Understanding and Controlling Software Costs

BARRY W. BOEHM, SENIOR MEMBER, IEEE, AND PHILIP N. PAPACCIO

Abstract—**Understanding of software costs is important because of the overall magnitude of these costs (in 1985, roughly $70 billion per year in the U.S. and over $140 billion per year worldwide) and the fundamental impact software will have on our future quality of life. Section I of this paper discusses these issues.**

Section II, the main portion of the paper, discusses the two primary ways of understanding software costs. The "black-box" or *influence-function* approach provides useful experimental and observational insights on the relative software productivity and quality leverage of various management, technical, environmental, and personnel options. The "glass-box" or *cost distribution* approach helps identify strategies for integrated software productivity and quality improvement programs, via such structures as the *value chain* and the *software productivity opportunity tree*.

The individual strategies for improving software productivity identified in Section II are:

- **writing less code;**
- **getting the best from people;**
- **avoiding rework;**
- **developing and using integrated project support environments.**

Section II provides overall surveys of early and recent progress along these and other lines identified by the opportunity tree.

Better understanding of software costs leads to better methods of controlling software project costs, and vice versa. Section III discusses these issues. It points out that a good framework of techniques exists for controlling software budgets, schedules, and work completed, but that a great deal of further progress is needed to provide an overall set of planning and control techniques covering software product qualities and end-user system objectives.

Index Terms—**Programming productivity, software costs, software engineering economics, software management, software metrics, software productivity.**

I. The Need to Understand and Control Software Costs

In this section, we will explore three main reasons why it is important to understand and control software costs:

1) Software costs are big and growing. Thus, any percentage cost savings will be big and growing, also.

2) Many useful software products are not getting developed. Helping good software people work more efficiently will provide time for them to attack this backlog of needed software.

3) Understanding and controlling software costs can get us better software, not just more software. As our lives and lifestyles continue to depend more and more on software, this factor becomes the most important of all.

Manuscript received August 15, 1986; revised May 26, 1987.

The authors are with TRW Inc., One Space Park, Redondo Beach, CA 90278.

IEEE Log Number 8823076.

A. Software Cost Trends

A number of studies have indicated that software costs are large and rapidly increasing. For the United States in 1980, using two separate approaches and relatively conservative assumptions, Reference [24] derived a total of 900 000–1 000 000 software personnel, with a resulting annual cost of $40 billion, or roughly 2 percent of the U.S. Gross National Product. Reference [69] derived a comparable figure of 900 000 professional programmers in the U.S., and a total world programmer population of 3 250 000 (another 900 000 in Western Europe, 500 000 in the Far East, and about 950 000 elsewhere).

Reference [69] estimated the rate of growth of programming personnel at roughly 7 percent per year, which would yield a U.S. professional programmer population of roughly 3 000 000 people by the year 2000, and a world programmer population in the year 2000 of roughly 10 000 000 people. Recent estimates of the dollar growth in U.S. software costs have typically indicated around a 12 percent per year increase (indicating a 5 percent annual increase in personnel cost plus the 7 percent increase in number of personnel). This is consistent with the trends in U.S. Defense Department costs, which went from roughly $3.3 billion in 1974 [46] to roughly $10 billion in 1984 [78]. The recent Electronic Industries Association study of U.S. Defense Department mission critical software costs also predicted a 12 percent annual growth rate from $11.4 billion in 1985 to $36 billion in 1995 [42].

Using a 12 percent annual growth rate, the annual U.S. software cost would be roughly $70 billion in 1985 and $125 billion in 1990. Comparable world software costs are difficult to calculate due to differing salary scales, but they would be at least twice this high: over $140 billion in 1985 and over $250 billion in 1990. Clearly, these costs are sufficiently large to merit serious efforts to understand and control them.

B. The Software Backlog

Several studies (e.g., [23], [82]) have indicated that the demand for new software is increasing faster than our ability to develop it. For example, the U.S. Air Force Data Systems Design Office has identified a four-year backlog of important business data processing software functions which cannot be implemented because of a limited supply of personnel and funding, much of which must currently be devoted to supporting the evolution of existing software (often misleadingly called "software main-

0098-5589/88/1000-1462$01.00 © 1988 IEEE

tenance''). A number of other government and commercial organizations have identified similar backlogs.

This software backlog exacerbates two serious problems. First, it acts as a brake on our ability to achieve productivity gains in other sectors of the economy. It has been estimated that roughly 20 percent of the productivity gains in the U.S. are achieved via automation and data processing. The software backlog means that many nonsoftware people's jobs still have a great deal of tedious, repetitive, and unsatisfying content, because the software to eliminate those parts of the job cannot be developed.

Second, and more serious, the software backlog creates a situation which yields a great deal of bad software, with repercussions on our safety and quality of life. Specifically, the backlog creates a personnel market in which *just about anybody can get a job to work off this software backlog, whether they are capable or not.*

Several studies have shown that, as with productivity, differences between people account for the largest source of variation in software quality. For example, the comparative experiment in Reference [30] showed a 10 : 1 difference in error rates between personnel. The numerous instances of risks to the public summarized by Neumann in *ACM Software Engineering Notes* provide graphic examples of how serious a problem we have created by unleashing unqualified software personnel onto projects producing critical applications software. This leads us to two primary conclusions:

- We need to understand and control software costs as a way of reducing software backlog, and thus of reducing the chances that bad programmers will continue to provide us with more and more bad software to live with.
- We need to understand and control software qualities as well as software costs.

C. Understanding and Controlling Software Costs and Qualities

The interactions between software cost and the various software qualities (reliability, ease of use, ease of modification, portability, efficiency, etc.) are quite complex—as are the interactions between the various qualities themselves. Overall, though, there are two primary situations which create significant interactions between software costs and qualities:

a) A project which tries to reduce software development costs at the expense of quality can do so, but only in ways which increase operational and life cycle costs.

b) A project which tries to simultaneously reduce software costs and improve software quality can do so, by intelligent and cost-effective use of modern software techniques.

Going for Low-Cost, Low-Quality Software: One example of situation a) is provided by the Weinberg–Schulman [121] experiment, in which several teams were asked to develop a program to perform the same function, but each team was asked to optimize a different objective. Almost uniformly, each team finished first on the objective they were asked to optimize, and fell behind on the other objectives. In particular, the team asked to minimize effort finished with the smallest effort to complete the program, but also finished last in program clarity, second to last on program size and required storage, and third to last in output clarity.

Another example is provided by the COCOMO data base of 63 development projects and 25 evolution or maintenance projects [23]. This analysis showed that if the effects of other factors such as personnel, use of tools, and modern programming practices were held constant, then the cost to develop reliability-critical software was almost twice the cost of developing minimally reliable software. However, the trend was reversed in the maintenance projects; low-reliability software required considerably more budget to maintain than high-reliability software. Thus, there is a ''value of quality'' which makes it generally undesirable to reduce development cost at the expense of quality.

Achieving Low-Cost, High-Quality Software: Certainly, though, if we want better software quality at a reasonable cost, we are not going to hold constant our use of tools, modern programming practices, and better people. This leads to situation b), in which many organizations have been able to achieve simultaneous improvements in both software quality and productivity. For example, the extensive survey in Reference [50] of about 800 user installations found that the four most strongly experienced effects of using modern programming practices were ''code quality,'' ''early error detection,'' ''programmer productivity,'' and ''maintenance time or cost.'' Thus, attempts to build quality into a software product will also lead to gains in productivity as well.

However, getting the right mix of the various qualities (reliability, efficiency, ease of use, ease of change) can be a very complex job. Several studies have explored these qualities and their interactions, e.g. [19] and [85]. Also, some initial approaches have had some success in providing methods for reconciling and managing to multiple quality objectives, such as Design by Objectives [51] and the GOALS approach [23, ch. 3]. An excellent review of the state of the art in software quality metrics is [49].

II. Understanding Software Costs

We can consider two primary ways of understanding software costs:

A) The ''black-box'' or *influence-function* approach, which performs comparative analyses on the overall results of a number of entire software projects, and which tries to characterize the overall effect on software costs of such factors as team objectives, methodological approach, hardware constraints, turnaround time, or personnel experience and capability.

B) The ''glass-box'' or *cost-distribution* approach, which analyzes one or more software projects to charac-

terize their internal distribution of costs among such sources as labor versus capital costs, code versus documentation costs, development versus maintenance costs, or other distributions of costs by phase or activity.

These two primary perspectives complement each other, and certainly both are needed to achieve a thorough understanding of software costs. The two parts of this section will explore each of these perspectives in greater detail.

A. Software Cost Influence Functions

The study of software cost influence functions similarly branches in two main directions: controlled experimentation and observational analysis. We shall discuss the results of each approach in turn below.

1) Experimental Results: Some of the earliest experimental results on software cost influence functions were the studies in Reference [54] comparing the effects of batch versus time-sharing computer operation on programming productivity. The experiments typically indicated a 20 percent productivity gain due to time shared interactive operation, but a much more remarkable variation in productivity (up to 26:1) due to differences in programming personnel.

Another set of significant insights resulted from the experiments in Reference [121] discussed earlier, showing the striking effect to team objectives on project productivity and product quality.

During the late 1970's, a number of experiments helped to illuminate the programming process, investigating the effects of code structuring, programming language constructs, code formatting, commentary, and mnemonic variable names on programming productivity, program comprehensibility, and error rates. A good summary of these experiments is given in [104].

