The State of Software Maintenance

NORMAN F. SCHNEIDEWIND, SENIOR MEMBER, IEEE

***Abstract*—A state of software maintenance survey is presented, indicating the incongruity of the simultaneous existence of importance and neglect in this field. An overview is given of selected developments and activities covering the following topics:**

- **The "Maintenance Problem."**
- **Models.**
- **Methods for improving maintenance.**
- **Metrics.**
- **Maintenance information management.**
- **Standards.**
- **Maintenance of existing code.**
- **Surveys.**

The paper concludes with a prognosis of what is ahead in maintenance: a battle and tradeoff between the forces for maintaining the base of existing software and the forces for the evolution of new systems. An Appendix is provided for the reader who desires information about a software maintenance conference and a special interest group.

***Index Terms*—Metrics, models, software maintenance.**

I. Introduction

To gauge the state of software maintenance, ask yourself these questions:

• How many articles have appeared in this TRANSACTIONS on the subject of maintenance in the last couple of years? Answer: none between August 1985 and November 1986, inclusive, and few prior to this period in the history of the TRANSACTIONS. However, before rushing to judgement about the "guilt" of TSE, realize that it is not unique in this regard among technical journals.

Some additional questions to ponder:

• How many computer science departments have a course in maintenance?

• How many doctoral dissertations have there been in maintenance?

To work in maintenance has been akin to having bad breath. Yet, examine the "Problem" below and ask yourself whether there is any justification for this neglect. But, first, a disclaimer, followed by some definitions so that we may proceed from a common reference point.

The information which follows represents a selected overview of the state of the software maintenance field. Since this is a survey paper, it is difficult to cover every aspect of the field, given time and page limitations. We apologize for any significant work which may not be covered.

Manuscript received October 31, 1986.
The author is with the Naval Postgraduate School, Monterey, CA 93943.
IEEE Log Number 8612830.

A. Definitions

Maintenance: Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment [1].

Maintainability: The ease with which a software system can be corrected when errors or deficiencies occur, and can be expanded or contracted to satisfy new requirements [2].

B. Why Is There a Maintenance Problem?

There is a maintenance problem because [3]:

• 75–80 percent of existing software was produced prior to significant use of structured programming.

• It is difficult to determine whether a change in code will affect something.

• It is difficult to relate specific programming actions to specific code.

The main problem in doing maintenance is that we cannot do maintenance on a system which was not designed for maintenance. Unless we design for maintenance, we will always be in a lot of trouble after a system goes into production.

In addition, there is the very significant personnel problem concerning the myth that there is no challenge for creative people in maintenance.

According to Zvegintzov, most software is immortal (immoral?). He says that all surveys of the distribution of effort between new systems and present systems show about a 50–50 split [4]. This is the case because of the following important considerations:

• Functions are added, not replaced.

• Every new function must be tied into the present system.

• Systems are not totally replaced, except for overriding economic or technical reasons.

• Organizations strive for compatibility in systems, not perfection.

Specifically, Lientz and Swanson report from a survey of data processing managers in 487 data processing organizations that: departments spend about half of their application staff time on maintenance; over 40 percent of the effort in supporting an operational application system is spent on providing user enhancements and extensions; the average application system is between three and four years old, consists of about 55 programs and 23 000 source statements, and is growing at a rate of over 10 percent a year; and about one-half man-year is allocated annually to maintain the average system [5].

If all programs to be maintained were well documented and cleanly structured, and used third-normal form data models and data dictionaries for generating the programmer's data, the task of the maintainer would be much easier. The problem for most maintainers is that they have to maintain ill-documented code that is covered with patches with no comprehensible structure and that has data representations buried in the program code. It is a major detective operation to find out how the program works, and each attempt to change it sets off mysterious bugs from the tangled undergrowth of unstructured code [2].

C. Why Is Maintenance Hard?

Maintenance is hard because [6]:

- We cannot trace the product nor the process that created the product.
- Changes are not adequately documented.
- Lack of change stability (See Metrics Section below).
- Ripple effect of making changes.
- Myopic view that maintenance is strictly a postdelivery activity.

