Software Engineering

Barry W. Boehm

Manuscript received June 24, 1976; revised August 16, 1976. The author is with the TRW Systems and Energy Group, Redondo Beach, CA 90278.

***Abstract*—This paper provides a definition of the term "software engineering" and a survey of the current state of the art and likely future trends in the field. The survey covers the technology available in the various phases of the software life cycle—requirements engineering, design, coding, test, and maintenance—and in the overall area of software management and integrated technology-management approaches. It is oriented primarily toward discussing the domain of applicability of techniques (where and when they work), rather than how they work in detail. To cover the latter, an extensive set of 104 references is provided.**

***Index Terms*—Computer software, data systems, information systems, research and development, software development, software engineering, software management.**

I. INTRODUCTION

The annual cost of software in the U.S. is approximately 20 billion dollars. Its rate of growth is considerably greater than that of the economy in general. Compared to the cost of computer hardware, the cost of software is continuing to escalate along the lines predicted in Fig. 1 [1].[1] A recent SHARE study [2] indicates further that software demand over the years 1975–1985 will grow considerably faster (about 21–23 percent per year) than the growth rate in software supply at current estimated growth rates of the software labor force and its productivity per individual, which produce a combined growth rate of about 11.5-17 percent per year over the years 1975–1985.

[1] Another trend has been added to Fig. 1: the growth of software maintenance, which will be discussed later.

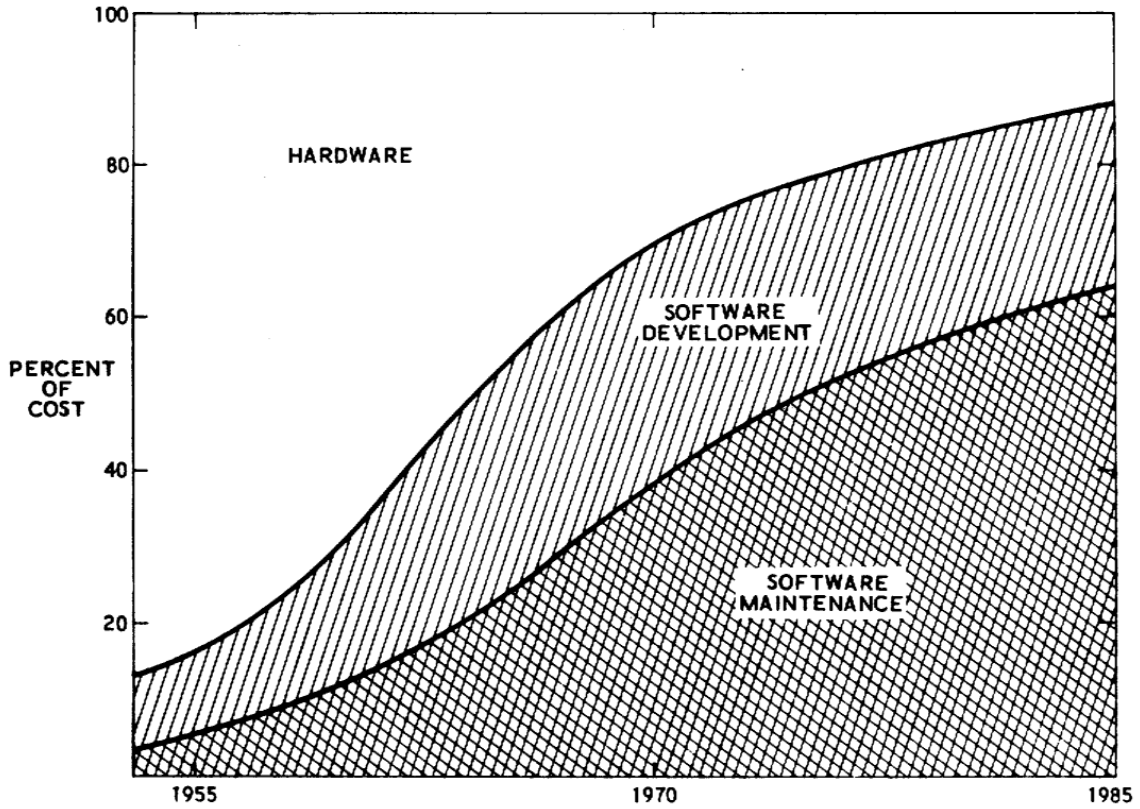


Fig. 1. Hardware–software cost trends.

In addition, as we continue to automate many of the processes which control our life-style—our medical equipment, air traffic control, defense system, personal records, bank accounts—we continue to trust more and more in the reliable functioning of this proliferating mass of software. *Software engineering* is the means by which we attempt to produce all of this software in away that is both cost-effective and reliable enough to deserve our trust. Clearly, it is a discipline which is important to establish well and to perform well.

This paper will begin with a definition of "software engineering." It will then survey the current state of the art of the discipline, and conclude with an assessment of likely future trends.

II. DEFINITIONS

Let us begin by defining "software engineering." We will define software to include not only computer programs, but also the associated documentation required to develop, operate, and maintain the programs. By defining software in this broader sense, we wish to emphasize the necessity of considering the generation of timely documentation as an integral portion of the software development process. We can then combine this with a definition of "engineering" to produce the following definition.

Software Engineering: The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them.

Three main points should be made about this definition. The first concerns the necessity of considering a broad enough interpretation of the word “design” to cover the extremely important activity of software requirements engineering. The second point is that the definition should cover the entire software life cycle, thus including those activities of redesign and modification often termed “software maintenance.” (Fig. 2 indicates the overall set of activities thus encompassed in the definition.) The final point is that our store of knowledge about software which can really be called “scientific knowledge” is a rather small base upon which to build an engineering discipline. But, of course, that is what makes software engineering such a fascinating challenge at this time.

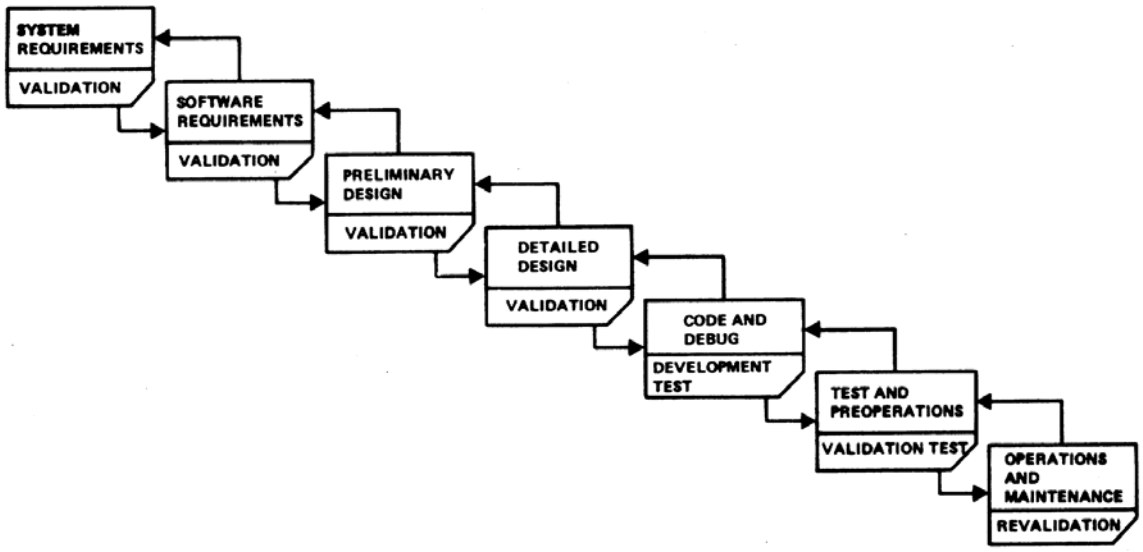


Fig. 2. Software life cycle.

The remainder of this paper will discuss the state of the art of software engineering along the lines of the software life cycle depicted in Fig. 2. Section III contains a discussion of software requirements engineering, with some mention of the problem of determining overall system requirements. Section IV discusses both preliminary design and detailed design technology trends. Section V contains only a brief discussion of programming, as this topic is also covered in a companion article in this issue [3]. Section VI covers both software testing and the overall life cycle concern with software reliability. Section VII discusses the highly important but largely neglected area of software maintenance. Section VIII surveys software management concepts and techniques, and discusses the status and trends of integrated technology-management approaches to software development. Finally, Section IX concludes with an assessment of the current state of the art of software engineering with respect to the definition above.

Each section (sometimes after an introduction) contains a short summary of current practice in the area, followed by a survey of current frontier technology, and concluding with a short summary of likely trends in the area. The survey is oriented primarily toward discussing the domain of applicability of techniques (where and when they work) rather than how they work in detail. An extensive set of references is provided for readers wishing to pursue the latter.