Some initial experiments have explored the effects on productivity of prototyping and fourth-generation languages. A seven-project experiment comparing a specification-oriented versus a prototyping-oriented approach to the development of small, user-intensive application software products [28] found primarily that (see Fig. 1):

- Both approaches resulted in roughly equivalent "productivity" in delivered source instructions per manhour (DSI/MH).
- The prototyping projects developed products with roughly equivalent performance, but requiring roughly 40 percent fewer DSI and 40 percent fewer manhours than the specifying projects ($\bar{P}$ versus $\bar{S}$ in Fig. 1).
- The specifying projects had less difficulty in debugging and integration due to their development of good interface specifications.

A six-project experiment comparing the use of a third-generation programming language (COBOL) and a fourth-generation language (FOCUS) on a mix of small business-application projects involving both experts and beginners developing both simple and complex applications [59] found primarily that (see Fig. 2):

- On an overall average ($\bar{F}$ versus $\bar{C}$ in Fig. 2), the fourth-generation approach produced equivalent products to the third-generation approach, with about 60 percent fewer DSI and 60 percent fewer manhours (again with roughly equivalent "Productivity" in DSI/MH).
- From project to project, there was a significant variation in the ratio of third generation : fourth generation DSI (0.9:1 to 27:1), manhours (1.5:1 to 8:1) and DSI/MH (0.5:1 to 5:1).

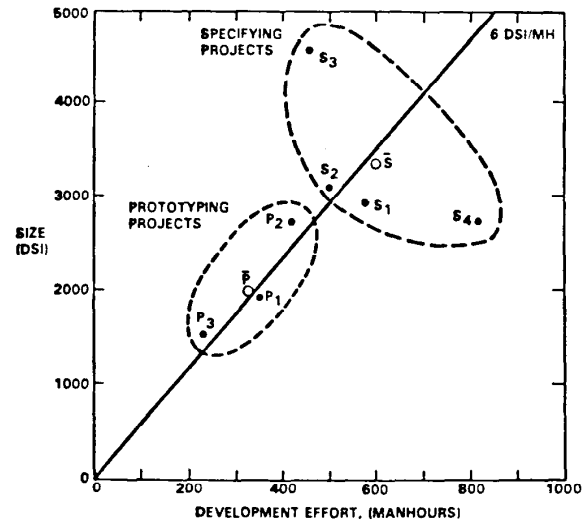

Fig. 1. Prototyping versus specifying size and effort comparisons.

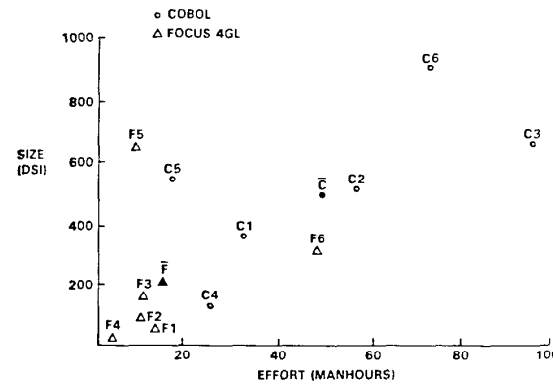

Fig. 2. Fourth-generation language size and effort comparisons.

Implications for Software Productivity Metrics: These two experiments and the earlier Weinberg experiments make it clear that we need better metrics for software productivity than DSI/MH. A number of alternative metrics have been suggested, such as:

- "Software science" or program information-content metrics [58].
- Program control-flow complexity metrics [84].
- Design complexity metrics [36].
- Program-external metrics, such as number of inputs, outputs, files, inquiries interfaces, or function points (a linear combination of those five quantities) [2], [70].
- Work-transaction metrics [38], [114].

In comparing the relative effectiveness of these produc-

tivity metrics to a DSI/MH metric, the following conclusions to date can be advanced:

- Each has advantages over DSI/MH in some situations.
- Each has more difficulties than DSI/MH in some situations.
- Each has equivalent difficulties to DSI/MH in relating software achievement units to measures of the software's value added to the user organization.

Thus, the area of software productivity metrics remains in need of further research and experimentation in search of more robust and broadly relevant metrics.

2) Observational Analyses: Having summarized the major *experimental* investigation of software cost drivers, let us look at the related *observational* studies.

A major early observational analysis of software productivity factors was the study done by SDC for the U.S. Air Force in the mid-1960's [91]. This study collected over 100 attributes of 169 software projects. Although the study was not successful in establishing a definitive set of software cost influence functions robust enough for accurate cost estimation, it did identify some of the more significant candidate influence functions for further investigation, such as requirements and design volatility and concurrent hardware development.

Similar early studies which helped to identify significant candidate software cost influence factors were those of [7] and [124]. As an example, the analysis in [124] yielded a set of quantitative software cost influence factors (number of object instructions, type of application, novelty of application, and degree of difficulty) and relationships which were able to support practical software cost estimates across a range of command-control type applications. Some concurrent studies [123], [21] established a reasonably definitive relationship showing the asymptotic increase in software cost as hardware speed and storage constraints approached 100 percent.

A landmark study in analyzing the effect of modern programming practices on software costs was the IBM [115] study of over 50 software projects. It provided conclusive evidence that the use of such practices as structured code, top-down design, structured walkthroughs, and chief programmer teams correlated with software productivity increases on the order of 50 percent. The study also confirmed the significant impact of such factors as personnel capability and hardware constraints on software productivity, as well as such additional factors as personnel experience and database size.

In the late 1970's number of software cost models were developed, representing a further level of predictive understanding of the factors influencing software costs. Besides the IBM model based on the results in Reference [115], these included the Doty model [61], the Boeing model [11], the SLIM model [96], the RCA PRICE S model [48], and the COCOMO model [23]. More recently, some further software cost estimation models have been developed such as the Jensen model [68], the Estimacs model [103] and the SPQR model [70]. A comparison of these models (except the two most recent models) in terms of their primary cost driver factors, has been provided in [25].

Software Productivity Ranges: In the context of understanding and controlling software costs, a significant feature of some of these models is the *productivity range* for a software cost driver: the relative multiplicative amount by which that cost driver can influence the software project cost estimated by the model. An example of a set of recently updated *productivity ranges* for the COCOMO model is shown in Fig. 3.[1]

Similar productivity ranges have been provided for some other cost models, e.g., [68].

The primary conclusions that can be drawn from the productivity ranges in Fig. 3 are as follows.

- The most significant influence on software costs is the number of source instructions one chooses to program. This leads to cost-reduction strategies involving the use of fourth-generation languages or reusable components to reduce the number of source instructions developed; the use of prototyping and other requirements analysis techniques to ensure that unnecessary functions are not developed, and the use of already-developed software products.
- The next most significant influence by far is that of the selection, motivation, and management of the people involved in the software process. In particular, employing the best people possible is usually a bargain, because the productivity range for people usually is much wider than the range of people's salaries. An overall discussion of the concerns involved here is provided in [23, ch. 33]. More extensive treatments of personnel and management considerations are provided in [120], [33], [88], and [98].
- Some of the factors, such as product complexity, required reliability, and database size, are largely fixed features of the software product and not management controllables. Even here, though, appreciable savings can be achieved by reducing unnecessary complexity, and by focusing on appropriate cost-quality tradeoffs as discussed in Section I.
- Requirements volatility is an important and neglected source of cost savings and control. A great deal can be done in particular in using incremental development to control requirements volatility. Frequently, users request (or demand, or require) new features while a software product is under development. In a single-shot full-product development, it is very hard to refuse these requests; as a result, the developers are continually thrashing as the ripple effects of the changes are propagated through the product (and through the project's highly interlocked schedules). With incremental development, on the other hand, it is relatively easy to say, "Fine, that's a good

[1] The differences between Fig. 3 and its counterpart in [23] are the inclusion of the Requirements Volatility factor, the extension of the Modern Programming Practices range to cover lifecycle costs (using a 30:70 development-maintenance lifecycle cost ratio, this ranges from 1.57 for 2 KDSI products to 1.92 for 512 KDSI products), a widening of the software tools and turnaround time ranges to reflect recent experience with advanced software support environments [20], [26], and the addition of the open-ended range representing the number of software source instructions developed by the project.

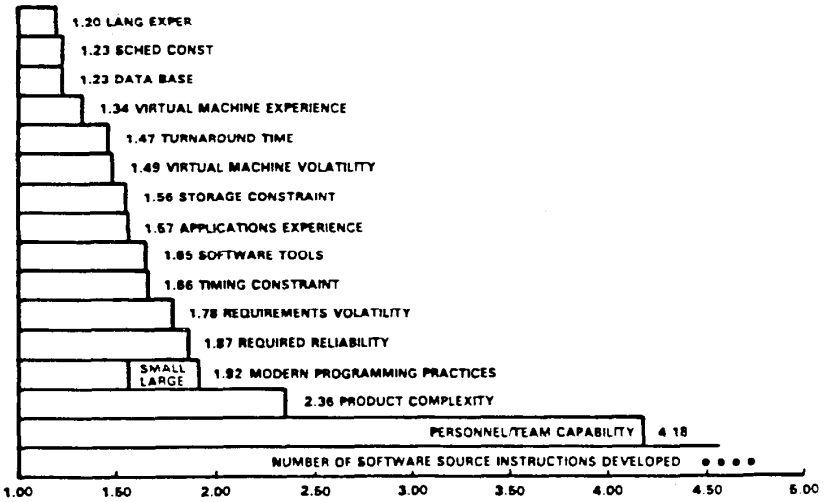

Fig. 3. COCOMO software lifecycle productivity ranges, 1985.

feature. We will schedule it for Increment 4." This allows each increment to operate to a stable plan, thus significantly decreasing the requirements volatility cost escalation factor.