One consequence of this lack of attention to maintainability requirements during design is loss of traceability. This is defined as the ability to identify the technical information which pertains to a software error detected during the operational phase (or other postrequirements phase) and thereby trace the error to the applicable design specifications and user requirements statements [7].

D. Why Is Maintenance Expensive?

In the early days of programming, when programmers' salaries were an almost insignificant percentage of the data processing budget, when programmers spent most of their time writing new programs, and when machine resources were expensive, the mark of a well written program was efficiency. Twenty years later, when programmers' salaries consume the majority of the data processing budget, when programmers spend most of their time maintaining programs, and when hardware is cheap, a new standard for well written programs has emerged: how maintainable are they, especially for future generations of programmers [8]?

E. Should Existing Code Be Discarded?

If existing code is so bad, why should it be retained? Belady believes that we cannot and should not declare "old" software obsolete or not worth studying [9]. This collection of functions is an important asset, embodying a wealth of experience, and constitutes an inventory of "ideas" for identifying the building blocks of future systems. Even if the code itself is inelegant and possibly not reusable, a study of the specifications and identification of the most frequently used components could reveal a set of generic classes of algorithms and functions which could be usable in future systems.

II. Models

A. Do We Have the Wrong Model for Maintenance?

Since the software industry does not seem to have a good understanding of and model for maintenance, it is worthwhile to consider some proposed models which provide insight into the maintenance process. Lehman suggests that change is intrinsic in software, and must be accepted as a fact of life, and since software undergoes change throughout its life, there is no reason to distinguish maintenance from initial development. Evolutionary development is inevitable [10], [11]. Furthermore, the very act of installing software changes the environment; pressures operate to modify the environment, the problem, and technological solutions. Changes generated by users and the environment and the consequent need for adapting the software to the changes is unpredictable and cannot be accommodated without iteration. Programs must be more alterable and the resultant change process must be planned and controlled. According to Lehman, large programs are never completed, they just continue to evolve. In other words, with software, we are dealing with a moving target and that, in effect, "maintenance" is performed continuously. Lehman suggests that the word "maintenance" not be used and that the term "program evolution" be used instead. If this model of the software process is correct, it suggests that change activity and change management should be an integral part of development and all other phases of the life of software. In this view, a change would be no more associated with "maintenance" than with development.

B. Do Requirements End in the Requirements Phase?

Lehman's view seems to be supported by Lientz and Swanson [12]. They state that the approach which is in vogue of getting requirements right before starting the design may be based on the fallacious assumption that requirements are fixed. The reality is that requirements change continually, often in response to organizational change. These changes are more likely to emanate from experience in the use of the system than from an abstract specification in the early design of the system. The major problem in requirements assessment may not be the development of a complete, consistent and unambiguous specification, prior to design, but, rather, the evolution of requirements which allow a timely response of the software to organizational change. Requirements assessment during maintenance may be as demanding as during development.

C. Is the Life Cycle Model Appropriate for Maintenance?

The traditional view of the software life cycle has done a disservice to maintenance by depicting it solely as a single step at the end of the cycle. In fact, it would be more accurately portrayed as 2nd, 3rd, · · · , *n*th round development [13]. The traditional view also fosters the idea that structured techniques are best applied to development, whereas their application to maintenance is equally valid.

In contrast, the traditional view of maintenance is that it is an activity confined to the postdelivery phase, is not directly related to development, and has its own special requirements.

III. Methods for Improving Maintenance

Software maintenance authors have made many suggestions for improving the maintainability of software. These suggestions can be classified into three categories: design approach, maintenance practices, and management.

A. What Design Approaches Are Needed?

Software design practices should include criteria for maintainability [14]. These criteria are the following:

- Design software with maintainability in mind.
- Develop design criteria for achieving maintainability.
- Simplicity should outweigh completeness.
- Change management should be used to:
  —Limit the effects in the maintenance phase of a change made in the design phase.
  —Determine ripple effects on other modules of making a change to a common module:
    - global variables.
    - modules which invoke or are invoked by a common module.

  —Determine the effect on a module of making a change to a local variable.
- Evaluate the design for excessive complexity.