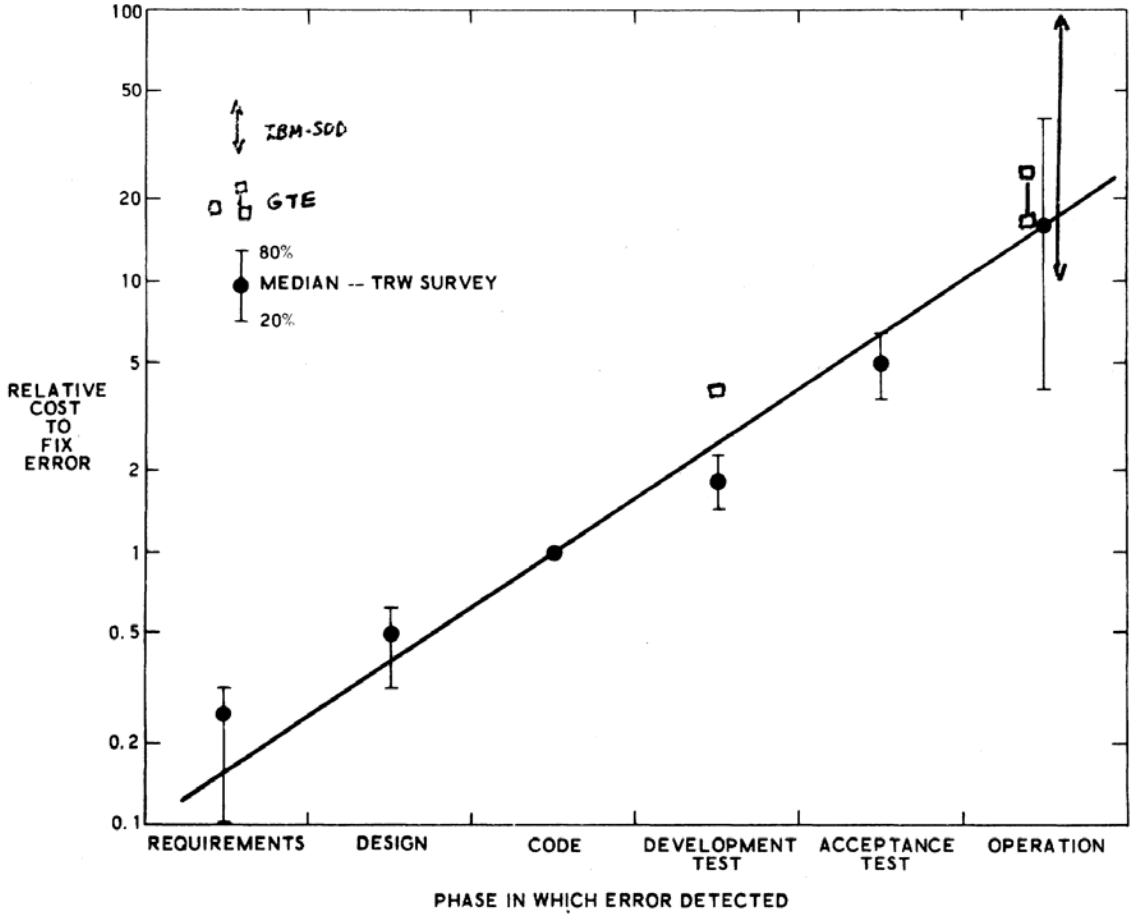
III. SOFTWARE REQUIREMENTS ENGINEERING

A. Critical Nature of Software Requirements Engineering

Software requirements engineering is the discipline for developing a complete, consistent, unambiguous specification—which can serve as a basis for common agreement among all parties concerned—describing what the software product will do (but *not how* it will do it; this is to be done in the design specification).

The extreme importance of such a specification is only now becoming generally recognized. Its importance derives from two main characteristics: 1) it is easy to delay or avoid doing thoroughly; and 2) deficiencies in it are very difficult and expensive to correct later.

Fig. 3 shows a summary of current experience at IBM [4], GTE [5], and TRW on the relative cost of correcting software errors as a function of the phase in which they are corrected. Clearly, it pays off to invest effort in finding requirements errors early and correcting them in, say, 1 man-hour rather than waiting to find the error during operations and having to spend 100 man-hours correcting it.



Besides the cost-to-fix problems, there are other critical problems stemming from a lack of a good requirements specification. These include [6]: 1) top-down designing is impossible, for lack of a well-specified “top”; 2) testing is impossible, because there is nothing to test against; 3) the user is frozen out, because there is no clear statement of what is being produced for him; and 4) management is not in control, as there is no clear statement of what the project team is producing.

B. Current Practice

Currently, software requirements specifications (when they exist at all) are generally expressed in free-form English. They abound with ambiguous terms ("suitable," "sufficient," "real-time," "flexible") or precise-sounding terms with unspecified definitions ("optimum," "99.9 percent reliable") which are potential seeds of dissension or lawsuits once the software is produced. They have numerous errors; one recent study [7] indicated that the first independent review of a fairly good software requirements specification will find from one to four nontrivial errors per page.

The techniques used for determining software requirements are generally an ad hoc manual blend of systems analysis principles [8] and common sense. (These are the good ones; the poor ones are based on ad hoc manual blends of politics, preconceptions, and pure salesmanship.) Some formalized manual techniques have been used successfully for determining business system requirements, such as accurately defined systems (ADS), and time automated grid (TAG). The book edited by Couger and Knapp [9] has an excellent summary of such techniques.

C. Current Frontier Technology: Specification Languages and Systems

1) *ISDOS*: The pioneer system for machine-analyzable software requirements is the ISDOS system developed by Teichroew and his group at the University of Michigan [10]. It was primarily developed for business system applications, but much of the system and its concepts are applicable to other areas. It is the only system to have passed a market and operations test; several commercial, aerospace, and government organizations have paid for it and are successfully using it. The U.S. Air Force is currently using and sponsoring extensions to ISDOS under the Computer Aided Requirements Analysis (CARA) program.

ISDOS basically consists of a problem statement language (PSL) and a problem statement analyzer (PSA). PSL allows the analyst to specify his system in terms of formalized entities (INPUTS, OUTPUTS, REAL WORLD ENTITIES), classes (SETS, GROUPS), relationships (USES, UPDATES, GENERATES), and other information on timing, data volume, synonyms, attributes, etc. PSA operates on the PSL statements to produce a number of useful summaries, such as: formated problem statements; directories and keyword indices; hierarchical structure reports; graphical summaries of flows and relationships; and statistical summaries. Some of these capabilities are actually more suited to supporting system design activities; this is often the mode in which ISDOS is used.

Many of the current limitations of ISDOS stem from its primary orientation toward business systems. It is currently difficult to express real-time performance requirements and man-machine interaction requirements, for example. Other capabilities are currently missing, such as support for configuration control, traceability to design and code, detailed consistency checking, and automatic simulation generation. Other limitations reflect deliberate, sensible design choices: the output graphics are crude, but they are produced in standard 8½ x 11 in size on any standard line printer. Much of the

current work on ISDOS/CARA is oriented toward remedying such limitations, and extending the system to further support software design.

2) *SREP*: The most extensive and powerful system for software requirements specification in evidence today is that being developed under the Software Requirements Engineering Program (SREP) by TRW for the U.S. Army Ballistic Missile Defense Advanced Technology Center (BMDATC) [11]–[13]. Portions of this effort are derivative of ISDOS; it uses the ISDOS data management system, and is primarily organized into a language, the requirements statement language (RSL), and an analyzer, the requirements evaluation and validation system (REVS).

SREP contains a number of extensions and innovations which are needed for requirements engineering in real-time software development projects. In order to represent real-time performance requirements, the individual functional requirements can be joined into stimulus-response networks called R-Nets. In order to focus early attention on software testing and reliability, there are capabilities for designating "validation points" within the R-Nets. For early requirements validation, there are capabilities for automatic generation of functional simulators from the requirements statements. And, for adaptation to changing requirements, there are capabilities for configuration control, traceability to design, and extensive report generation and consistency checking.

Current SREP limitations again mostly reflect deliberate design decisions centered around the autonomous, highly real-time process-control problem of ballistic missile defense. Capabilities to represent large file processing and man-machine interactions are missing. Portability is a problem: although some parts run on several machines, other parts of the system run only on a TI-ASC computer with a very powerful but expensive multicolor interactive graphics terminal. However, the system has been designed with the use of compiler generators and extensibility features which should allow these limitations to be remedied.

3) *Automatic Programming and Other Approaches*: Under the sponsorship of the Defense Advanced Research Projects Agency (DARPA), several researchers are attempting to develop "automatic programming" systems to replace the functions of currently performed by programmers. If successful, could they drive software costs down to zero? Clearly not, because there would still be the need to determine what software the system should produce, i.e., the software requirements. Thus, the methods, or at least the forms, of capturing software requirements are of central concern in automatic programming research.

Two main directions are being taken in this research. One, exemplified by the work of Balzer at USC-ISI [14], is to work within a general problem context, relying on only general rules of information processing (items must be defined or received before they are used, an "if" should have both a "then" and an "else," etc.) to resolve ambiguities, deficiencies, or inconsistencies in the problem statement. This approach encounters formidable problems in natural language processing and may require further restrictions to make it tractable.

The other direction, exemplified by the work of Martin at MIT [15], is to work within a particular problem area, such as inventory control, where there is enough of a general model of software requirements and acceptable terminology to make the problems of resolving ambiguities, deficiencies, and inconsistencies reasonably tractable.

This second approach has, of course, been used in the past in various forms of "programming-by-questionnaire" and application generators [1], [2]. Perhaps the most widely used are the parameterized application generators developed for use on the IBM System/3. IBM has some more ambitious efforts on requirements specification underway, notably one called the Application Software Engineering Tool [16] and one called the Information Automat [17], but further information is needed to assess their current status and directions.

Another avenue involves the formalization and specification of required properties in a software specification (reliability, maintainability, portability, etc.). Some success has been experienced here for small-to-medium systems, using a "Requirements-Properties Matrix" to help analysts infer additional requirements implied by such considerations [18].

D. Trends

In the area of requirements statement languages, we will see further efforts either to extend the ISDOS-PSL and SREP-RSL capabilities to handle further areas of application, such as man-machine interactions, or to develop language variants specific to such areas. It is still an open question as to how general such a language can be and still retain its utility. Other open questions are those of the nature, "which representation scheme is best for describing requirements in a certain area?" BMDATC is sponsoring some work here in representing general data-processing system requirements for the BMD problem, involving Petri nets, state transition diagrams, and predicate calculus [11], but its outcome is still uncertain.

A good deal more can and will be done to extend the capability of requirements statement analyzers. Some extensions are fairly straightforward consistency checking; others, involving the use of relational operators to deduce derived requirements and the detection (and perhaps generation) of missing requirements are more difficult, tending toward the automatic programming work.

Other advances will involve the use of formal requirements statements to improve subsequent parts of the software life cycle. Examples include requirements-design-code consistency checking (one initial effort is underway), the automatic generation of test cases from requirements statements, and, of course, the advances in automatic programming involving the generation of code from requirements.

Progress will not necessarily be evolutionary, though. There is always a good chance of a breakthrough: some key concept which will simplify and formalize large regions of the problem space. Even then, though, there will always remain difficult regions which will require human insight and sensitivity to come up with an acceptable set of software requirements.

Another trend involves the impact of having formal, machine-analyzable requirements (and design) specifications on our overall inventory of software code. Besides improving software reliability, this will make our software much more portable; users will not be tied so much to a particular machine configuration. It is interesting to speculate on what impact this will have on hardware vendors in the future.

IV. SOFTWARE DESIGN

A. The Requirements/Design Dilemma

Ideally, one would like to have a complete, consistent, validated, unambiguous, machine-independent specification of software requirements before proceeding to software design. However, the requirements are not really validated until it is determined that the resulting system can be built for a reasonable *cost*—and to do so requires developing one or more software designs (and any associated hardware *designs* needed).

This dilemma is complicated by the huge number of degrees of freedom available to software/hardware system designers. In the 1950's, as indicated by Table I, the designer had only a few alternatives to choose from in selecting a central processing unit (CPU), a set of peripherals, a programming language, and an ensemble of support software. In the 1970's, with rapidly evolving mini- and microcomputers, firmware, modems, smart terminals, data management systems, etc., the designer has an enormous number of alternative design components to sort out (possibilities) and to seriously choose from (likely choices). By the 1980's, the number of possible design combinations will be formidable.