- The other cost driver variables in Fig. 3 are also quite significant particularly if they are addressed in an integrated manner. For more details, see [23, ch. 33] for a discussion of potential productivity strategies for their successful application to an integrated software productivity improvement program.
- The productivity ranges can also be used to assess the impact of other proposed software strategy changes, such as a transition to Ada (and its associated support-environment and modern programming practices). Two such studies have been done for Ada to date. Reference [39], using the COCOMO framework and an expert-consensus approach, estimated a typical 30 percent cost penalty for using Ada in the near term and a cost reduction of at least 40 percent for using Ada in the long-term. Reference [61], using the Jensen-model framework, estimated a significantly larger cost penalty for using Ada in the near term, and a typical 25 percent cost reduction for using Ada in the long term.

B. Software Cost Distribution Insights

Having looked at the experimental and observational "black-box" approaches to understanding software costs, let us now look within the software-production "glass box" for further insight.

There are several approaches to analyzing the distribution of software costs which have provided valuable insights on software cost control. In this section, we will summarize some of the insights gained from analyzing the distribution of:

1) development and rework costs;
2) code and documentation costs;
3) labor and capital costs;
4) software costs by phase and activity.

We will conclude by presenting a particular type of phase and activity distribution called the *value chain*, and show how it leads to a useful characterization of productivity improvement avenues called here the *software productivity opportunity tree*.

1) Development Versus Rework Costs: One of the key insights in improving software productivity is that a large fraction of the effort on a software project is devoted to rework. This rework effort is needed either to compensate for inappropriately-defined requirements, or to fix errors in the specifications, code or documentation. For example, Reference [70] provides data indicating that the cost of rework is typically over 50 percent on very large projects.

A significant related insight is that the cost of fixing or reworking software is much smaller (by factors of 50 to 200) in the earlier phases of the software life cycle than in the later phases [22], [44], [35]. This has put a high premium on early error detection and correction techniques for software requirements and design specification and verification such as the Software Requirements Engineering Methodology, or SREM [3], [4] and the Problem Statement Language/Problem Statement Analyzer [111]. More recently, it has focussed attention on such techniques as rapid prototyping [126], [28], [118] and rapid simulation [125], [109], which focuses on getting the right user requirements early and ensuring that their performance is supportable, thus eliminating a great deal of expensive downstream rework.

Another important point is that rework instances tend to follow a Pareto distribution: 80 percent of the rework costs typically result from 20 percent of the problems. Fig. 4 shows some typical distributions of this nature from recent TRW software projects; similar trends have been indicated in [102], [47], and [13]. The major implication of this distribution is that software verification and validation activities should focus on identifying and eliminating the specific *high-risk* problems to be encountered

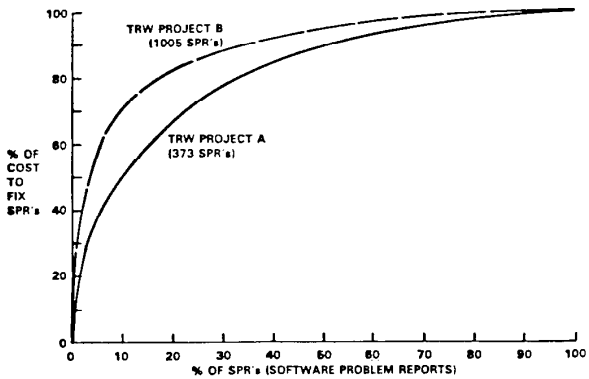

Fig. 4. Rework costs are concentrated in a few high-risk items.

by a software project, rather than spreading their available early-problem-elimination effort uniformly across trivial and severe problems. Even more strongly, this implies that a risk-driven approach to the software lifecycle such as the spiral model [27] is preferable to a more *document-driven* model such as the traditional waterfall model.

The Spiral Model: The spiral model is illustrated in Fig. 5. The radial dimension in Fig. 5 represents the cumulative cost incurred in accomplishing the steps to date; the angular dimension represents the progress made in completing each cycle of the spiral. The model holds that each cycle involves a progression through the same sequence of steps, for each portion of the products and for each of its levels of elaboration, from an overall concept-of-operation document down to the coding of each individual program.

Each cycle of the spiral begins with the identification of:

- The *objectives* of the portion of the product being elaborated (performance, functionality, ability to accommodate change, etc.).
- The *alternative* means of implementing this portion of the product (design A, design B, reuse, buy, etc.).
- The *constraints* imposed on the application of the alternatives (cost, schedule, interface, etc.).

The next step is to *evaluate* the alternatives with respect to the objectives and constraints. Frequently, this process will identify areas of uncertainty which are significant sources of project risk. If so, the next step should involve the *formulation* of a cost-effective strategy for *resolving the sources of risk*. This may involve prototyping, simulation, administering user questionnaires, analytic modeling, or combinations of these and other risk-resolution techniques.

Once the risks are evaluated, the next step is determined by the relative risks remaining. If performance or user-interface risks strongly dominate program development or internal interface-control risks, the next step may be an evolutionary development step: a minimal effort to specify the overall nature of the product, a plan for the next level of prototyping, and the development of a more detailed prototype to continue to resolve the major risk issues. On the other hand, if previous prototyping efforts have already resolved all of the performance or user-interface risks, and program development or interface-control risks dominate, the next step follows the basic waterfall approach, modified as appropriate to incorporate incremental development.

The spiral model also accommodates any appropriate mixture of specification oriented, prototype-oriented, simulation-oriented, automatic transformation oriented, or other approaches to software development, where the appropriate mixed strategy is chosen by considering the relative magnitude of the program risks, and the relative effectiveness of the various techniques in resolving the risks. (In a similar way, risk-management considerations determine the amount of time and effort which should be devoted to such other project activities as planning, configuration management, quality assurance, formal verification, or testing.)

An important feature of the spiral model is that each cycle is completed by a review involving the primary people or organizations concerned with the products. This review covers all of the products developed during the previous cycle, including the *plans for the next cycle* and the resources required to carry them out. The major objective of the review is to ensure that all concerned parties are mutually committed to the approach to be taken for the next phase.

The plans for succeeding phases may also include a *partition* of the product into increments for successive development, or components to be developed by individual organizations or persons. Thus, the *review and commitment* step may range from an individual walkthrough of the design of a single programmer component, to a major requirements review involving developer, customer, user, and maintenance organizations.

2) Code Versus Documentation Costs: Most of the efforts to date in developing software support environments have been focused on capabilities to improve people's productivity in developing code. However, recent analyses have shown that most projects to develop production-engineered software products spend more of the project's effort in activities leading to a document as their immediate end product, as compared to activities whose immediate end product is code. These documents include not only specifications and manuals, but also plans, studies, reports, memoranda, letters and a wide variety of forms. The volume of documentation with respect to lines of code tends to vary by application; Reference [70] reports a typical 28 pages of documentation per thousand instructions (pp/KDSI) for internal commercial programs and a typical 66 pp/KDSI for commercial software products of the same size (50 KDSI).

The proportion of documentation-related to code-related effort averaged about 60:40 over the COCOMO data base of projects [23] and about 67:33 for large TRW projects [20]. These proportions have caused some recent software development environments such as the Xerox Cedar system [112] and the TRW Software Productivity System [20] to focus on the provision of extensive docu-

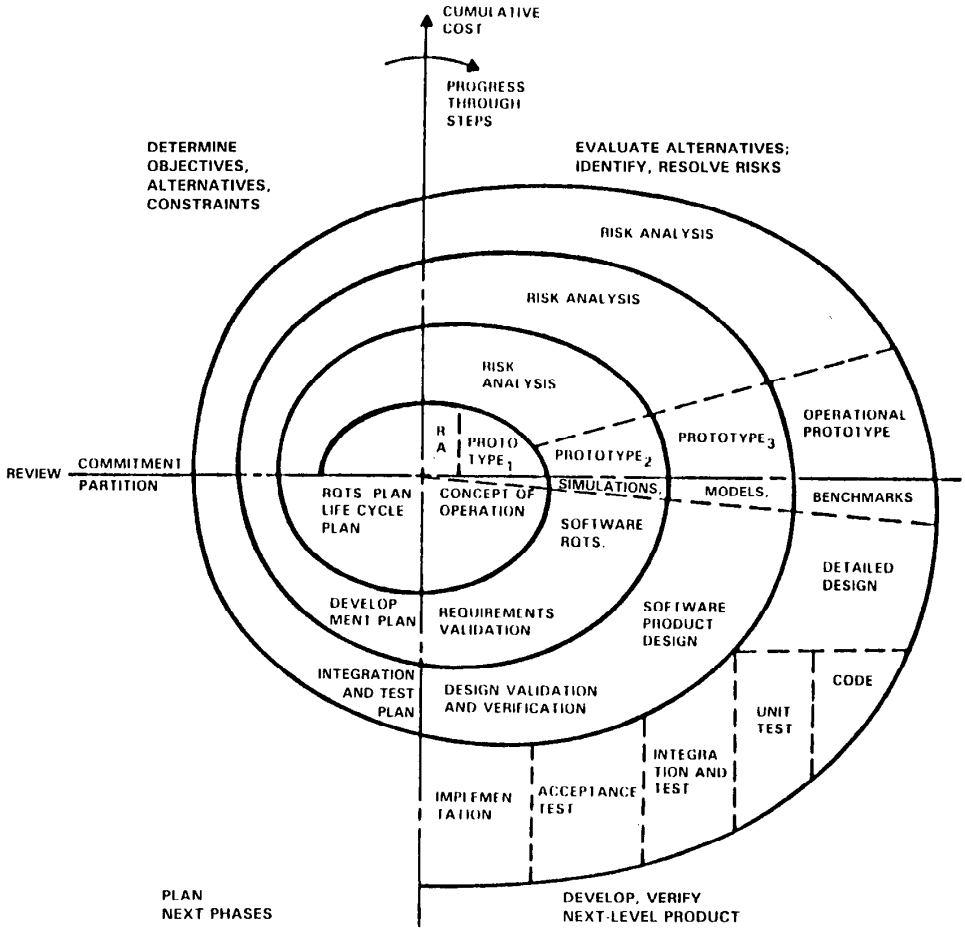

Fig. 5. Spiral model of the software process (not to scale).

mentation and office-automation aids, and on the close integration of these functions with code-oriented functions.