Another design approach to aid maintenance describes the design in terms of parts and the interconnections of those parts [15]. With parts interconnections as the focal point of the design and documentation, the system maintainer can more readily judge the possible ripple effect of change. Three levels—system, assembly, and component—are shown in list and graphical form, where each succeeding level provides a more detailed description of the previous level. A parts list and connections list is shown for each level. The parts list shows functions and the data associated with the functions. The connections list shows the input/output relationships between functions and data.

B. What Maintenance Practices Are Needed?

Maintainability can be significantly improved if the following practices are used [16], [14], [17], [18]:

- Change Management:
  —Make easiest changes first.
  —Change one module at a time.
  —Inspect proposed changes for each type of side effect.
  —Run regression tests after every change.
- Produce guidelines for modifying and retesting software.
  —Provide information to support assessment of the impact of a change in various parts of the software.

C. What Other Practices Are Needed?

- Identify source statements which have been changed with a number which is associated with the change request.
- Learn to *read* programs (alien code).
- Keep diaries of bugs and maintenance issues.
- Centralize variable declarations in a program.
  —Use abstract data types to define the legitimate types and values which objects of a type may assume and a set of operations which may be performed on that type.
- Centralize symbolically defined and referenced database definitions in a computer processible data dictionary.
- Since it can be taken for granted that software will evolve and change, each programmer in the process should give consideration to the next programmer in the life cycle.

One of the major sources of error in making maintenance modifications arises when neither the program nor the documentation reveal that sections of a program that are far apart are related. As suggested by Letovsky and Soloway, this may cause the programmer to make assumptions about the plans of a program which are based purely on local information. This can lead to an inaccurate understanding of the program as a whole [19]. Their solution to this problem is to provide answers to the two questions:

> What information needs to be provided to the reader of the program?
> When and how should this information be provided?

Easily accessible information is needed to form correct interpretations of delocalized plans. What is needed is to move from documenting the code itself to documenting the plans in the code. A tool which may provide this capability is under development.

D. What Management Policies Are Needed?

Some guidelines offered by McClure [14] for improving the management of maintenance are the following:

- Involve maintainers in design and testing.
- Put the same emphasis on the use of standards in maintenance as in design.
- Rotate personnel between design and maintenance.
- Make design documentation available to maintainers at design time.
- Carry over the use of design tools into maintenance.
- Use configuration management and change request procedures.
- Establish a liaison between users and maintenance.

Boehm suggests the following for maximizing the motivation and, hence, productivity of maintenance personnel [20]:

• Couple software objectives to organizational goals.

• Couple software maintenance rewards to organizational performance.

• Integrate software maintenance personnel into operational teams.

• Create a discretionary perfective maintenance budget.

• Create the perquisites of software ownership.

—Participation by maintenance personnel in: development standards creation, development reviews, and acceptance test preparation.

• Rectify the negative image of software maintenance.

The reader can see that the objective of these guidelines is to *integrate* design and maintenance—to reverse the current procedure of considering and managing these activities in disparate and separate functions.

E. *What Tools Are Needed?*

Tools are needed in maintenance to look for (and hopefully find) structure [2]:

• Looking for structure. Several types of structure need to be understood:

—Procedural structure.

—Control structure.

—Data structure.

—Input/output structure.

• Understanding data aliases:

—Data may be referred to by several names.

• Following data flow:

—Where do data originate? Where are they used?

• Following control flow:

—The consequences of executing each path must be understood.

• Understanding versions of a program:

—How does a change affect different versions of a program?

Tools are available which address the above areas. These consist of displaying the following [2]:

• Structure Chart: Shows hierarchy and call/called relationships.

• Data Trace: Origins, uses, and modifications of variables.

• Control Trace: Shows control flow statements and indicates how a destination can be reached from a given origin.

• Version Comparisons: Statements which differ between two versions of a program are highlighted.

Additionally, the following tool capabilities are useful for maintenance [21]:

• Stored test execution information giving dynamic behavior of a program.

• Test cases to exercise modified sections of a program.

• Symbolic and actual execution information.