TABLE I
Design Degrees of Freedom for New Data Processing Systems (Rough Estimates)

Element	Choices (1950's)	Possibilities (1970's)	Likely Choices (1970's)
CPU	5	200	100
Op-Codes	fixed	variable	variable
Peripherals (per function)	1	200	100
Programming language	1	50	5–10
Operating system	0–1	10	5
Data management system	0	100	30

The following are some of the implications for the designer. 1) It is easier for him to do an outstanding design job. 2) It is easier for him to do a terrible design job. 3) He needs more powerful analysis tools to help him sort out the alternatives. 4) He has more opportunities for designing-to-cost. 5) He has more opportunities to design and develop tunable systems. 6) He needs a more flexible requirements-tracking and hardware procurement mechanism to support the above flexibility (particularly in government systems). 7) Any rational standardization (e.g., in programming languages) will be a big help to him, in that it reduces the number of alternatives he must consider.

B. Current Practice

Software design is still almost completely a manual process; There is relatively little effort devoted to design validation and risk analysis before committing to a particular software design. Most software errors are made during the design phase. As seen in Fig. 4, which summarizes several software error analyses by IBM [4], [19] and TRW [20], [21], the ratio of design to coding errors generally exceeds 60:40. (For the TRW data, an error was called a design error if and only if the resulting fix required a change in the detailed design specification.)

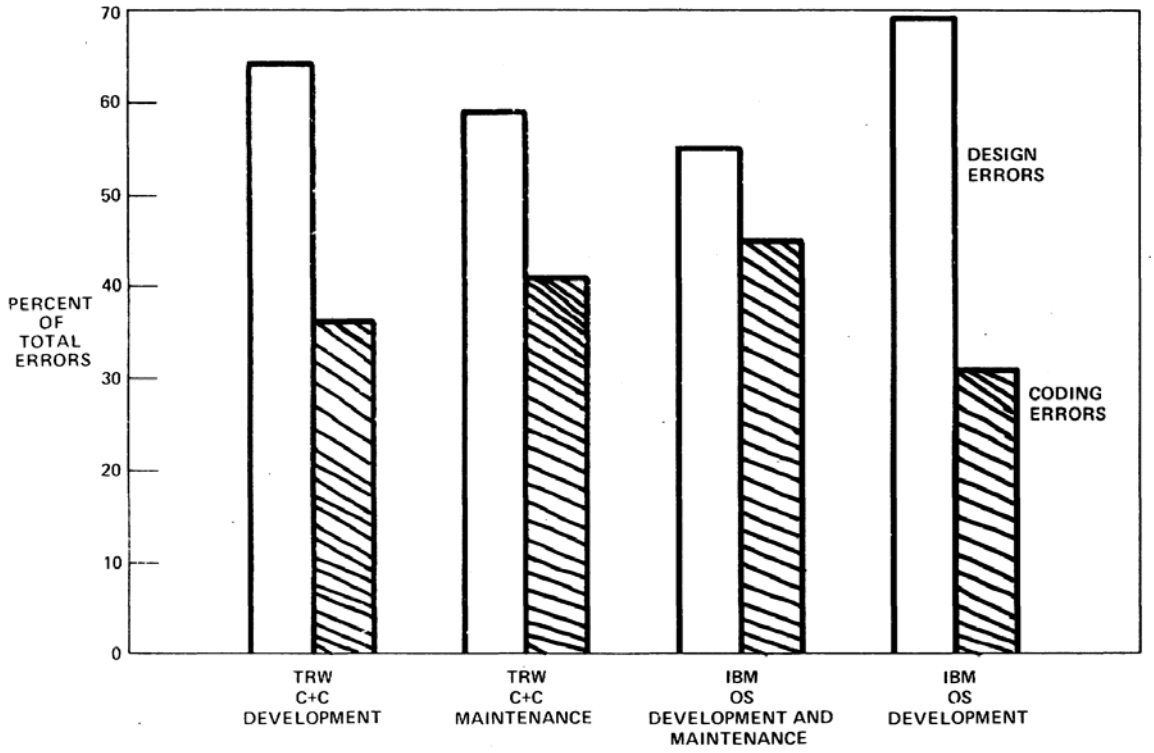


Fig. 4. Most errors in large software systems are in early stages.

Most software design is still done bottom-up, by developing software components before addressing interface and integration issues. There is, however, increasing successful use of top-down design. There is little organized knowledge of what a software designer does, how he does it, or of what makes a good software designer, although some initial work along these lines has been done by Freeman [22].

C. Current Frontier Technology

Relatively little is available to help the designer make the overall hardware-software tradeoff analyses and decisions to appropriately narrow the large number of design degrees of freedom available to him. At the micro level, some formalisms such as LOGOS [23] have been helpful, but at the macro level, not much is available beyond general system engineering techniques. Some help is provided via improved techniques for simulating information systems, such as the Extendable Computer System Simulator (ECSS) [24], [25], which make it possible to develop a fairly thorough functional

simulation of the system for design analysis in a considerably shorter time than it takes to develop the complete design itself.

1) *Top-Down Design*: Most of the helpful new techniques for software design fall into the category of "top-down" approaches, where the "top" is already assumed to be a firm, fixed requirements specification and hardware architecture. Often, it is also assumed that the data structure has also been established. (These assumptions must in many cases be considered potential pitfalls in using such top-down techniques.)

What the top-down approach does well, though, is to provide a procedure for organizing and developing the control structure of a program in a way which focuses early attention on the critical issues of integration and interface definition. It begins with a top-level expression of a hierarchical control structure (often a top level "executive" routine controlling an "input," a "process," and an "out put" routine) and proceeds to iteratively refine each successive lower-level component until the entire system is specified. The successive refinements, which may be considered as "levels of abstraction" or "virtual machines" [26], provide a number of advantages in improved understanding, communication, and verification of complex designs [27], [28]. In general, though, experience shows that some degree of early attention to bottom-level design issues is necessary on most projects [29].

The technology of top-down design has centered on two main issues. One involves establishing guidelines for *how to perform* successive refinements and to group functions into modules; the other involves techniques of *representing* the design of the control structure and its interaction with data.

2) *Modularization*: The techniques of structured design [30] (or composite design [31]) and the modularization guidelines of Parnas [32] provide the most detailed thinking and help in the area of module definition and refinement. Structured design establishes a number of successively stronger types of binding of functions into modules (coincidental, logical, classical, procedural, communicational, informational, and functional) and provides the guideline that a function should be grouped with those functions to which its binding is the strongest. Some designers are able to use this approach quite successfully; others find it useful for reviewing designs but not for formulating them; and others simply find it too ambiguous or complex to be of help. Further experience will be needed to determine how much of this is simply a learning curve effect. In general, Parnas' modularization criteria and guidelines are more straightforward and widely used than the levels-of-binding guidelines, although they may also be becoming more complicated as they address such issues as distribution of responsibility for erroneous inputs [33]. Along these lines, Draper Labs' Higher Order Software (HOS) methodology [34] has attempted to resolve such issues via a set of six axioms covering relations between modules and data, including responsibility for erroneous inputs. For example, Axiom 5 states, "Each module controls the rejection of invalid elements of its own, and only its own, input set."[2]

3) *Design Representation*: Flow charts remain the main method currently used for design representation. They have a number of deficiencies, particularly in representing hierarchical control structures and data interactions. Also, their free-form nature makes it too easy to construct complicated, unstructured designs which are hard to understand and

[2] Problems can arise, however, when one furnishes such a *design choice* with the power of an *axiom*. Suppose, for example, the input set contains a huge table or a master file. Is the module stuck with the job of checking it, by itself, every time?

maintain. A number of representation schemes have been developed to avoid these deficiencies.

The hierarchical input-process-output (HIPO) technique [35] represents software in a hierarchy of modules, each of which is represented by its inputs, its outputs, and a summary of the processing which connects the inputs and outputs. Advantages of the HIPO technique are its ease of use, ease of learning, easy-to-understand graphics, and disciplined structure. Some general disadvantages are the ambiguity of the control relationships (are successive lower level modules in sequence, in a loop, or in an if/else relationship?), the lack of summary information about data, the unwieldiness of the graphics on large systems, and the manual nature of the technique. Some attempts have been made to automate the representation and generation of HIPO's such as Univac's PROVAC System [36].

The structure charts used in structured design [30], [31] remedy some of these disadvantages, although they lose the advantage of representing the processes connecting the inputs with the outputs. In doing so, though, they provide a more compact summary of a module's inputs and outputs which is less unwieldy on large problems. They also provide some extra symbology to remove at least some of the sequence/loop/branch ambiguity of the control relationships.

Several other similar conventions have been developed [37]–[39], each with different strong points, but one main difficulty of any such manual system is the difficulty of keeping the design consistent and up-to-date, especially on large problems. Thus, a number of systems have been developed which store design information in machine-readable form. This simplifies updating (and reduces update errors) and facilitates generation of selective design summaries and simple consistency checking. Experience has shown that even a simple set of automated consistency checks can catch dozens of potential problems in a large design specification [21]. Systems of this nature that have been reported include the Newcastle TOPD system [40], TRW's DACC and DEVISE systems [21], Boeing's DECA System [41], and Univac's PROVAC [361; several more are under development.

Another machine-processable design representation is provided by Caine, Farber, and Gordon's Program Design Language (PDL) System [42]. This system accepts constructs which have the form of hierarchical structured programs, but instead of the actual code, the designer can write some English text describing what the segment of code will do. (This representation was originally called "structured pidgin" by Mills [43].) The PDL system again makes updating much easier; it also provides a number of useful formatted summaries of the design information, although it still lacks some wished-for features to support terminology control and version control. The program-like representation makes it easy for programmers to read and write PDL, albeit less easy for nonprogrammers. Initial results in using the PDL system on projects have been quite favorable.

D. Trends

Once a good deal of design information is in machine-readable form, there is a fair amount of pressure from users to do more with it: to generate core and time budgets, software cost estimates, first-cut data base descriptions, etc. We should continue to see

such added capabilities, and generally a further evolution toward computer-aided-design systems for software. Besides improvements in determining and representing control structures, we should see progress in the more difficult area of data structuring. Some initial attempts have been made by Hoare [44] and others to provide a data analog of the basic control structures in structured programming, but with less practical impact to date. Additionally, there will be more integration and traceability between the requirements specification, the design specification, and the code—again with significant implications regarding the improved portability of a user's software.