3) Labor Versus Capital Costs: It is generally recognized that software development and evolution are extremely labor-intensive activities, and that a great deal of productivity leverage can be gained by making software production a more capital-intensive activity. Typically, capital investment per software worker has been little different from the $2000–3000 per person typical of office workers in general. However, a number of organizations such as Xerox, TRW, IBM, and Bell Laboratories have indicated that significantly higher investments per person have more than recaptured the investment via improved software productivity. Similar results on the payoffs of capital investments in better facilities and support capabilities have been reported in [81] and [37]. An excellent overall survey of software capitalization strategies is provided in [119].

4) Software Costs by Phase and Activity: A great deal of insight into controlling software costs has come from analyses of the distribution of costs by phase and activity. Some of the earliest results, such as [16], indicated the high proportion of project effort devoted to integration and test, and the importance of good test planning, test support, and interface specification. (Another early paper [63] stated that "a good software interface specification was quite literally worth its weight in gold.")

Subsequent analysis of software development effort distribution such as [124] indicated the significant fraction of project effort devoted to nonprogramming activities (configuration management, quality assurance, planning and control, etc.), and the high potential leverage involved in making these activities more productive.

Another major insight has been the recognition that most of the cost of a software product is incurred after its initial development is complete [43], [22], [35]. Subsequent analyses of the sources and distribution of these software life-cycle evolution costs (often misleadingly called maintenace costs) such as [14] and [79], provided a number of insights on how to reduce software evolution costs. Several recent sources such as [2], and [6] have provided more specific detail on software evolution cost reduction activities.

5) The Software Product Value Chain: The value

chain, developed by Porter and his associates at the Harvard Business School [94], [95], is a useful method of understanding and controlling the costs involved in a wide variety of organizational enterprises. It identifies a canonical set of cost sources or value activities, representing the basic activities an organization can choose from to create added value for its products. Fig. 6 shows a value chain for software development representative of experiment at TRW. Definitions and explanations of the component value activities are given below. These are divided into what Reference [95] calls primary activities (inbound logistics, outbound logistics, marketing and sales, service, and operations) and support activities (infrastructure, human resource management, technology development, and procurement).

Primary Activities: Inbound logistics covers activities associated with receiving, storing, and disseminating inputs to the products. This can be quite large for a manufacturer of, say, automobiles; for software it consumes less than 1 percent of the development outlay. (For software, the related support activity of *procurement* is also included here.)

Outbound logistics covers activities concerned with collecting, storing, and physically distributing the product to buyers. Again, for software, this consumes less than 1 percent of the total.

Marketing and sales covers activities associated with providing a means by which buyers can purchase the product and inducing them to do so. A 5 percent figure is typical of government contract software organizations. Software product houses would typically have a higher figure; internal applications-programming shops would typically have a lower figure.

Service covers activities associated with providing service to enhance or maintain the value of the product. For software, this comprises the activities generally called software maintenance or evolution.

Operations covers activities associated with transforming inputs into the final product form. For software, operations typically involves roughly four-fifths of the total development outlay.

In such a case, the value-chain analysis involves breaking up a large component into constituent activities. Fig. 6 shows such a breakup into management (7 percent), quality assurance and configuration management (5 percent), and the distribution of technical effort among the various development phases. This phase breakdown also covers the cost sources due to rework. Thus, for, example, of the 20 percent overall cost of the technical effort during the integration and test phase, 13 percent is devoted to activities required to rework deficiencies in or reorientations of the requirements, design, code, or documentation; the other 7 percent represents the amount of effort required to run tests, perform integration functions, and complete documentation even if no problems were detected in the process.

Support Activities: Infrastructure covers such activities as the organization's general management planning,

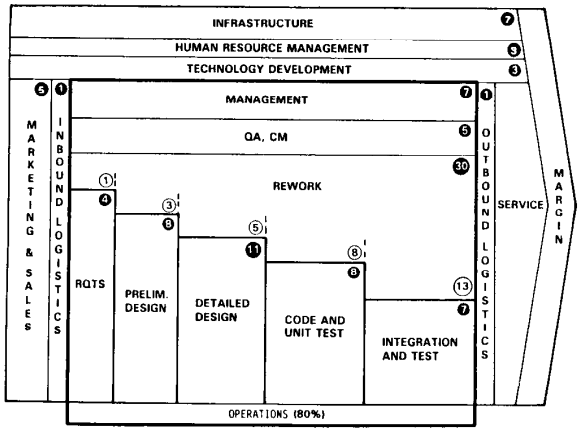

Fig. 6. Software development value chain.

finance, accouting, legal, and government affairs of the organization. The 8 percent figure is typical of most organizations.

Human resource management covers activities involved in recruiting, hiring, training, development, and compensation of all types of personnel. Given the labor-intensive and technology-intensive nature of software development, the 3 percent figure indicated here is a less-than-optimal investment.

Technology development covers activities devoted to creating or tailoring new technology to improve the organizations products or processes. The 3 percent investment figure here is higher than many software organizations, but still less than optimal as an investment to improve software productivity and quality.

Margin and Service: Margin in the value chain is the difference between the value of the resulting product and the collective costs of performing the value activities. As this difference varies widely among software products, it is not quantitatively defined in Fig. 6. Similarly, "service" or "evolution" costs have not been assigned a value in Fig. 6. Evolution costs are typically 70 percent of software lifecycle costs, but since some initial analyses have indicated that the detailed value chain distribution of software costs is not markedly different from the distribution of development costs in Fig. 6, we will use Fig. 6 to represent the distribution of lifecycle costs.

Software Development Value Chain Implications: The primary implication of the software development value chain is that the "Operations" component is the key to significant improvements. Not only is it the major source of software costs, but also most of the remaining components such as "Human Resources" will scale down in a manner proportional to the scaling down of Operations costs.

Another major characteristic of the value chain is that virtually all of the components are still highly labor-intensive. Thus, as discussed in Section II-B-3, there are significant opportunities in providing automated aids to make these activities more efficient and capital-investive.

Further, it implies that human-resource and management activities have much higher leverage than their 3 percent and 7 percent investment levels indicate.

The breakdown of the Operations component indicates that the leading strategies for cost savings in software development involve:

- *Making individual steps more efficient*, via such capabilities as automated aids to software requirements analysis or testing.
- *Eliminating steps*, via such capabilities as automatic programming or automatic quality assurance.
- *Eliminating rework*, via early error detection or via such capabilities as rapid prototyping to avoid later requirements rework.

In addition, further major cost savings can be achieved by reducing the total number of elementary Operations steps, by developing products requiring the creation of fewer lines of code. This has the effect of reducing the overall size of the Value Chain itself. This source of savings breaks down into two primary options:

- *Building simpler products*, via more insightful front-end activities such as prototyping or risk management.
- *Reusing software components*, via such capabilities as fourth-generation languages or component libraries.

6) The Software Productivity Improvement Opportunity Tree: This breakdown of the major sources of software cost savings leads to the *Software Productivity Improvement Opportunity Tree* shown in Fig. 7. This hierarchical breakdown helps us to understand how to fit the various attractive productivity options into an overall integrated software productivity improvement strategy.

Most of the individual productivity options have been discussed in earlier sections of this paper. Here, we will provide a recap of the previous options, and further discussion of the additional options identified in the Opportunity Tree.

Making People More Effective: The major sources of opportunity in dealing with people were covered in discussing the large productivity range due to personnel capability in Section II-A-2, and the labor versus capital costs discussion in Section II-B-3. Additional facilities-oriented gains were covered in the discussions of interactive software development in Section II-A-1, and of avoiding hardware constraints in Section II-A-1. Providing software personnel with private offices is another cost-effective facilities opportunity, leading to productivity gains of roughly 11 percent at IBM-Santa Teresa [70] and 8 percent at TRW [20]. In addition, the productivity leverage of *creative incentive structures* can be quite striking. For example, a program to provide extra bonuses for people who reuse rather than rebuild software has led to significant increases in the amount of software reused from previous applications.

Making Steps More Efficient: The primary leverage factor in making the existing software process steps more efficient is the use of software tools to automate the current repetitive and labor-intensive portions of each step. Such tools have a long history of development; some good surveys of various classes of tools are given in [74] and [99].