An interesting tool is one designed to restructure unstructured code. One tool of this type is called structured retrofit, involving the restructuring of Cobol programs [22]. The application of this kind of tool rests on the premise that with 7 out of 10 programmers involved in maintenance, and costs for this activity soaring, maintenance must be made easier and, hence, less costly. Furthermore, the argument is made that even if the code is bad, the logic of the design may not be bad. In other words, the design concept was good but its implementation was poor. Since this poor code is meeting user requirements but is difficult and costly to maintain, it should be salvaged, where feasible.

The structured retrofit procedure consists of the following steps:

• Scoring: Programs are evaluated as candidates for restructuring by scoring them against the following criteria:

—Degree of structure.

—Level of nesting.

—Degree of complexity.

—Breakout of verb utilization.

—Analysis of potential failure modes.

—Trace of control logic.

However, even if a program scores low on the above criteria but still runs with little time required for maintenance, it will not be retrofitted.

• Compilation: Programs which are to be retrofitted are compiled. Programs which do not compile cleanly are referred to others for resolution. Programs continue in the retrofit process only if they compile cleanly.

• Restructuring: Programs that are unstructured are put through this process to make them structured. The resulting program will produce the same transformation on the same input data as the original program.

• Formatting: Programs are made more readable through the formatting process. They are then recompiled to pick up possible syntax errors.

• Validation: The same inputs are applied to the original and restructured programs and the outputs are compared on a bit-by-bit basis by a file-to-file compare utility.

• Optimization: The code is optimized to reduce overhead which may have been introduced by the restructuring process.

It is claimed that structured retrofit has been able to restructure 60 percent of programs offered automatically, another 20 percent with some manual intervention, and 20 percent cannot be restructured cost-effectively.

IV. Metrics

In order to perform maintenance effectively, we must be able to measure the effects of design approaches on maintenance and, especially important, be able to measure the effects of maintenance approaches on future maintenance!

Ideally, we want maintenance to improve software. Our minimum objective is that maintenance should have a neutral effect. Unfortunately, too often, maintenance makes software worse, due to unforseen ripple effect. In order to minimize ripple effect, software must be stable. Stability must be achieved at design time, not during the maintenance phase. Stability in design is achieved by minimizing potential ripple effect caused by interaction between modules (i.e., a change to a module causes undesirable changes to other modules). The definitions and concepts applicable to achieving design stability were developed by Yau and Collofello [23]; these are the following:

- Program stability: Quality attribute indicating the resistance to the potential ripple effect which a program would have when it is modified.
- Module stability: A measure of the resistance to the potential ripple effect of a modification of the module on other modules in the program.
- Logical stability: Measure of resistance to impact of modification on other modules in terms of logical considerations.
- Performance stability: Measure of resistance to impact of modification on other modules in terms of performance considerations.
- ''Maintenance activity'' is a change to a single variable.
- Intramodule change propagation involves flow of changes within the module as a consequence of a modification.
- Intermodule change propagation involves flow of changes across modules as a consequence of a modification.
- Intramodule change propagation is utilized to identify the set of interface variables which are affected by logical ripple effect as a consequence of a modification to a variable definition in a module. This requires an identification of which variables constitute the module's interfaces and the potential intramodule change propagation among the variables in the module.
- Once an interface variable is affected by a change, the flow of changes may cross module boundaries and affect other modules. Interface change propagation is used to identify the set of modules involved in intermodule change propagation as a consequence of affecting an interface variable in a module.
- Measure the complexity of affected modules to analyze the possible relationship between complexity and vulnerability to ripple effect.
- Compare stability of alternate versions of module for the purpose of making a design choice. (However, there may be no time available to design alternatives.)
- Use as predictor of amount of maintenance required.
- Reject request for maintenance if it involves modifying unstable modules.
- Restructure modules with poor stability.

The measure of design stability of a module, proposed by Yau and Collofello [23], [24], is the reciprocal of the total number of assumptions made by other modules about the given module. If the given module has poor design stability and it is modified, it is likely to produce undesirable effects on other modules, which either invoke, share global data with, or are invoked by the given module. The rationale of this metric is that modules which cause large ripple effects, if modified, are among the modules with poor design stability. This definition of stability only applies to modular software. This point illustrates one of the difficulties in trying to improve maintenance: much of the existing software which must be maintained is not modular!