The proliferation of minicomputers and microcomputers will continue to complicate the designer's job. It is difficult enough to derive or use principles for partitioning software jobs on single machines; additional degrees of freedom and concurrency problems just make things so much harder. Here again, though, we should expect at least some initial guidelines for decomposing information processing jobs into separate concurrent processes.

It is still not clear, however, how much one can formalize the software design process. Surveys of software designers have indicated a wide variation in their design styles and approaches, and in their receptiveness to using formal design procedures. The key to good software design still lies in getting the best out of good people, and in structuring the job so that the less-good people can still make a positive contribution.

V. PROGRAMMING

This section will be brief, because much of the material will be covered in the companion article by Wegner on "Computer Languages" [3].

A. Current Practice

Many organizations are moving toward using structured code [28], [43] (hierarchical, block-oriented code with a limited number of control structures—generally SEQUENCE, IFTHENELSE, CASE, DOWHILE, and DOUNTIL—and rules for formatting and limiting module size). A great deal of terribly unstructured code is still being written, though, often in assembly language and particularly for the rapidly proliferating minicomputers and microcomputers.

B. Current Frontier Technology

Languages are becoming available which support structured code and additional valuable features such as data typing and type checking (e.g., Pascal [45]). Extensions such as concurrent Pascal [46] have been developed to support the programming of concurrent processes. Extensions to data typing involving more explicit binding of procedures and their data have been embodied in recent languages such as ALPHARD [47] and CLU [48]. Metacompiler and compiler writing system technology continues to improve, although much more slowly in the code generation area than in the syntax analysis area.

Automated aids include support systems for top-down structured programming such as the Program Support Library [49], Process Construction [50], TOPD [40], and

COLUMBUS [51]. Another novel aid is the Code Auditor program [50] for automated standards compliance checking-which guarantees that the standards are more than just words. Good programming practices are now becoming codified into style handbooks, i.e., Kernighan and Plauger [52] and Ledgard [53].

C. Trends

It is difficult to clean up old programming languages or to introduce new ones into widespread practice. Perhaps the strongest hope in this direction is the current Department of Defense (DoD) effort to define requirements for its future higher order programming languages [54], which may eventually lead to the development and widespread use of a cleaner programming language. Another trend will be an increasing capability for automatically generating code from design specifications.

VI. SOFTWARE TESTING AND RELIABILITY

A. Current Practice

Surprisingly often, software testing and reliability activities are still not considered until the code has been run the first time and found not to work. In general, the high cost of testing (still 40-50 percent of the development effort) is due to the high cost of reworking the code at this stage (see Fig. 3), and to the wasted effort resulting from the lack of an advance test plan to efficiently guide testing activities.

In addition, most testing is still a tedious manual process which is error-prone in itself. There are few effective criteria used for answering the question, “How much testing is enough?” except the usual “when the budget (or schedule) runs out.” However, more and more organizations are now using disciplined test planning and some objective criteria such as “exercise every instruction” or “exercise every branch,” often with the aid of automated test monitoring tools and test case planning aids. But other technologies, such as mathematical proof techniques, have barely begun to penetrate the world of production software.

B. Current Frontier Technology

1) *Software Reliability Models and Phenomenology*: Initially, attempts to predict software reliability (the probability of future satisfactory operation of the software) were made by applying models derived from hardware reliability analysis and fitting them to observed software error rates [55]. These models worked at times, but often were unable to explain actual experienced error phenomena. This was primarily because of fundamental differences between software phenomenology and the hardware-oriented assumptions on which the models were based. For example, software components do not degrade due to wear or fatigue; no imperfection or variations are introduced in making additional copies of a piece of software (except possibly for a class of easy-to-check copying errors); repair of a software fault generally results in a different software configuration than previously, unlike most hardware replacement repairs.

Models are now being developed which provide explanations of the previous error histories in terms of appropriate software phenomenology. They are based on a view of a software program as a mapping from a space of inputs into a space of outputs [56], of program operation as the processing of a sequence of points in the input space, distributed according to an operational profile [57], and of testing as a sampling of points from the input space [56] (see Fig. 5). This approach encounters severe problems of scale on large programs, but can be used conceptually as a means of appropriately conditioning time-driven reliability models [58]. Still, we are a long way off from having truly reliable reliability-estimation methods for software.

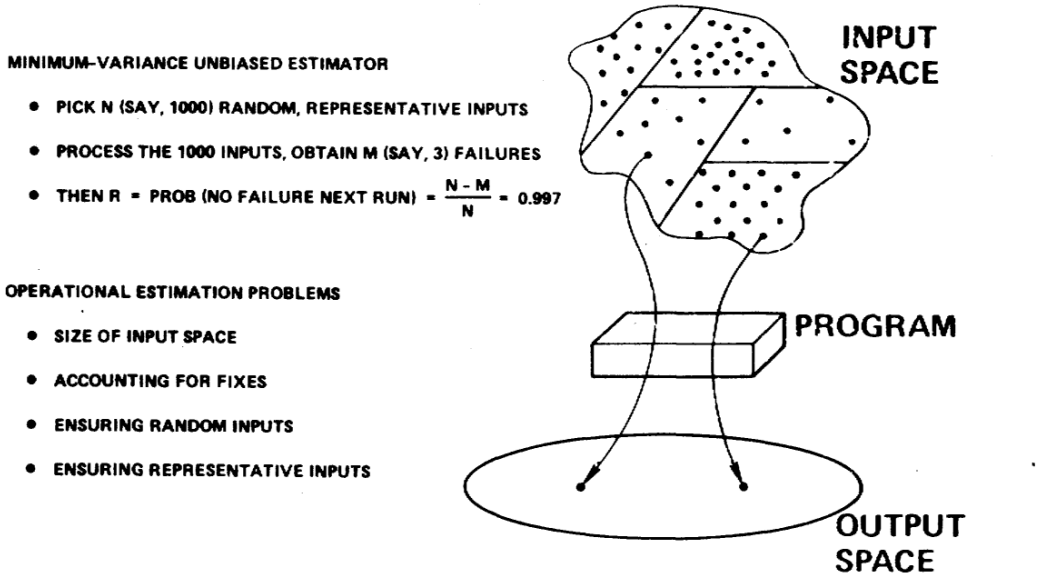


Fig. 5. Input space sampling provides a basis for software reliability measurement.

2) *Software Error Data*: Additional insights into reliability estimation have come from analyzing the increasing data base of software errors. For example, the fact that the distributions of serious software errors are dissimilar from the distributions of minor errors [59] means that we need to define "errors" very carefully when using reliability prediction models. Further, another study [60] found that the rates of fixing serious errors and of fixing minor errors vary with management direction. ("Close out all problems quickly" generally gets minor simple errors fixed very quickly, as compared to "Get the serious problems fixed first.")

Other insights afforded by software data collection include better assessments of the relative efficacy of various software reliability techniques [4], [19], [601, identification of the requirements and design phases as key leverage points for cost savings by eliminating errors earlier (Figs. 2 and 3), and guidelines for organizing test efforts (for example, one recent analysis indicated that over half the errors were experienced when the software was handling data singularities and extreme points [60]). So far, however, the proliferation of definitions of various terms (error, design phase, logic error, validation test), still make it extremely difficult to compare error data from different sources. Some efforts to establish a unified software reliability data base and associated standards, terminology, and data collection procedures are now under way at

USAF Rome Air Development Center, and within the IEEE Technical Committee on Software Engineering.

3) *Automated Aids*: Let us sketch the main steps of testing between the point the code has been written and the point it is pronounced acceptable for use, and describe for each stop the main types of automated aids which have been found helpful. More detailed discussion of these aids can be found in the surveys by Reifer [61] and Ramamoorthy and Ho [62] which in turn have references to individual contributions to the field.

a) *Static code analysis*: Automated aids here include the usual compiler diagnostics, plus extensions involving more detailed data-type checking. Code auditors check for standards compliance, and can also perform various type-checking functions. Control flow and reachability analysis is done by structural analysis programs (flow charters have been used for some of the elementary checks here, "structurizers" can also be helpful). Other useful static analysis tools perform set-use analysis of data elements, singularity analysis, units consistency analysis, data base consistency checking, and data-versus-code consistency checking.

b) *Test case preparation*: Extensions to structural analysis programs provide assistance in choosing data values which will make the program execute along a desired path.. Attempts have been made to automate the generation of such data values; they can generally succeed for simple cases, but run into difficulty in handling loops or branching on complex calculated values (e.g., the results of numerical integration). Further, these programs only help generate the *inputs*; the tester must still calculate the expected outputs himself.

Another set of tools will automatically insert instrumentation to verify that a desired path has indeed been exercised in the test. A limited capability exists for automatically determining the minimum number of test cases required to exercise all the code. But, as yet, there is no tool which helps to determine the most appropriate sequence in which to run a series of tests.

c) *Test monitoring and output checking*: Capabilities have been developed and used for various kinds of dynamic data-type checking and assertion checking, and for timing and performance analysis. Test output post-processing aids include output comparators and exception report capabilities, and test-oriented data reduction and report generation packages.

d) *Fault isolation, debugging*: Besides the traditional tools-the core dump, the trace, the snapshot, and the breakpoint-several capabilities have been developed for interactive replay or backtracking of the program's execution. This is still a difficult area, and only a relatively few advanced concepts have proved generally useful.

e) *Retesting* (*once a presumed fix has been made*): Test data management systems (for the code, the input data, and the comparison output data) have been shown to be most valuable here, along with comparators to check for the differences in code, inputs, and outputs between the original and the modified program and test case. A promising experimental tool performs a comparative structure analysis of the original and modified code, and indicates which test cases need to be rerun.

f) *Integration of routines into systems*: In general, automated aids for this process are just larger scale versions of the test data management systems above. Some additional capabilities exist for interface consistency checking, e.g., on the length and form of

parameter lists or data base references. Top-down development aids are also helpful in this regard.

g) *Stopping*: Some partial criteria for thoroughness of testing can and have been automatically monitored. Tools exist which keep a cumulative tally of the number or percent of the instructions or branches which have been exercised during the test program, and indicate to the tester what branch conditions must be satisfied in order to completely exercise all the code or branches. Of course, these are far from complete criteria for determining when to stop testing; the completeness question is the subject of the next section.

4) *Test Sufficiency and Program Proving*: If a program's input space and output space are finite (where the input space includes not only all possible incoming inputs, but also all possible values in the program's data base), then one can construct a set of "black box" tests (one for each point in the input space) which can show conclusively that the program is correct (that its behavior matches its specification).