More recently, it has become clear that such tools are much more effective if they are part of an *Integrated Project Support Environment* (IPSE). The primary features which distinguish an IPSE from an ad hoc collection tools are as follows:

- A *set of common assumptions* about the software process model being supported by the tools (or, more strongly, a particular software development method being supported by the tools).
- An integrated *Project Master Database* or *Persistent Object Base* serving as a unified repository of the entities created during the software process, along with their various versions, attributes, and relationships.
- *Support of the entire range of users and activities* involved in the software project, not just of programmers developing code.
- A *unified user interface* providing easy and natural ways for various classes of project personnel (expert programmers, novice librarians, secretaries, managers, planning and control personnel, etc.) to draw on the tools in the IPSE.
- A *critical-mass ensemble of tools*, covering significant portions of software project activities.
- *A computer-communication architecture* facilitating user access to data and resources in the IPSE.

Some good references describing the nature and functions of IPSE's are [32], [117], [64], [90], and [110]. Some good examples of IPSE's with extensive usage experience include CADES [87], Interlisp [113], the AT&T Unix environment [73], the U.S. Navy FASP system [108], the TRW Software Productivity System [20], and the Xerox Cedar System [112]. Some early examples of advanced concepts and prototype environments are found in [117]. Later examples are so abundant that it is virtually impossible to summarize them concisely; a good recent source is [12].

Eliminating Steps: A good many automated aids go beyond simply making steps more efficient, to the point of fully eliminating previous manual steps. If we compare software development today with its counterpart in the 1950's, we see that *assemblers and compilers* are excellent examples of ways of vastly improving productivity by eliminating steps. More recent examples of eliminating steps are process construction systems [122], [45], software standards checkers and other quality assurance functions [19], [106]; and requirements and design consistency checkers [3], [15], [111].

More ambitious efforts to eliminate steps involve the automation of the entire programming process, by providing capabilities which operate directly on a set of software specifications to automatically generate computer programs. There are two major branches to this approach: *domain-specific* and *domain-independent* automatic programming.

The domain-specific approach gains advantages by capitalizing on domain knowledge in transforming specifi-

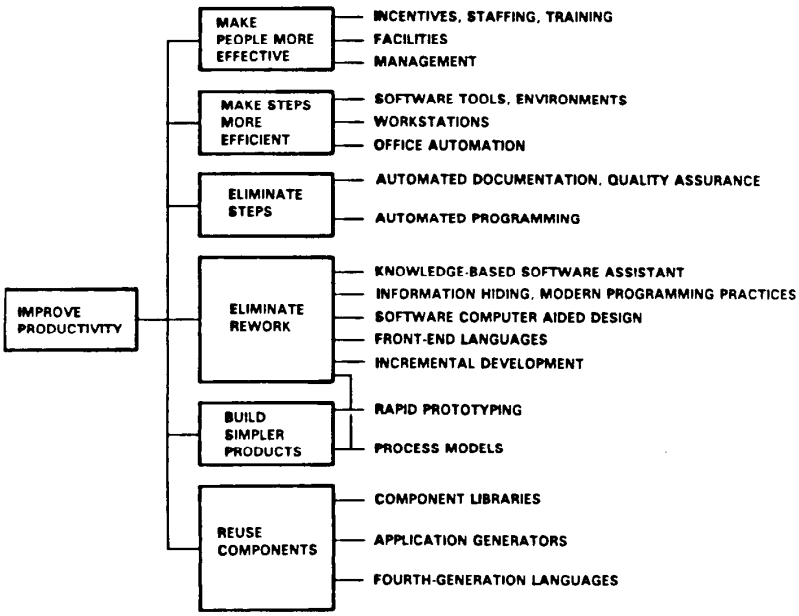

Fig. 7. Productivity improvement opportunity tree.

cations into programs, and in constraining the universe of programming discourse to a relatively smaller domain. In the limit, one reaches the boundary with fourth-generation languages such as Visicalc, which are excellent automatic programming systems within a very narrow domain, and relatively ineffective outside that domain. A good example and survey of more general approaches to domain-specific automatic programming is given in [11].

The domain-independent approach offers much broader payoff in the long run, but has more difficulty in achieving efficient implementations of larger-scale programs. Some good progress is being made in this direction, such as the USC-ISI work culminating in the FSD system [9], the Kestrel Institute work on the PSI and and CHI systems [56], [105], and the MIT Progammer's Apprentice project [101], [118]. An excellent summary of automatic programming approaches can be obtained from the November 1985 issue of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING.

Eliminating Rework: One can also extend automatic programming in a direction which provides expert assistance to programmers (and more generally, to all software project members) to aid them in making the right decisions in algorithm selection, data structuring, choice of reusable components, change control, test planning, and overall software project planning and control. This concept of a *knowledge based software assistant* (KBSA) has been thoroughly described in [55]. The primary benefit of a KBSA will be the elimination of much of the rework currently experienced on software projects due to the belated appreciation that a previous programming or project decision was inappropriate, resulting in work that needs to be redone. A number of prototype KBSA's are currently under development.

If we specialize the KBSA concept of the area of software design, we find the rich area of *software computer aided design* (CAD). In the hardware area, CAD has been a major source of improving productivity by eliminating rework via automated design checking and simulation, and also of promoting better designs via better visualization of a design and its effects. Recent examples of software CAD capabilities include *interactive graphics support systems* such as the Xerox CEDAR system [112], the Brown PECAN system [100], the Carleton CAEDE system [31], and such commercial systems as Excelerator, Teamwork, ProMod, Software Through Pictures, CASE, Ada Graph, and PRISM; *rapid simulation capabilities* such as RSA [109]; and executable specification capabilities such as PAISLEY [125].

A short step from software CAD systems are the *requirements and design language-oriented systems*, which eliminate a great deal of rework through more formal and unambiguous specifications, automated consistency and completeness checking and automated traceability of requirements to design. Probably the most extensive of these systems is the Distributed Computing Design System [4], which includes a system specification language (SSL), a software requirements specification language (RSL), a distributed-system design language (DDL), and a module description language (MDL).

One of the main difficulties in developing good software CAD systems is our incomplete understanding of the software design process. Examples of recent progress in this direction can be found in [34], [1], and [71].

A most powerful technique for eliminating rework is the *information-hiding* approach developed by Parnas [92] and applied in the U.S. Navy A-7 project [93]. This approach minimizes rework by hiding implementation de-

cisions within modules; thus minimizing the ripple effects usually encountered when software implementation decisions need to be changed. The information hiding approach can be particularly effective in eliminating rework during software evolution, by *identifying the portions of the software most likely to undergo change* (characteristics of workstations, input data formats, etc.) and hiding these sources of evolutionary change within modules.

Some other sources for eliminating rework have been discussed earlier, such as the use of *modern programming practices* in Section II-C and II-A-2, the use of *incremental development* to reduce requirements volatility in Section II-A-2, and the use of *rapid prototyping* and risk-driven *software process models* in the discussion of development versus rework costs in Section II-B-1.

Building Simpler Products: The last two approaches associated with eliminating rework in the Opportunity Tree in Fig. 7, rapid prototyping and improved software process models can also be very effective in improving bottom-line productivity by building simple products. This is done largely by eliminating software gold-plating: extra software which not only consumes extra effort, but also reduces the conceptual integrity of the product. The [28] prototyping versus specifying experiment discussed in Section II-A-1 indicated that prototyping resulted in an average of 40 percent less code, 40 percent less effort, and a set of products that were easier to use and learn. One of the telling insights in this experiment was the comment of one of the participants using the specification approach: "Words are cheap." During the specification phase, it is all too easy to add gold-plating functions to the product specification, without a good understanding of their effect on the product's conceptual integrity or the project's required effort. As expressed in the excellent book, *The Elements of Friendly Software Design* [60]:

> "Most programmers . . . defined their use of a software feature by saying, 'You don't have to use it if you don't want to, so what harm can it do?' It can do a great deal of harm. The user might spend time trying to understand the feature, only to decide it isn't needed, or he may accidentally use the feature and not know what has happened or how to get out of the mistake. If a feature is inconsistent with the rest of the user interface, the user might draw false conclusions about the other commands. The feature must be documented, which makes the user's manual thicker. The cumulative effect of such features if to overwhelm the user and obscure communication with your program . . ."

A further discussion of typical sources of software gold-plating, and an approach for evaluating potential gold-plating features, is provided in [23]. A related phenomenon to avoid is the "second system syndrome" discussed in [29]. A recent useful technique for product feature prioritization called the request-success grid is provided in [107]. Further useful principles of good user-interface design are provided in [41] and [53].

Some of the newer *software process models* stimulate the development of simpler products. One of the difficulties of the traditional waterfall model is that its specification-driven approach can frequently lead one along the "words are cheap" road toward gold-plated products, as discussed above. The evolutionary development model [86] emphasizes the use of prototyping capabilities to converge on the necessary or high-leverage software product features essential to the user's mission. The related transformational model [10] shortcuts the problem by providing (where available) a direct transformation from specification to executing code, thus supporting both a specification-based and an evolutionary-development approach. The spiral model [27] focuses on a continuing determination of user's mission objectives, and a continuing cost-benefit analysis of candidate software product features in terms of their contribution to mission objectives. Further information on recent progress in software process models can be found in [77] and [40].

Reusing Components: Another key to improving productivity by writing less code involves the reuse of existing software components. The simplest approach in this direction involves the development and use of *libraries of software components*. A great deal of progress has been made in this direction, particularly in such areas as mathematical and statistical routines and operating system related utilities. A great deal of further progress is possible via similar capabilities in user-application areas. For example, Raytheon's library and system of reusable business-application components has achieved typical figures of 60 percent reusable code for new applications [75] and typical cost savings of 10 percent in the design phase, 50 percent in the code and test phase, and 60 percent in the maintenance phase [97]. Toshiba's system of reusable components for industrial process control [83] has resulted in typical productivity rates of over 2000 source instructions per man-month for high-quality industrial software products.

