. Short of the IEEE TRANSACTIONS ON COMPUTERS is appreciated. Their decision to depart from the traditional IEEE TRANSACTIONS ON COMPUTERS editorial policy and publish a software technical management paper is gratefully acknowledged by the author. Their aim is to provide information useful to the hardware computer systems designer in learning more about the software development process, in pointing to the tasks where the programmer expends his major effort and, it is hoped, in identifying problem areas to attack so future computer systems may ease the programmer’s task.

REFERENCES


Ray W. Wolverton (S’48–A’50–M’52) was born in Hopkins, Mo., on October 29, 1924. He received the B.S.E.E. degree from Stanford University, Stanford, Calif., in 1950, and did graduate work in mathematics, controls, and computer system design at the University of Southern California, Los Angeles, and the University of California, Los Angeles, in 1950–1952 and 1952–1955, respectively; and in management theory at the Industrial College of the Armed Forces, Fort Lesley, U. McNair, Washington, D. C. in 1971–1972.

From 1950 to 1955 he was a member of the technical staff at Hughes Aircraft Company, working in the communication, navigation, and landing activity for the MX-11 79 manned interceptor. He joined the Ramo-Wholdridge Guided Missile Research Division in 1955, where his initial interests were in closed-loop trajectory simulations and orbital operations. Since 1965 he has been with the TRW Systems Group, Redondo Beach, Calif., working in the fields of flight mechanics, real-time command and control operations, computer software development and test management, and project management. His present interests are in large-scale computer program management, cost modeling and forecasting, and reliability (error-free software). Currently, he is a Senior Staff Engineer with the Systems Design and Integration Operations, Systems Engineering and Information Division of the TRW Systems Group. He is a coauthor and editor of the Flight Performance Handbook for Orbital Operations, produced under contract to NASA (New York: Wiley, 1963). He is a member of Sigma Xi, the Scientific Research Society of North America, Phi Kappa Phi, the American National Standards Institute, and various professional societies. He is listed in Who’s Who in the West and is a recipient of both NASA’s Apollo Achievement Award and the astronaut’s Silver Snoopy Award.