In general, though, a program's input space is infinite;- for example, it must generally provide for rejecting unacceptable inputs. In this case, a finite set of black-box tests is not a sufficient demonstration of the program's correctness (since, for any input x, one must assure that the program does not wrongly treat it as a special case). Thus, the demonstration of correctness in this case involves some formal argument (e.g., a proof using induction) that the dynamic performance of the program indeed produces the static transformation of the input space indicated by the formal specification for the program. For finite portions of the input space, a successful exhaustive test of all cases can be considered as a satisfactory formal argument. Some good initial work in sorting out the conditions under which testing is equivalent to proof of a program's correctness has been done by Goodenough and Gerhart [63] and in a review of their work by Wegner [64].

5) *Symbolic Execution*: An attractive intermediate step between program testing and proving is "symbolic execution," a manual or automated procedure which operates on symbolic inputs (e.g., variable names) to produce symbolic outputs. Separate cases are generated for different execution paths. If there are a finite number of such paths, symbolic execution can be used to demonstrate correctness, using a finite symbolic input space and output space. In general, though, one cannot guarantee a finite number of paths. Even so, symbolic execution can be quite valuable as an aid to either program testing or proving. Two fairly powerful automated systems for symbolic execution exist, the EFFIGY system [65] and the SELECT system [66].

6) *Program Proving* (*Program Verification*): Program proving (increasingly referred to as program verification) involves expressing the program specifications as a logical proposition, expressing individual program execution statements as logical propositions, expressing program branching as an expansion into separate cases, and performing logical transformations on the propositions in a way which ends by demonstrating the equivalence of the program and its specification. Potentially infinite loops can be handled by inductive reasoning.

In general, nontrivial programs are very complicated and time-consuming to prove. In 1973, it was estimated that about one man-month of expert effort was required to prove 100 lines of code [67]. The largest program to be proved correct to date contained about 2000 statements [68]. Again, automation can help out on some of the complications. Some automated verification systems exist, notably those of London *et al*.

[69] and Luckham et al. [70]. In general, such systems do not work on programs in the more common languages such as Fortran or Cobol. They work in languages such as Pascal [45], which has (unlike Fortran or Cobol) an axiomatic definition [71] allowing, clean expression of program statements as logical propositions. An excellent survey of program verification technology has been given by London [72].

Besides size and language limitations, there are other factors which limit the utility of program proving techniques. Computations on "real" variables involving truncation and roundoff errors are virtually impossible to analyze with adequate accuracy for most nontrivial programs. Programs with nonformalizable inputs (e.g., from a sensor where one has just a rough idea of its bias, signal-to-noise ratio, etc.) are impossible to handle. And, of course, programs can be proved to be consistent with a specification which is itself incorrect with respect to the system's proper functioning. Finally, there is no guarantee that the proof is correct or complete; in fact, many published "proofs" have subsequently been demonstrated to have holes in them [63].

It has been said and often repeated that "testing can be used to demonstrate the presence of errors but never their absence" [73]. Unfortunately, if we must define "errors" to include those incurred by the two limitations above (errors in specifications and errors in proofs), it must be admitted that "program proving can be used to demonstrate the presence of errors but never their absence."

7) *Fault-Tolerance*: Programs do not have to be error-free to be reliable. If one could just detect erroneous computations as they occur and compensate for them, one could achieve reliable operation. This is the rationale behind schemes for fault-tolerant software. Unfortunately, both detection and compensation are formidable problems. Some progress has been made in the case of software detection and compensation for hardware errors; see, for example, the articles by Wulf [741 and Goldberg [75]. For software errors, Randell has formulated a concept of separately- programmed, alternate "recovery blocks" [761. It appears attractive for parts of the error compensation activity, but it is still too early to tell how well it will handle the error detection problem, or what the price will be in program slowdown.

C. Trends

As we continue to collect and analyze more and more data on how, when, where, and why people make software errors, we will get added insights on how to avoid making such errors, how to organize our validation strategy and tactics (not only in testing but throughout the software life cycle), how to develop or evaluate new automated aids, and how to develop useful methods for predicting software reliability. Some automated aids, particularly for static code checking, and for some dynamic-type or assertion checking, will be integrated into future programming languages and compilers. We should see some added useful criteria and associated aids for test completeness, particularly along the lines of exercising "all data elements" in some appropriate way. Symbolic execution capabilities will probably make their way into automated aids for test case generation, monitoring, and perhaps retesting.

Continuing work into the theory of software testing should provide some refined concepts of test validity, reliability, and completeness, plus a better theoretical base for supporting hybrid test/proof methods of verifying programs. Program proving techniques

and aids will become more powerful in the size and range of programs they handle, and hopefully easier to use and harder to misuse. But many of their basic limitations will remain, particularly those involving real variables and nonformalizable inputs.

Unfortunately, most of these helpful capabilities will be available only to people working in higher order languages. Much of the progress in test technology will be unavailable to the increasing number of people who find themselves spending more and more time testing assembly language software written for minicomputers and microcomputers with poor test support capabilities. Powerful cross-compiler capabilities on large host machines and microprogrammed diagnostic emulation capabilities [77] should provide these people some relief after a while, but a great deal of software testing will regress back to earlier generation "dark ages."

VII. SOFTWARE MAINTENANCE

A. Scope of Software Maintenance

Software maintenance is an extremely important but highly neglected activity. Its importance is clear from Fig. 1: about 40 percent of the overall hardware-software dollar is going into software maintenance today, and this number is likely to grow to about 60 percent by 1985. It will continue to grow for a long time, as we continue to add to our inventory of code via development at a faster rate than we make code obsolete.

The figures above are only very approximate, because our only data so far are based on highly approximate definitions. It is hard to come up with an unexceptional definition of software maintenance. Here, we define it as "the process of modifying existing operational software while leaving its primary functions intact." It is useful to divide software maintenance into two categories: software *update*, which results in a changed functional specification for the software, and software *repair*, which leaves the functional specification intact. A good discussion of software repair is given in the paper by Swanson [78], who divides it into the subcategories of corrective maintenance (of processing, performance, or implementation failures), adaptive maintenance (to changes in the processing or data environment), and perfective maintenance (for enhancing performance or maintainability).

For either update or repair, three main functions are involved in software maintenance [79].

Understanding the existing software: This implies the need for good documentation, good traceability between requirements and code, and well-structured and well-formatted code.

Modifying the existing software: This implies the need for software, hardware, and data structures which are easy to expand and which minimize side effects of changes, plus easy-to-update documentation.

Revalidating the modified software: This implies the need for software structures which facilitate selective retest, and aids for making retest more thorough and efficient.

Following a short discussion of current practice in software maintenance, these three functions will be used below as a framework for discussing current frontier technology in software maintenance.

B. Current Practice

As indicated in Fig. 6, probably about 70 percent of the overall cost of software is spent in software maintenance. A recent paper by Elshoff [80] indicates that the figure for General Motors is about 75 percent, and that GM is fairly typical of large business software activities. Daly [5] indicates that about 60 percent of GTE's 10-year life cycle costs for real-time software are devoted to maintenance. On two Air Force command and control software systems, the maintenance portions of the 10-year life cycle costs were about 67 and 72 percent. Often, maintenance is not done very efficiently. On one aircraft computer, software development costs were roughly $75/instruction, while maintenance costs ran as high as $4000/instruction [81].

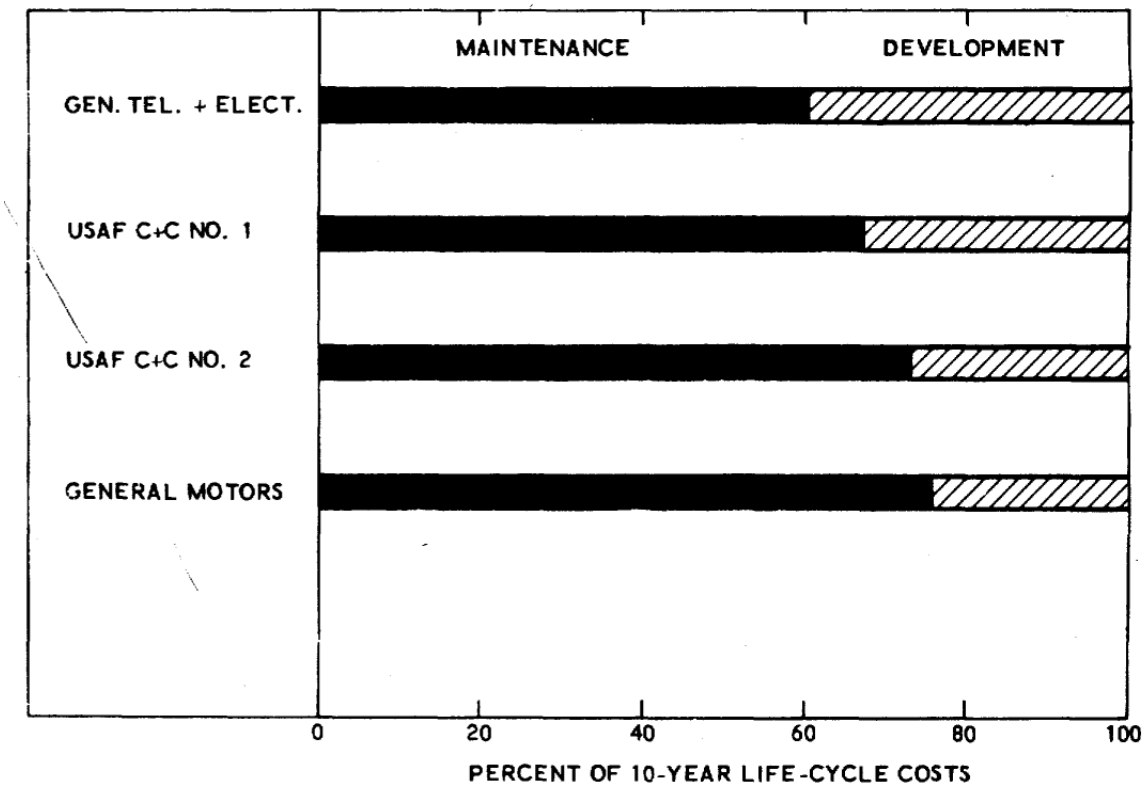


Fig. 6. Software life-cycle cost breakdown.

Despite its size, software maintenance is a highly neglected activity. In general, less-qualified personnel are assigned to maintenance tasks. There are few good general principles and few studies of the process, most of them inconclusive.