More information needs to be captured in a metric than just the effects of a change to a single variable or the effects of changes to a set of variables. What is needed is the effects of changes on other aspects of a program, such as documentation. Also, since all assumptions are not equally important, this metric could possibly be improved by weighting the assumptions. Although this metric addresses an important aspect of maintainability dealing with assumptions that are made about interfaces between modules, it is silent on the subject of intramodule design characteristics. Despite these limitations, this metric would be very useful, primarily, for deciding among design alternatives for *new* software.

An approach to assessing the difficulty of *maintaining* a program is to quantify program difficulty as the sum of the difficulties of the constituent parts [25]. A Maintainability Analysis Tool was developed to analyze the difficulty of understanding and maintaining Fortran programs by assigning weights, which represent relative difficulty of understanding, to various program attributes, such as syntactic elements (e.g., parameter) and syntactic attributes (e.g., name in COMMON). The numeric weights and factors are summed for a program to yield a measure of difficulty. Obviously, there can be a lot of subjectivity involved in assigning weights and measures.

A strategy for determining whether to continue to *maintain* software is to focus on modules which may be candidates for rewriting. These error prone modules need to be identified. One method for identifying error prone modules is to have maintenance personnel record information about: 1) which modules were changed, 2) how much effort was involved in making the changes, and 3) reasons for making the changes [26].

Another aspect of applying metrics to maintenance is the establishment of criteria for determining whether maintenance is being *performed* effectively. Arnold and Parker [27] established the following criteria for 40 telemetry processing projects at the NASA/Goddard Space Flight Center:

- Desired effort distribution: Distribution of maintenance effort between enhancements/restructurings and fixes.
- Desired frequency distribution: Distribution of reports approved for action between enhancements/restructurings and fixes.
- Completion rates: Rates for enhancements/restructurings and fixes.
- Effort per change: Labor time limits for enhancements/restructurings and fixes.

V. Maintenance Information Managemeent

Since maintenance usually involves having to understand what someone else did to the code, information about the characteristics of the code and specifications (if they exist) are essential to doing an effective job of maintenance. Important elements of the information base are: control flow information, data flow information, and declaration information [28]. This information base should be established as part of every *design* and *maintenance* activity.

VI. Standards

In general, development standards have been inappropriate for use in maintenance [29]. Of greater concern is the fact that standards efforts have not addressed maintenance.

Although no standards exist for maintenance, management guides are available from the National Bureau of Standards, which provide methodologies and procedures for conducting an effective maintenance program [30], [31]. Among the recommendations of [30] are the following:

- Develop a software maintenance plan.
- Recognize improvement of maintainability.
- Elevate maintenance visibility in the organization.
- Reward maintenance personnel; provide a career path and training.
- Establish and enforce standards.

The major conclusion of [31] is that, in addition to developing software with maintenance in mind, software must also be maintained with maintenance in mind!

VII. Maintenance of Existing Code

This activity involves maintaining software which has not been modularly designed. It dominates maintenance work.

A. Restructuring

As reported above, in the description of the structured retrofit system, unstructured code can be converted to a structured format. Due a proof by Jacopini and Bohm, any computable algorithm in any language, can be represented by a structured graph. This result is the basis for restructuring programs. Unstructured programs typically have graphs whose nodes are so connected that the graph cannot be effectively partitioned into independent regions. However, by means of a graph simplification process, an unstructured program can be rendered into a structured form. The original unstructured program is parsed into an abstract syntax tree. Several tree to tree transformations are performed to reduce the tree to a few simple control flow expressions. When the tree is sufficiently simple, it is transformed into a control flow graph. This simplification process terminates when the topology of the graph represents a structured algorithm.

Although the method sounds impressive there are problems in restructuring when the program has GOTO's. In addition, an enormous amount of machine time may be required to develop new control graphs representing the new structure [8]. However, if it is determined that restructuring is more economical than rewriting, it is clear that restructuring is only feasible when an automated method such as this is used.

It is not always feasible to make unstructured code look like structured code and more readable by using structured documentation. In a study conducted by Schneidewind [32] to analyze the effectiveness of documentation for maintenance purposes, where the documentation had been created with the intent of making the software more readable and understandable by showing a 'hierarchical structure' of unstructured code, it was found that the new documentation did not always tell the truth about the code logic as represented by the program listing. The reason for this was that the hierarchical documentation could not faithfully describe software which was unstructured. Lesson learned: the code must also be restructured.