At this level of sophistication, such systems should better be called *application generators*, rather than component libraries, because they have addressed several system-oriented component-compatibility issues such as component interface conventions, data structuring, and program control and error handling conventions. Similar characteristics have made Unix a particularly strong foundation for developing *application generators* [72], [116].

One can proceed even further in this direction to create a *very high level language or fourth generation language* (4GL) by adding a language for specifying desired applications and a set of capabilities for interpreting user specifications, configuring the appropriate set of components, and executing the resulting program. Currently, the most fertile areas for 4GL's are in the areas of spreadsheet calculators (Visicalc, Multiplan, 1-2-3, etc.), and small-business systems typically featuring a DBMS, report generator, database query language, and graphics package (NOMAD, RAMIS, FOCUS, ADF, DBase II, etc). A good survey of these latter 4GL's is [62].

As discussed in Section II-A-1, the most definitive experiment to date comparing a 3GL (COBOL) and a 4GL

(FOCUS) found an average reduction of about 60 percent in both lines of code developed and in manhours expended to develop a sample of six applications. Reference [57] provides further evidence from a survey of 43 organizations that such 4GL's reduce personnel costs, reduce user frustrations, and more quickly satisfy user information needs within their domain of applicability. On the other hand, the survey found 4GL's extremely inefficient of computer resources and difficult to interface with conventional applications programs. Some major disasters have occurred in attempting to apply 4GL's to large, high-performance applications such as the New Jersey motor vehicle registration system [8].

Overall, though, 4GL's offer an extremely attractive option for significantly improving software productivity, and attempts are underway to create 4GL capabilities for other application areas. Short of a 4GL capability, the other more limited approaches to reusability such as component libraries and application generators can both generate near-term cost savings and serve as a foundation of more ambitious 4GL capabilities in the long run. A very good collection of articles on reusability in software development is the September 1984 issue of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING.

III. CONTROLLING SOFTWARE COSTS

Now that we have a better understanding of the primary sources of software costs and of the ways of reducing them, how can we use this understanding to improve our ability to control software costs? There are two primary avenues for doing this, as discussed below:

1) Building our understanding into a framework of objectives, which serve as a basis for a set of management-by-objectives (MBO) control loops.

2) Optimizing our software development and evolution strategy around predictability and control.

A. Management By Objectives (MBO)

The simplest sort of MBO for software project predictability and control is exemplified by the earned-value framework discussed in [23, ch. 32], and illustrated in Fig. 8. In this framework, a set of cost and schedule estimates by phase, activity, and product components are used to generate a set of PERT charts, work breakdown structures, personnel plans, summary task planning sheets, and other scarce-resource allocations which determine a set of "should-cost" targets for each job. As the project progresses, various instruments such as unit development folders and earned value systems are used to compare actual progress and expenditure of time, cost, personnel, or other scarce resources versus the plans. Then, comparing the actual progress and expenditure versus the plans can generate a set of exception reports which flag key areas for MBO attention.

This generic approach has been highly successful in many situations, but it frequently needs extension to balance cost, schedule, and functionality objectives with other important quality-oriented objectives. The best approach to date in handling these additional objectives has been to incorporate them as additional specific MBO targets, as in Design by Objectives [51] and the GOALS approach [23].

Actually, it is even better to do this in terms of the software end-user's mission objectives. This implies that the users must perform an analysis of the relative costs and benefits of alternative software product functions and features, to relate these to incremental gains in mission cost-effectiveness, and to use this information in an overall MBO control loop in which the software is only a part. For examples of this type of approach, see [80], [5], [65], and [76].

B. Optimizing Around Software Predictability and Control

Frequently, software customers are more concerned about predictability and control of software cost and schedule than they are about the absolute values of the cost and schedule [89]. Such customers prefer a project which may cost a bit more, but which allows them to confidently synchronize their software development with other critical developments such as a satellite launch, a factory opening, or a major service cutover. In such situations, customers will generally prefer a risk-driven development approach which invests some additional early time and effort into identifying and eliminating the primary sources of project risk—as contrasted with a "success-oriented" approach which will be very efficient if all the project's optimistic assumptions are true, but very costly if reality runs out otherwise (as it frequently does). The spiral model discussed in Section II-B-1 is an example of such a risk-driven development approach.

Another option which can be derived from the risk-driven spiral approach is the option to trade marginal product functionality for project predictability and control, using a design-to-cost or design-to-schedule approach. Thus, if the highest project risk is associated with exceeding the available budget or with missing a crucial delivery date, the project can reduce risk by designating borderline product capabilities as a management reserve to be traded against budget and schedule pressures as necessary.

IV. CONCLUSIONS

The information and discussions above support the following primary conclusions:

1) Understanding and controlling software costs is extremely important, not just from an economic standpoint, but also in terms of our future quality of life.

2) Understanding and controlling software costs inevitably requires us to understand and control the various aspects of software quality as well.

3) There are two primary ways of understanding software costs. The "black box" or influence function approach provides useful insights on the relative productivity and quality leverage of various management, technical,

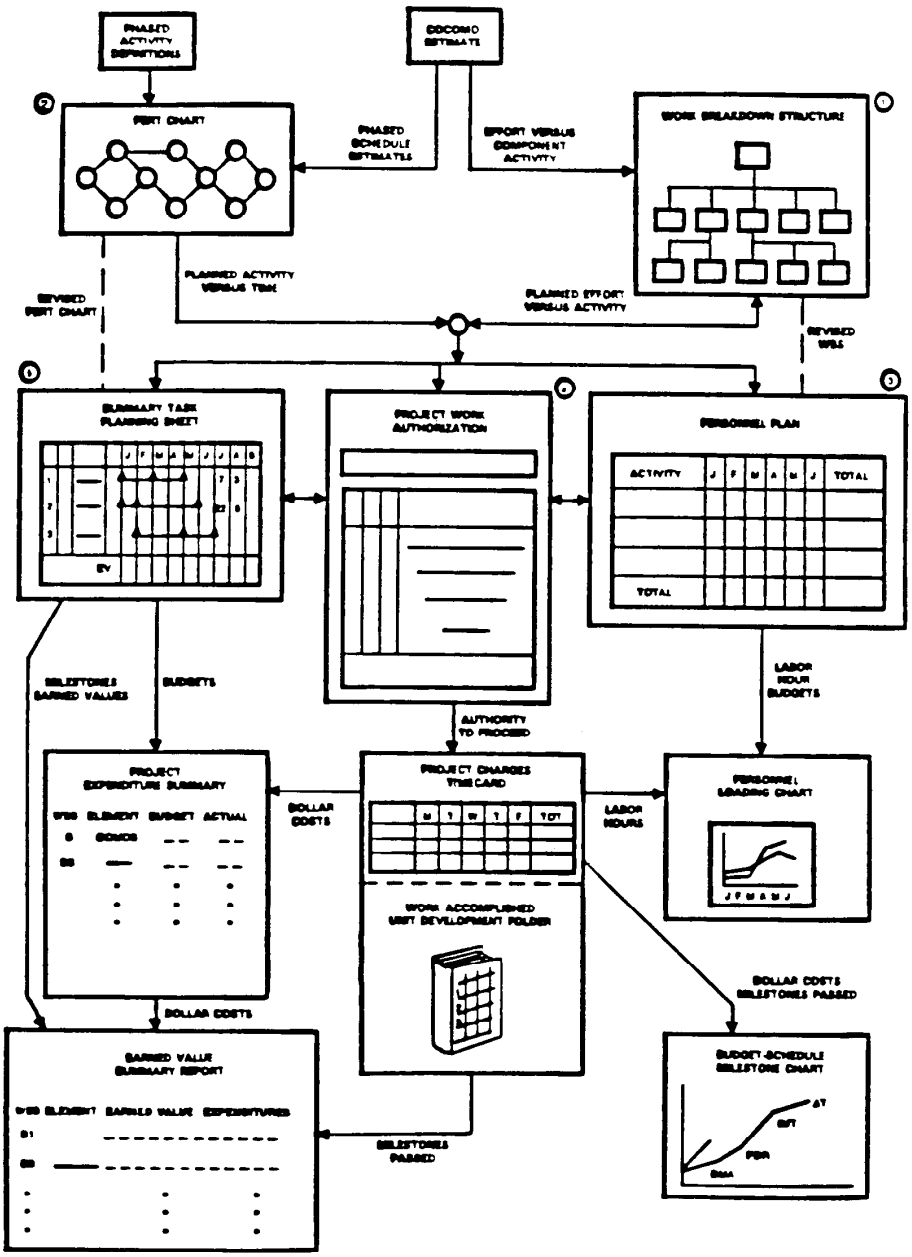

Fig. 8. Software project planning and control framework.

environment, and personnel options. The "glass box" or cost distribution approach helps identify strategies for integrated software productivity and quality improvement programs, via such structures as the *value chain* and the *software productivity opportunity tree*.

4) The most attractive individual strategies for improving software productivity are:

- *writing less code*, by reusing software components, developing and using very high level languages, and avoiding software gold-plating;
- *getting the best from people*, via better management, staffing, incentives, and work environments;
- *avoiding rework*, via better risk management, prototyping, incremental development, software computer aided design, and modern programming practices, particularly information hiding;
- *developing and using integrated project support environments*.