Further, data processing practices are usually optimized around other criteria than maintenance efficiency. Optimizing around development cost and schedule criteria generally leads to compromises in documentation, testing, and structuring. Optimizing around hardware efficiency criteria generally leads to use of assembly language and skimping on hardware, both of which correlate strongly with increased software maintenance costs [1].

C. Current Frontier Technology

1) *Understanding the Existing Software*: Aids here have largely been discussed in previous sections: structured programming, automatic formatting, and code auditors for standards compliance checking to enhance code readability; machine-readable requirements and design languages with traceability support to and from the code. Several systems exist for automatically updating documentation by excerpting information from the revised code and comment cards.

2) *Modifying the Existing Software*: Some of Parnas' modularization guidelines [32] and the data abstractions of the CLU [48] and ALPHARD [47] languages make it easier to minimize the side effects of changes. There may be a maintenance price, however. In the past, some systems with highly coupled programs and associated data structures have had difficulties with data base updating. This may not be a problem with today's data dictionary capabilities, but the interactions have not yet been investigated. Other aids to modification are structured code, configuration management techniques, programming support libraries, and process construction systems.

3) *Revalidating the Modified Software*: Aids here were discussed earlier under testing; they include primarily test data management systems, comparator programs, and program structure analyzers with some limited capability for selective retest analysis.

4) *General Aids*: On-line interactive systems help to remove one of the main bottlenecks involved in software maintenance: the long turnaround times for retesting. In addition, many of these systems are providing helpful capabilities for text editing and software module management. They will be discussed in more detail under "Management and Integrated Approaches" below. In general, a good deal more work has been done on the maintainability aspects of data bases and data structures than for program structures; a good survey of data base technology is given in a recent special issue of *ACM Computing Surveys* [82].

D. Trends

The increased concern with life cycle costs, particularly within the U.S. DoD [83], will focus a good deal more attention on software maintenance. More data collection and analysis on the growth dynamics of software systems, such as the Belady-Lehman studies of OS/360 [84], will begin to point out the high-leverage areas for improvement. Explicit mechanisms for confronting maintainability issues early in the development cycle, such as the requirements- properties matrix [18] and the design inspection [4] will be refined and used more extensively. In fact, we may evolve a more general concept of software quality assurance (currently focused largely on reliability concerns), involving such activities as independent reviews of software requirements and design specifications by experts in software maintainability. Such activities will be enhanced considerably with the advent of more powerful capabilities for analyzing machine-readable requirements and design specifications. Finally, advances in automatic programming [14], [15] should reduce or eliminate some maintenance activity, at least in some problem domains.

VIII. SOFTWARE MANAGEMENT AND INTEGRATED APPROACHES

A. Current Practice

There are more opportunities for improving software productivity and quality in the area of management than anywhere else. The difference between software project successes and failures has most often been traced to good or poor practices in software management. The biggest software management problems have generally been the following.

Poor Planning: Generally, this leads to large amounts of wasted effort and idle time because of tasks being unnecessarily performed, overdone, poorly synchronized, or poorly interfaced.

Poor Control: Even a good plan is useless when it is not kept up-to-date and used to manage the project. Poor Resource Estimation: Without a firm idea of how much time and effort a task should take, the manager is in a poor position to exercise control.

Unsuitable Management Personnel: As a very general statement, software personnel tend to respond to problem situations as designers rather than as managers.

Poor Accountability Structure: Projects are generally organized and run with very diffuse delineation of responsibilities, thus exacerbating all the above problems.

Inappropriate Success Criteria: Minimizing development costs and schedules will generally yield a hard-tomaintain product. Emphasizing "percent coded" tends to get people coding early and to neglect such key activities as requirements and design validation, test planning, and draft user documentation.

Procrastination on Key Activities: This is especially prevalent when reinforced by inappropriate success criteria as above.

B. Current Frontier Technology

1) *Management Guidelines*: There is no lack of useful material to guide software management. In general, it takes a book-length treatment to adequately cover the issues. A number of books on the subject are now available [85] -[95], but for various reasons they have not strongly influenced software management practice. Some of the books (e.g., Brooks [85] and the collections by Horowitz [86], Weinwurm [87], and Buxton, Naur, and Randell [88] are collections of very good advice, ideas, and experiences, but are fragmentary and lacking in a consistent, integrated life cycle approach. Some of the books (e.g., Metzger [89], Shaw and Atkins [90], Hice et al. [91], Ridge and Johnson [92], and Gildersleeve [93], are good on checklists and procedures but (except to some extent the latter two) are light on the human aspects of management, such as staffing, motivation, and conflict resolution. Weinberg [94] provides the most help on the human aspects, along with Brooks [85] and Aron [95], but in turn, these three books are light on checklists and procedures. (A second volume by Aron is intended to cover software group and project considerations.) None of the books have an adequate treatment of some items, largely because they are so poorly understood: chief among these items are software cost and resource estimation, and software maintenance.

In the area of software cost estimation, the paper by Wolverton [96] remains the most useful source of help. It is strongly based on the number of object instructions (modified by complexity, type of application, and novelty) as the determinant of software

cost. This is a known weak - spot, but not one for which an acceptable improvement has surfaced. One possible line of improvement might be along the “software physics” lines being investigated by Halstead [97] and others; some interesting initial results have been obtained here, but their utility for practical cost estimation remains to be demonstrated. A good review of the software cost estimation area is contained in [98].

2) *Management-Technology Decoupling*: Another difficulty of the above books is the degree to which they are decoupled from software technology. Except for the Horowitz and Aron books, they say relatively little about the use of such advanced-technology aids as formal, machine- readable requirements, top-down design approaches, structured programming, and automated aids-to software testing.

Unfortunately, the management-technology decoupling works the other way, also. In the design area, for example, most treatments of top-down software design are presented as logical exercises independent of user or economic considerations. Most automated aids to software design provide little support for such management needs as configuration management, traceability to code or requirements, and resource estimation and control. Clearly, there needs to be a closer coupling between technology and management than this. Some current efforts to provide integrated management-technology approaches are presented next.

3) *Integrated Approaches*: Several major integrated systems for software development are currently in operation or under development. In general, their objectives are similar: to achieve a significant boost in software development efficiency and quality through the synergism of a unified approach. Examples are the utility of having a complementary development approach (top-down, hierarchical) and set of programming standards (hierarchical, structured code); the ability to perform a software update and at the same time perform a set of timely, consistent project status updates (new version number of module, closure of software problem report, updated status logs); or simply the improvement in software system integration achieved when all participants are using the same development concept, ground rules, and support software.

The most familiar of the integrated approaches is the IBM “top-down structured programming with chief programmer teams” concept. A good short description of the concept is given by Baker [49]; an extensive treatment is available in a 15-volume series of reports done by IBM for the U.S. Army and Air Force [99]. The top-down structured approach was discussed earlier. The Chief Programmer Team centers around an individual (the Chief) who is responsible for designing, coding, and integrating the top-level control structure as well as the key components of the team’s product; for managing and motivating. the team personnel and personally reading and reviewing all their code; and also for performing traditional management and customer interface functions. The Chief is assisted by a Backup programmer who is prepared at anytime to take the Chiefs place, a Librarian who handles job submission, configuration control, and project status accounting, and additional programmers and specialists as needed.

In general, the overall ensemble of techniques has been quite successful, but the Chief Programmer concept has had mixed results [99]. It is difficult to find individuals with enough energy and talent to perform all the above functions. If you find one, the project will do quite well; otherwise, you have concentrated most of the project risk in a single individual, without a good way of finding out whether or not he is in trouble. The Librarian and Programming Support Library concept have generally been quite useful,

although to date the concept has been oriented toward a batch-processing development environment.

Another "structured" integrated approach has been developed and used at SofTech [38]. It is oriented largely around a hierarchical-decomposition design approach, guided by formalized sets of principles (modularity, abstraction, localization, hiding, uniformity, completeness, confirmability), processes (purpose, concept, mechanism, notation, usage), and goals (modularity, efficiency, reliability, understandability). Thus, it accommodates some economic considerations, although it says little about any other management considerations. It appears to work well for SofTech, but in general has not been widely assimilated elsewhere.

A more management-intensive integrated approach is the TRW software development methodology exemplified in the paper by Williams [50] and the TRW Software Development and Configuration Management Manual [100], which has been used as the basis for several recent government in-house software manuals. This approach features a coordinated set of high-level and detailed management objectives, associated automated aids-standards compliance checkers, test thoroughness checkers, process construction aids, reporting systems for cost, schedule, core and time budgets, problem identification and closure, etc.-and unified documentation and management devices such as the Unit Development Folder. Portions of the approach are still largely manual, although additional automation is underway, e.g., via the Requirements Statement Language [13].

The SDC Software Factory [101] is a highly ambitious attempt to automate and integrate software development technology. It consists of an interface control component, the Factory Access and Control Executive (FACE), which provides users access to various tools and data bases: a project planning and monitoring system, a software development data base and module management system, a top-down development support system, a set of test tools, etc. As the system is still undergoing development and preliminary evaluation, it is too early to tell what degree of success it will have.

Another factory-type approach is the System Design Laboratory (SDL) under development at the Naval Electronics Laboratory Center [102]. It currently consists primarily of a framework within which a wide range of aids to software development can be incorporated. The initial installment contains text editors, compilers, assemblers, and microprogrammed emulators. Later additions are envisioned to include design, development, and test aids, and such management aids as progress reporting, cost reporting, and user profile analysis.

SDL itself is only a part of a more ambitious integrated approach, ARPA's National Software Works (NSW) [102]. The initial objective here has been to develop a "Works Manager" which will allow a software developer at a terminal to access a wide variety of software development tools on various computers available over the ARPANET. Thus, a developer might log into the NSW, obtain his source code from one computer, text-edit it on another, and perhaps continue to hand the program to additional computers for test instrumentation, compiling, executing, and postprocessing of output data. Currently, an initial version of the Works Manager is operational, along with a few tools, but it is too early to assess the likely outcome and payoffs of the project.

C. Trends

In the area of management techniques, we are probably entering a consolidation period, particularly as the U.S. DoD proceeds to implement the upgrades in its standards and procedures called for in the recent DoD Directive 5000.29 [104]. The resulting government-industry efforts should produce a set of software management guidelines which are more consistent and up-to-date with today's technology than the ones currently in use. It is likely that they will also be more comprehensible and less encumbered with DoD jargon; this will make them more useful to the software field in general.

Efforts to develop integrated, semiautomated systems for software development will continue at a healthy clip. They will run into a number of challenges which will probably take a few years to work out. Some are technical, such as the lack of a good technological base for data structuring aids, and the formidable problem of integrating complex software support tools. Some are economic and managerial, such as the problems of pricing services, providing tool warranties, and controlling the evolution of the system. Others are environmental, such as the proliferation of minicomputers and microcomputers, which will strain the capability of any support system to keep up-to-date.

Even if the various integrated systems do not achieve all their goals, there will be a number of major benefits from the effort. One is of course that a larger number of support tools will become available to a larger number of people (another major channel of tools will still continue to expand, though: the independent software products marketplace). More importantly, those systems which achieve a degree of conceptual integration (not just a free-form tool box) will eliminate a great deal of the semantic confusion which currently slows down our group efforts throughout the software life cycle. Where we have learned how to talk to each other about our software problems, we tend to do pretty well.

IX. CONCLUSIONS

Let us now assess the current state of the art of tools and techniques which are being used to solve software development problems, in terms of our original definition of software engineering: the practical application of *scientific knowledge* in the design and construction of software. Table II presents a summary assessment of the extent to which current software engineering techniques are based on solid scientific principles (versus empirical heuristics). The summary assessment covers four dimensions: the extent to which existing scientific principles apply across the entire software life cycle, across the entire range of software applications, across the range of engineering-economic analyses required for software development, and across the range of personnel available to perform software development.

TABLE II
Applicability of Existing Scientific Principles

Dimension	Software Engineering	Hardware Engineering
Scope Across Life Cycle	Some principles for component construction and detailed design, virtually none for system design and integration, e.g., algorithms, automata theory.	Many principles applicable across life cycle, e.g., communication theory, control theory.
Scope Across Application	Some principles for "systems" software, virtually none for applications software, e.g., discrete mathematical structures.	Many principles applicable across entire application system, e.g., control theory application.
Engineering Economics	Very few principles which apply to system economics, e.g., algorithms.	Many principles apply well to system economics, e.g., strength of materials, optimization, and control theory.
Required Training	Very few principles formulated for consumption by technicians, e.g., structured code, basic math packages.	Many principles formulated for consumption by technicians, e.g., handbooks for structural design, stress testing, maintainability.

For perspective, a similar summary assessment is presented in Table II for hardware engineering. It is clear from Table II that software engineering is in a very primitive state as compared to hardware engineering, with respect to its range of scientific foundations. Those scientific principles available to support software engineering address problems in an area we shall call *Area 1: detailed design and coding of systems software* by *experts* in a relatively *economics-independent* context.

Unfortunately, the most pressing software development problems are in an area we shall call *Area 2: requirements analysis design, test, and maintenance of applications software* by *technicians*[3] in an *economics-driven* context. And in Area 2, our scientific foundations are so slight that one can seriously question whether our current techniques deserve to be called "software engineering."

Hardware engineering clearly has available a better scientific foundation for addressing its counterpart of these Area 2 problems. This should not be too surprising, since "hardware science" has been pursued for a much longer time, is easier to experiment with, and does not have to explain the performance of human beings.

What is rather surprising, and a bit disappointing, is the reluctance of the computer science field to address itself to the more difficult and diffuse problems in Area 2, as compared with the more tractable Area 1 subjects of automata theory, parsing, computability, etc. Like most explorations into the relatively unknown, the risks of addressing Area 2 research problems in the requirements analysis, design, test and maintenance of applications software are relatively higher. But the prizes, in terms of payoff to practical software development and maintenance, are likely to be far more rewarding. In fact, as software engineering begins to solve its more difficult Area 2 problems, it will begin to lead the way toward solutions to the more difficult large-systems problems which continue to beset hardware engineering.

REFERENCES

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

[2] T. A. Dolotta *et al*., *Data Processing in 1980-85*. New York: Wiley-Interscience, 1976.

[3] P. Wegner, "Computer languages," *IEEE Trans. Comput.*, this issue, pp. 1207-1225.

[4] M. E. Fagan, "Design and code inspections and process control in the development of programs," IBM, rep. IBM-SDD TR-21.572, Dec. 1974.

[5] E. B. Daly, "Management of software development," *IEEE Trans. Software Eng.*, to be published.

[6] W. W. Royce, "Software requirements analysis, sizing, and costing," in *Practical Strategies for the Development of Large Scale Software*, E. Horowitz, Ed. Reading, MA: Addison-Wesley, 1975.

[7] T. E. Bell and T. A. Thayer, "Software requirements: Are they a problem?," *Proc. IEEE/ACM 2nd Int. Conf. Software Eng.*, Oct. 1976.

[8] E. S. Quade, Ed., *Analysis for Military Decisions*. Chicago, IL: Rand-McNally, 1964.

[9] J. D. Couger and R. W. Knapp, Eds;, *System Analysis Techniques*. New York: Wiley, 1974.

[3] For example, a recent survey of 14 installations in one large organization produced the following profile of its "average coder": 2 years college-level education, 2 years software experience, familiarity with 2 programming languages and 2 applications, and generally introverted, sloppy, inflexible, "in over his head," and undermanaged. Given the continuing increase in demand for software personnel, one should not assume that this typical profile will improve much. This has strong implications for effective software engineering technology which, like effective software, must be well-matched to the people who must use it.

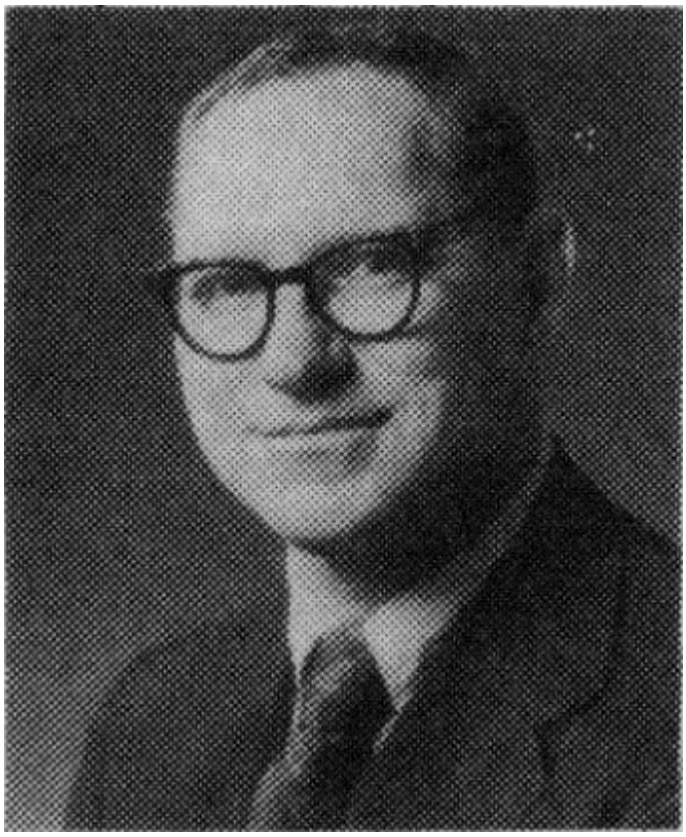
[10] D. Teichroew and H. Sayani, “Automation of system building,” *Datamation*, pp. 25-30, Aug. 15, 1971.
[11] C. G. Davis and C. R. Vick, “The software development system,” in *Proc. IEEE/ACM 2nd Int. Conf. Software Eng.*, Oct. 1976.
[12] M. Alford, “A requirements engineering methodology for real-time processing requirements,” in *Proc. IEEE/ACM 2nd Int. Conf. Software Eng.*, Oct. 1976.
[13] T. E. Bell, D. C. Bixler, and M. E. Dyer, “An extendable approach to computer-aided software requirements engineering,” in *Proc. IEEE/ACM 2nd Int. Conf. Software Eng.*, Oct. 1976.
[14] R. M. Balzer, “Imprecise program specification,” Univ. Southern California, Los Angeles, rep. ISI/RR-75-36, Dec. 1975.
[15] W. A. Martin and M. Bosyj, “Requirements derivation in automatic programming,” in *Proc. MRI Symp. Comput. Software Eng.*, Apr. 1976.
[16] N. P. Dooner and J. R. Lourie, “The application software engineering tool,” IBM, res. rep. RC 5434, May 29, 1975.
[17] M. L. Wilson, “The information automat approach to design and implementation of computer-based systems,” IBM, rep. IBM-FSD, June 27, 1975.
[18] B. W. Boehm, “Some steps toward formal and automated aids to software requirements analysis and design,” *Proc. IFIP Cong.*, 1974, pp. 192-197.
[19] A. B. Endres, “An analysis of errors and their causes in system programs,” *IEEE Trans. Software Eng.*, pp. 140-149, June 1975.
[201 T. A. Thayer, “Understanding software through analysis of empirical data,” *Proc. Nat. Comput. Conf.*, 1975, pp. 335-341.
[21] B. W. Boehm, R. L. McClean, and D. B. Urfrig, “Some experience with automated aids to the design of large-scale reliable software,” *IEEE Trans. Software Eng.*, pp. 125-133, Mar. 1975.
[22] P. Freeman, “Software design representation: Analysis and improvements,” Univ. California, Irvine, tech. rep. 81, May 1976.
[23] E. L. Glaser *et al.*, “The LOGOS project,” in *Proc. IEEE COMPCON*, 1972, pp. 175-192.
[24] N. R. Nielsen, “ECSS: Extendable computer system simulator,” Rand Corp., rep. RM-6132-PR/NASA, Jan. 1970.
[25] D. W. Kosy, “The ECSS II language for simulating computer systems,” Rand Corp., rep. R-1895-GSA, Dec. 1975.
[26] E. W. Dijkstra, “Complexity controlled by hierarchical ordering of function and variability,” in *Software Engineering*, P. Naur and B. Randell, Eds. NATO, Jan. 1969.
[27] H. D. Mills, “Mathematical foundations for structured programming,” IBM-FSD, rep. FSC 72-6012, Feb. 1972.
[28] C. L. McGowan and J. R. Kelly, *Top-Down Structured Programming Techniques.* New York: Petrocelli/Charter, 1975.
[29] B. W. Boehm et at., “Structured programming: A quantitative assessment,” *Computer*, pp. 38-54, June 1975.
[30] W. P. Stevens, G. J. Myers, and L. L. Constantine, “Structured design,” *IBM Syst. J.*, vol. 13, no. 2, pp. 115-139, 1974.
[31] G. J. Myers, *Reliable Software Through Composite Design*. New York: Petrocelli/Charter, 1975.