5) Good frameworks of techniques exist for controlling software budgets, schedules, and work completed. There

have been some initial attempts to extend these to support control with respect to software quality objectives and end-user system objectives, but a great deal more progress is needed in these directions.

6) The better we are able to understand software cost and qualities, the better we are able to control them—and vice versa.

References

[1] B. Adelson and E. Soloway, "The role of domain experience in software design," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1351-1360, Nov. 1985.

[2] A. J. Albrecht, "Measuring application development productivity," in *Proc. SHARE-GUIDE Applications Develoment Symp.*, Oct. 1979, pp.83-92.

[3] M. W. Alford, "A requirements engineering methodology for real-time processing requirements," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 60-68, Jan. 1977.

[4] ——, "SREM at the age of eight: The distributed computing design system," *Computer*, vol. 18, Apr. 1985.

[5] J. Allen and B. P. Lientz, *Systems in Action: A Managerial and Social Approach*, Goodyear, 1978.

[6] R. S. Arnold, Ed., *Software Maintenance Workshop Record*, IEEE, Dec. 1983.

[7] J. D. Aron, "Estimating resources for large programming systems," NATO Science Committee, Rome, Italy, Oct. 1969; in *Software Engineering Techniques*, Buxton and Randell, Eds.

[8] C. Babcock, "New Jersey motorists in software jam," *Computerworld*, pp. 1, 6, Sept. 30, 1985.

[9] R. M. Balzer, "A 15 year perspective on automatic programming," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1357-1268, Nov. 1985.

[10] R. M. Balzer, T. E. Cheatham, and C. Green, "Software technology in the 1990's: Using a new paradigm," *Computer*, vol. 16, pp. 39-45, Nov. 1983.

[11] D. R. Barstow, "Domain-specific automatic programming," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1321-1336, Nov. 1985.

[12] D. R. Barstow, H. Shrobe, and E. Sandewall, *Interactive Programming Environments*. New York: McGraw-Hill, 1984.

[13] V. R. Basili and D. M. Weiss, "Evaluation of a software requirements document by means of change data," in *Proc. Fifth Int. Conf. Software Engineering*, IEEE, Mar. 1981, pp. 314-323.

[14] L. A. Belady and M. M. Lehman, "Characteristics of large systems," *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: MIT Press, 1979.

[15] T. E. Bell, D. C. Bixler, and M. E. Dyer, "An extendible approach to computer-aided software requirements engineering," *IEEE Trans. Software Eng.*, pp. 49-59, Jan. 1977.

[16] H. D. Benington, "Production of large computer programs," in *Proc. ONR Symp. Advanced Programming Methods for Digital Computers*, June 1956, pp. 15-27; also in *Proc. 9th Int. Conf. Software Engineering*, IEEE, Mar. 1987.

[17] R. K. D. Black, R. P. Curnow, R. Katz, and M. D. Gray, "BCS software production data," Boeing Computer Services, Inc., Final Tech. Rep. RADC-TR-77-116, NTIS No. AD-A039852, Mar. 1977.

[18] B. H. Boar, *Application Prototyping*. New York: Wiley, 1984.

[19] B. H. Boehm, J. R. Brown, H. Kaspar, M. Lipow, E. J. MacLeod, and M. J. Merritt, *Characteristics of Software Quality*. Amsterdam, The Netherlands: North-Holland, 1978.

[20] B. W. Boehm, M. H. Penedo, E. D. Stuckle, R. D. Williams, and A. H. Pyster, "A software development environment for improving productivity," *Computer*, vol. 17, pp. 30-44, June 1984.

[21] B. W. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, pp. 48-59, May 1973.

[22] ——, "Software engineering," *IEEE Trans. Comput.*, vol. C-25, pp. 1226-1241, Dec. 1976.

[23] ——, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[24] ——, The hardware/software cost ratio: Is it a myth?" *Computer*, vol. 16, pp. 78-80, Mar. 1983.

[25] ——, "Software engineering economics," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 4-21, Jan. 1984.

[26] ——, "COCOMO: Answering the most frequent questions," in *Proc. COCOMO Users' Group*, Wang Institute, May 1985.

[27] ——, "A sprial model of software development and enhancement," *Computer*, vol. 21, pp. 61-72, May 1988.

[28] B. W. Boehm, T. E. Gray, and T. Seewaldt, "Prototyping vs. specifying: A multi-project experiment," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 133-145, May 1984.

[29] F. P. Brooks, Jr., *The Mythical Man-Month*. Reading, MA: Addison Wesley, 1975.

[30] J. R. Brown and M. Lipow, "The quantitative measurement of software safety and reliability," TRW Rep. QR 1776, Aug. 1973.

[31] R. J. A. Buhr, C. M. Woodside, G. M. Karam, K. Van Der Loo, and G. D. Lewis, "Experiments with Prolog design descriptions and tools in CAEDE: An iconic design environment for multitasking, embedded systems," in *Proc. 8th Int. Conf. Software Engineering*, Aug. 1985, pp. 62-67.

[32] J. Buxton, "Requirements for Ada programming support environments: 'Stoneman'," U.S. Dep. Defense, OSD/R&E, Washington, DC, Feb. 1980.

[33] J. D. Couger and R. A. Zawacki, *Motivating and Managing Computer Personnel*. New York: Wiley, 1980.

[34] B. Curtis, "Fifteen years of psychology in software engineering: Individual differences and cognitive science," in *Proc. 7th Int. Conf. Software Engineering*, Mar. 1984, pp. 97-106.

[35] E. B. Daly, "Management of software engineering," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 229-242, May 1977.

[36] T. De Marco, *Controlling Software Projects*. New York: Yourdon, 1982.

[37] T. A. De Marco and T. Lister, "Programmer performance and the effects of the workplace," in *Proc. 8th Int. Conf. Software Engineering*, Aug. 1985, pp. 268-272.

[38] W. J. Doherty and R. P. Kelisky, "Managing VM/CMS for user effectiveness," *IBM Syst. J.*, vol. 18, no. 1, pp. 143-163, 1979.

[39] A. Douville, J. Salasin, and T. H. Probert, "Ada impact on COCOMO workshop report," *Inst. Defense Analysis*, May 1985.

[40] M. Dowson and J. C. Wileden, Ed., *Proc. Second Software Process Workshop (ACM Software Eng. Notes)*, Aug. 1986.

[41] S. W. Draper and D. A. Norman, "Software engineering for user interfaces," *IEEE Trans. Software Eng.*, vol. SE-11, Mar. 1985.

[42] Electronic Industries Association, "DoD computing activities and programs: Ten year market forecast issues, 1985-1995," Oct. 1985.

[43] J. L. Elshoff, "An analysis of some commercial PL/I programs," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 113-120, June 1976.

[44] M. R. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 182-211, 1976.

[45] S. I. Feldman, "MAKE—A program for maintaining computer programs," *Unix Programmers' Manual*, vol. 9, pp. 255-265, Apr. 1979.

[46] D. Fisher, "Software costs in the Department of Defense," IDA Rep. R-1079, 1974.

[47] G. Formica, "Software management by the European space agency: Lessons learned and future plans," in *Proc. Third Int. Software Management Conf.*, AIAA/RAeS, London, Oct. 1978, pp. 15-35.

[48] F. R. Freiman and R. E. Park, "PRICE software model version 3: An overview," in *Proc. IEEE-PINY Workshop Quantitative Software Models*, IEEE Catalog No. TH0067-9, Oct. 1979, pp. 32-41.

[49] E. Frewin, P. Hamer, B. Kitchenham, N. Ross, and L. Wood, "Quality measurement and modeling—State of the art report," ESPRIT Rep. REQUEST/STC-gdf/001/51/QL-RP/00.7, July 1985.

[50] "GUIDE survey of new programming technologies," *Guide Proc.*, GUIDE, Inc., Chicago, IL, pp. 306-308, 1979.

[51] T. Gilb, *Design by Objectives*. Amsterdam, The Netherlands: North-Holland, 1985.

[52] R. L. Glass and R. A. Noiseux, *Software Maintenance Guidebook*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[53] J. D. Gould and C. Lewis, "Designing for usability: Key principles and what designers think," *Commun. ACM*, pp. 300-311, Mar. 1985.

[54] E. Grant and H. Sackman, "An exploratory investigation of programmer performance under on-line and off-line conditions," System Development Corp., Rep. SP-2581, Sept. 1966.

[55] C. C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich, "Report on a knowledge-based software assistant," USAF/RADC Rep. RADC-TR-195, Aug. 1983.

[56] C. C. Green, "The design of the PSI program synthesis system," in *Proc. 2nd Int. Conf. Software Engineering*, Oct. 1976, pp. 4-18.

[57] T. Guimaraes, "A study of application program development techniques," *Commun. ACM*, pp. 494-499, May 1985.