[32] D. L. Parnas, “On,the criteria to be used in decomposing systems into modules,” *CACM*, pp. 1053-1058, Dec. 1972.
[33] D. L. Parnas, “The influence of software structure on reliability,” in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 358-362, available from IEEE.
[34] M. Hamilton and S. Zeldin, “Higher order software-A methodology for defining software,” *IEEE Trans. Software Eng.*, pp. 9-32, Mar. 1976.
[35] “HIPO-A design aid and documentation technique,” IBM, rep. GC20-1851-0, Oct. 1974.
[36] J. Mortison, “Tools and techniques for software development process visibility and control,” in *Proc. ACM Comput. Sci. Conf.*, Feb. 1976.
[37] I. Nassi and B. Schneiderman, “Flowchart techniques for structured programming,” *SIGPLAN Notices*, pp. 12-26, Aug. 1973.
[38] D. T. Ross, J. B. Goodenough, and C. A. Irvine, “Software engineering: Process, principles, and gbals,” *Computer*, pp. 17-27, May 1975.
[39] M. A. Jackson, *Principles of Program Design*. New York: Academic, 1975.
[40] P. Henderson and R A. Snowden, “A tool for structured program development,” in *Proc. 1974 IFIP Cong.*, pp. 204-207.
[41] L. C. Carpenter and L. L. Tripp, “Software design validation tool,” in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 395-400, available from IEEE.
[42] S. H. Caine and E. K. Gordon, “PDL: A tool for software design,” in *Proc. 1975 Nat. Comput. Conf.*, pp. 271-276.
[43] H. D. Mills, “Structured programming in large systems,” IBM-FSD, Nov. 1970.
[44] C. A. R. Hoare, “Notes on data structuring,” in *Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. New York: Academic, 1972.
[45] N. Wirth, “An assessment of the programming language Pascal,” *IEEE Trans. Software Eng.*, pp. 192-198, June 1975.
[46] P. Brinch-Hansen, “The programming language concurrent Pascal,” *IEEE Trans. Software Eng.*, pp. 199-208, June 1975.
[47] W. A. Wulf, *ALPHARD: Toward a language to support structured programs*, Carnegie-Mellon Univ., Pittsburgh, PA, internal rep., Apr. 30, 1974.
[48] B. H. Liskov and S. Zilles, “Programming with abstract data types,” *SIGPLAN Notices*, pp. 50-59, April 1974.
[49] F. T. Baker, “Structured programming in a production programming environment,” *IEEE Trans. Software Eng.*, pp. 241-252, June 1975.
[501 R. D. Williams, “Managing the development of reliable software,” *Proc., 1975 Int. Conf. Reliable Software*, April 1975, pp. 3-8, available from IEEE.
[51] J. Witt, “The COLUMBUS approach,” *IEEE Trans. Software Eng.*, pp. 358-363, Dec. 1975.
[52] B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style*. New York: McGraw-Hill, 1974.
[53] H. F. Ledgard, *Programming Proverbs*. Rochelle Park, NJ: Hayden, 1975
[54] W. A. Whitaker *et al.*, “Department of Defense requirements for high order computer programming languages: ‘Tinman,’ “ Defense Advanced Research Projects Agency, Apr. 1976.
[55] *Proc. 1973 IEEE Symp. Comput. Software Reliability*, Apr.–May 1973.

[56] E. C. Nelson, "A statistical basis for software reliability assessment," TRW Systems, Redondo Beach, CA, rep. TRW-SS-73-03, Mar. 1973.
[57] J. R. Brown and M. Lipow, "Testing for software reliability," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 518-527.
[58] J. D. Musa, "Theory of software reliability and its application," *IEEE Trans. Software Eng.*, pp. 312-327, Sept. 1975.
[59] R. J. Rubey, J. A. Dana, and P. W. Biche, "Quantitative Aspects of software validation," *IEEE Trans. Software Eng.*, pp. 150-155, June 1975.
[60] T. A. Thayer, M. Lipow, and E. C. Nelson, "Software reliability study," TRW Systems, Redondo Beach, CA, rep. to RADC, Contract F30602-74-C-0036, Mar. 1976.
[61] D. J. Reifer, "Automated aids for reliable software," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 131-142.
[62] C. V. Ramamoorthy and S. B. F. Ho, "Testing large software with automated software evaluation systems," *IEEE Trans. Software Eng.*, pp. 46-58, Mar. 1975.
[63] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, pp. 156-173, June 1975.
[64] P. Wegner, "Report on the 1975 International Conference on Reliable Software," in *Findings and Recommendations of the Joint Logistics Commanders' Software Reliability Work Group*, Vol. II, Nov. 1975, pp. 45-88.
[65] J. C. King, "A new approach to program testing," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 228-233.
[66] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select-A formal system for testing and debugging programs," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 234-245.
[67] J. Goldberg, Ed., *Proc. Symp. High Cost of Software*, Stanford Research Institute, Stanford, CA, Sept. 1973, p. 63.
[68] L. C. Ragland, "A verified program verifier," Ph.D. dissertation, Univ. of Texas, Austin, 1973.
[69] D. I. Good, R. L. London, and W. W. Bledsoe, "An interactive program verification system," *IEEE Trans. Software Eng.*, pp. 59-67, Mar. 1975.
[70] F. W. von Henke and D. C. Luckham, "A methodology for verifying programs," in *Proc. 1975 Int. Conf. Reliable Software*, pp. 156-164, Apr. 1975.
[71] C. A. R. Hoare and N. Wirth, "An axiomatic definition of the programming language PASCAL," *Acta Informatica*, vol. 2, pp. 335-355, 1973.
[72] R. L. London, "A view of program verification," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 534-545.
[73] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. New York: Academic, 1972.
[74] W. A. Wulf, "Reliable hardware-software architectures," *IEEE Trans. Software Eng.*, pp. 233-240, June 1975.
[75] J. Goldberg, "New problems in fault-tolerant computing," in *Proc. 1975 Int. Symp. Fault-Tolerant Computing*, Paris, France, pp. 29-36, June 1975.
[76] B. Randell, "System structure for software fault-tolerance," *IEEE Trans. Software Eng.*, pp. 220-232, June 1975.
[77] R. K. McClean and B. Press, "Improved techniques for reliable software using microprogrammed diagnostic emulation," in *Proc. IFAC Cong.*, Vol. IV, Aug. 1975.

[78] E. B. Swanson, “The dimensions of maintenance,” in *Proc. IEEE/ACM 2nd Int. Conf. Software Eng.*, Oct. 1976.
[79] B. W. Boehm, J. R. Brown, and M. Lipow, “Quantitative evaluation of software quality,” in *Proc. IEEE/ACM 2nd Int. Conf. Software Eng.*, Oct. 1976.
[80] J. L. Elshoff, “An analysis of some commercial PL/I programs,” *IEEE Trans. Software Eng.*, pp. 113-120, June 1976.
[81] W. L. Trainor, “Software: From Satan to saviour,” in *Proc., NAECON*, May 1973.
[82] E. H. Sibley, Ed., *ACM Comput. Surveys (Special Issue on Data Base Management Systems)*, Mar. 1976.
[83] *Defense Management J. (Special Issue on Software Management)*, vol. II, Oct. 1975.
[84] L. A. Belady and M. M. Lehman, “The evolution dynamics of large programs,” IBM Research, Sept. 1975.
[85] F. P. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.
[86] E. Horowitz, Ed., *Practical Strategies for Developing Large-Scale Software*. Reading, MA: Addison-Wesley, 1975.
[87] G. F. Weinwurm, Ed., *On the Management of Computer Programming*. New York: Auerbach, 1970.
[88] P. Naur and B. Randell, Eds., *Software Engineering*, NATO, Jan. 1969.
[89] P. J. Metzger, *Managing a Programming Project*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[90] J. C. Shaw and W. Atkins, *Managing Computer System Projects*. New York: McGraw-Hill, 1970.
[91] G. F. Hice, W. S. Turner, and L. F. Cashwell, *System Development Methodology*. New York: American Elsevier, 1974
[92] W. J. Ridge and L. E. Johnson, *Effective Management of Computer Software*. Homewood, IL: Dow Jones-Irwin, 1973.
[93] T. R. Gildersleeve, *Data Processing Project Management*. New York: Van Nostrand Reinhold, 1974.
[94] G. F. Weinberg, *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.
[95] J. D. Aron, *The Program Development Process: The Individual Programmer*. Reading, MA: Addison-Wesley, 1974.
[96] R. W. Wolverton, “The cost of developing large-scale software,” *IEEE Trans. Comput.*, 1974.
[97] M. H. Halstead, “Toward a theoretical basis for estimating programming effort,” in *Proc. Ass. Comput. Mach. Conf.*, Oct. 1975, pp. 222-224.
[98] *Summary Notes, Government/Industry Software Sizing and Costing Workshop*, USAF Electron. Syst. Div., Oct. 1974.
[99] B. S. Barry and J. J. Naughton, “Chief programmer team operations description,” U. S. Air Force, rep. RADC-TR-74-300, Vol. X (of 15-volume series), pp. 1-2-1-3.
[100] *Software Development and Configuration Management Manual*, TRW Systems, Redondo Beach, CA, rep. TRW-SS-73-07, Dec. 1973.
[101] H. Bratman and T. Court, “The software factory,” *Computer*, pp. 28-37, May 1975.
[102] “Systems design laboratory: Preliminary design report,” Naval Electronics Lab. Center, Preliminary Working Paper, TN-3145, Mar. 1976.

[103] W. E. Carlson and S. D. Crocker, “The impact of networks on the software marketplace,” in *Proc. EASCON*, Oct. 1974.
[104] “Management of computer resources in major defense systems,” Department of Defense, Directive 6000.29, Apr. 1976.



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

He entered the computing field as a Programmer in 1955 and has variously served as a Numerical Analyst, System Analyst, Information System Development Project Leader, Manager of groups performing such tasks, Head of the Information Sciences Department at the Rand Corporation, and as Director of the 1971 Air Force CCIP-85 study. He is currently Director of Software Research and Technology within the TRW Systems and Energy Group, Redondo Beach, CA. He is the author of papers on a number of computing subjects, most recently in the area of software requirements analysis and design technology. He serves on several governmental advisory committees, and currently is Chairman of the NASA Research and Technology Advisory Committee on Guidance, Control, and Information Systems.

Dr. Boehm is a member of the IEEE Computer Society, in which he currently serves on the Editorial Board of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and on the Technical Committee on Software Engineering.