[58] M. H. Halstead, *Elements of Software Science*. New York: Elsevier, 1977.
[59] E. Harel and E. R. Mc Lean, "The effects of using a nonprocedural language on programmer productivity," UCLA Grad. School Management, Inform. Syst. Working Paper 3-83, Nov. 1982.
[60] P. Heckel, *The Elements of Friendly Software Design*. Warner Books, 1984.
[61] J. R. Herd, J. N. Postak, W. E. Russel, and K. R. Stewart, "Software cost estimation study—Study results," Doty Associates, Inc., Rockville, MD, Final Tech. Rep. RADC-TR-77-220, Vol. I (of two), June 1977.
[62] E. Horowitz, A. Kemper, and B. Narasimhan, "A survey of application generators," *IEEE Software*, vol. 2, pp. 40–54, Jan. 1985.
[63] W. A. Hosier, "Pitfalls and safeguards in real-time digital systems with emphasis on programming," *IRE Trans. Eng. Management*, pp. 99–115, June 1961; in *Proc. 9th Int. Conf. Software Engineering*, IEEE, Mar. 1987.
[64] H. Hunke, Ed., *Software Engineering Environments*. Amsterdam, The Netherlands: North-Holland, 1981.
[65] M. A. Jackson, *System Development*. Englewood Cliffs, NJ: Prentice-Hall, 1983.
[66] R. W. Jensen, "An improved macrolevel software development resource estimation model," in *Proc. 5th ISPA Conf.*, Apr. 1983, pp. 88–92.
[67] —, "Projected productivity impact of near-term Ada use in software system development," in *Proc. 7th ISPA Conf.*, May 1985.
[68] R. W. Jensen and S. Lucas, "Sensitivity analysis of the Jensen software model," in *Proc. 5th ISPA Conf.*, Apr. 1983, pp. 384–389.
[69] T. C. Jones, "Demographic and technical trends in the computing industry," Software Productivity Research, Inc., July 1983.
[70] —, *Programming Productivity*. New York: McGraw-Hill, 1986.
[71] E. Kant, "Understanding and automating algorithm design," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1361–1374, Nov. 1985.
[72] B. W. Kernighan, "The Unix system and software reusability," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 513–518, Sept. 1984.
[73] B. W. Kernighan and J. R. Mashey, "The Unix programming environment," *Computer*, vol. 14, pp. 12–24, Apr. 1981.
[74] B. W. Kernighan and P. J. Plauger, *Software Tools*. Reading, MA: Addison-Wesley, 1976.
[75] R. G. Lanegran and C. A. Grasso, "Software engineering with reusable design and code," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 498–501, Sept. 1984.
[76] J. Z. Lavi, "A systems engineering approach to software engineering," in *Proc. IEEE Software Workshop*, Feb. 1984, pp. 49–57.
[77] M. M. Lehman, V. Stenning, and C. Potts, Eds., *Proc. Software Process Workshop*, IEEE, Feb. 1984.
[78] E. Lieblein, "STARS program overview," in *Proc. DoD/Industry STARS Workshop*, EIA, May 1985.
[79] B. P. Lientz and E. B. Swanson, *Software Maintenance Management: A study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Reading, MA: Addison-Wesley, 1980.
[80] M. Lundeberg, G. Goldkuhl, and A. Nilsson, *Information Systems Development: A Systematic Approach*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[81] J. H. Manley, "Software engineering provisioning process," in *Proc. 8th Int. Conf. Software Engineering*, Aug. 1985, pp. 273–284.
[82] E. W. Martin, "Strategy for a DoD software initiative," *Computer*, vol. 16, pp. 52–59, Mar. 1983.
[83] Y. Matsumoto, "Management of industrial software production," *Computer*, vol. 17, pp. 59–70, Feb. 1984.
[84] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308–320, Dec. 1976.
[85] J. A. McCall, P. K. Richards, and G. F. Walters, "Factors in software quality," General Electric, Co., Rep. GE-TIS-77 CIS 02, 1977.
[86] D. D. McCracken and M. A. Jackson, "Life cycle concept considered harmful," *ACM Software Eng. Notes*, pp. 29–32, Apr. 1982.
[87] R. W. McGuffin, A. E. Elliston, B. R. Tranter, and P. N. Westmacott, "CADES—Software engineering in practices," in *Proc. 4th Int. Conf. Software Engineering*, Sept. 1979, pp. 136–144.
[88] P. J. Metzger, *Managing a Programming Project*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[89] J. Munson, "Report of the USAF Scientific Advisory Board committee on the high cost and risk of mission-critical software," Dec. 1983.
[90] Naval Ocean Systems Center, "SEATECS: Software engineering automation for tactical embedded computer systems," Aug. 31, 1982.
[91] E. A. Nelson, *Management Handbook for the Estimation of Computer Programming Costs*, Systems Development Corp., Ad-A648750, Oct. 31, 1966.
[92] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 128–137, Mar. 1979.
[93] D. L. Parnas, P. C. Clements, and D. M. Weiss, "The modular structure of complex systems," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 259–266, Mar. 1985.
[94] M. E. Porter, *Competitive Strategy: Techniques for Analyzing Industries and Competitors*. New York: Free Press, 1980.
[95] —, *Competitive Advantage*. New York: Free Press, 1985.
[96] L. H. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 345–361, July 1978.
[97] Raytheon Computer Services, "Reusable software: Theory and implementation," Raytheon Co., 1983.
[98] D. J. Reifer, *Tutorial: Software Management*. Washington, DC: IEEE Computer Society, 1981.
[99] D. J. Reifer and S. Trattner, "A glossary of software tools and techniques," *Computer*, vol. 10, pp. 52–60, July 1977.
[100] S. P. Reiss, "PECAN: Program development systems that support multiple views," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 276–285, Mar. 1985.
[101] C. Rich and H. E. Shrobe, "Initial report on a programmer's apprentice," *IEEE Trans. Software Eng.*, pp. 456–467, Nov. 1978.
[102] R. J. Rubey, J. A. Dana, and P. W. Biche, "Quantitative aspects of software validation," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 150–155, June 1975.
[103] H. A. Rubin, "A comparison of cost estimation tools," in *Proc. 8th Int. Conf. Software Eng.*, Aug. 1985, pp. 174–180.
[104] B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*. Cambridge, MA: Winthrop, 1980.
[105] D. R. Smith, G. B. Kotik, and S. J. Westfold, "Research on knowledge-based software environments at Kestrel Institute," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1278–1295, November 1985.
[106] H. M. Sneed and A. Marey, "Automated software quality assurance," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 909–916, Sept. 1985.
[107] D. Spadaro, "Project evaluation made simple," *Datamation*, pp. 121–124, Nov. 1985.
[108] H. G. Steubing, "A software engineering environment (SEE) for weapon system software," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 384–397, July 1984.
[109] G. E. Swinson, "Workstation-based rapid simulation aids for distributed processing networks," in *Proc. IEEE Simulation Conf.*, 1984.
[110] STARS Joint Program Office, "STARS—SEE operational concept document," Oct. 2, 1985.
[111] D. Teichroew and E. A. Hershey III, "PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 41–48, Jan. 1977.
[112] W. Teitelman, "A tour through Cedar," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 285–302, Mar. 1985.
[113] W. Teitelman and L. Masinter, "The Interlisp programming environment," *Computer*, vol. 14, pp. 25–33, Apr. 1981.
[114] A. J. Thadhani, "Factors affecting programmer productivity during application development," *IBM Syst. J.*, vol. 23, pp. 19–35, Nov. 1984.
[115] 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.
[116] S. P. Wartik and M. H. Penedo, "Fillin: A reusable tool for form-oriented software," *IEEE Software*, vol. 3, pp. 61–69, Mar. 1986.
[117] A. I. Wasserman, *Tutorial: Software Development Environments*. Washington, DC: Computer Society, 1981.
[118] R. G. Waters, "The Programmer's Apprentice: A session with KBEmacs," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1296–1320, Nov. 1985.
[119] P. Wegner, "Capital-intensive software technology," *IEEE Software*, vol. 1, pp. 7–45, July 1984.
[120] G. M. Weinberg, *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.

[121] G. M. Weinberg and E. L. Schulman, "Goals and performance in computer programming," *Human Factors*, vol. 16, no. 1, pp. 70–77, 1974.
[122] R. D. Williams, "Managing the development of reliable software," in *Proc. 1975 Int. Conf. Reliable Software*, IEEE/ACM, Apr. 1975, pp. 3–8.
[123] A. O. Williman and C. O'Donnell, "Through the central 'multiprocessor' avionics enters the computer era," *Astronautics and Aeronautics*, July 1970.
[124] R. W. Wolverton, "The cost of developing large-scale software," *IEEE Trans. Comput.*, vol. C-24, pp. 615–636, June 1975.
[125] P. Zave, "The operational versus the conventional approach to software development," *Commun. ACM*, pp. 104–118, Feb. 1984.
[126] M. Zelkowitz and S. Squires, Ed., *Proc. ACM Rapid Prototyping Symp.*, ACM, Oct. 1982.

Barry W. Boehm (SM'84) received the B.A. degree in mathematics from Harvard University, Cambridge, MA, in 1957 and the M.A. and Ph.D. degrees in mathematics from the University of California, Los Angeles, in 1961 and 1964, respectively.

He is the Chief Scientist for the Redondo Beach, California-based TRW Defense Systems Group. He is responsible for the Group's Ada office as well as its technology education program and the Quantum Leap program in software development. He is also an Adjunct Professor of Computer Science at the University of California, Los Angeles.

Philip N. Papaccio received the B.S.E.E. degree from the U.S. Naval Academy and the M.B.A. degree from the University of Southern California, Los Angeles. He is also a graduate of the UCLA Executive Management Program.

He is Vice President and General Manager of the System Development Division for the TRW Defense Systems Group of the Electronics and Defense Sector. The Division, located in Manhattan Beach, CA, is responsible for software system integration and sensor data processing for major Department of Defense programs. Previously, he was Assistant General Manager of the Software and Information Systems Division of the TRW Defense Systems Group and was responsible for the development and operation of computer-based systems. Prior to joining TRW in 1977, he was a member of the United States Air Force where he achieved the rank of Colonel. He was associated with research and development for the U.S. space program.