B. Recovering the Design with Abstract Specifications

For situations in which no specifications exist, a technique called Maintenance by Abstraction is claimed to allow one to recover the design by using the following steps [33]:

- Inspect the code.
- Propose a set of abstractions (directed graph representations of the code).
- Choose the most suitable set of abstractions.
- Construct a specification from the abstractions.

The recovered design (i.e., the specification derived above) is then applied to the Transformation-based Maintenance Model. The directed graph representation of the code is examined to find nodes representing design decisions such that the order of design decisions can be reversed—for the purpose of making maintenance changes—in a way that will not affect the final implementation. It appears that this complex procedure would only be cost effective on large programs.

The IBM Federal Systems Division is upgrading the Federal Aviation Administration National Airspace System, 20 year old, 100 000 line, en route software by modeling programs as either function abstractions (transforms a value in input domain to output range) or data abstractions (class of data objects and the set of operations performed on them) [34]. Function abstractions can also be regarded as entities which do not retain data across invocations and data abstractions as entities which do retain data. The abstractions were used by the designer to determine the required change (added, deleted, and updated functions as needed).

VIII. Surveys

To provide a feel for the characteristics of maintenance as practiced in various organizations, results from several surveys are presented briefly below.

In a survey of 487 data processing organizations, it was found that most maintenance is perfective (55 percent): performed to enhance performance, improve maintainability, or improve executing efficiency. This is followed by adaptive maintenance (25 percent): performed to adapt software to changes in the data requirements or processing environments. Lastly, there is corrective maintenance (20 percent): performed to identify and correct software failures, performance failures, and implementation failures [12].

Chapin [35] reports that from a limited survey of users of fourth generation languages that although these languages are beneficial for development, their use may make maintenance more difficult and expensive. One reason he cites for this situation is that interprogram and intersystem communication of data with these languages is often obscure, thus rendering the effect of a maintenance action unclear.

In another survey by Chapin [36], he reports on information collected from supervisory personnel closest to software maintenance work. The survey consisted of 260 questionnaires collected from 123 data processing installations; there were 769 responses across the various questions. The biggest problems identified were poor documentation and inadequate staff. With regard to the latter, there is a problem in matching the characteristics of the software to be maintained with appropriate personnel.

IX. Prognosis

Much of the problem will remain of being condemned to maintain existing, nonstructured code for a long time into the future—perhaps 20 years. This situation will only change when two things happen: 1) software development environments become so effective and programmer productivity becomes so great that it will be more economical to develop new systems than to maintain old systems; 2) organizations want to do business in *new* ways. Thus the decision will not be over the cost of reprogramming or redesign, but about whether organizations will adapt their information systems to support the organization's survival in a changing world. In the interim, restructuring techniques will be an important tool for attempts to convert a "sow's ear into a silk purse." In making the restructuring decision, only relevant costs should be considered. The fact that a lot of money has been spent in the past is irrelevant to making a decision about the future. These are sunk costs; only future costs should be considered. The cost to rewrite, redesign, or develop a new system *are* relevant costs.

On the personnel front, there is hope. Software engineers are being sensitized to the need for considering maintainability in their designs. More academics will do research in maintenance when academic administrators recognize the importance of maintenance. Computer science programs will contain a course on maintenance when academics themselves recognize the importance of maintenance!

Appendix

The following lists some information about an important conference and a special interest group in the field of software maintenance:

A. Conference

The first conference in software maintenance, sponsored by technical societies, was the Software Maintenance Workshop, held at the Naval Postgraduate School, Monterey, CA, December 6–8, 1983 [37]. It was sponsored by the IEEE Technical Committee on Software Engineering of the IEEE Computer Society, National Bureau of Standards, and the Naval Postgraduate School, and in cooperation with the ACM Special Interest Group on Software Engineering.

The second conference was the Conference on Software Maintenance–1985, held at the Sheraton Inn Washington–Northwest, Washington, DC, November 11–13, 1985. It was sponsored by the same organizations as above, minus the Naval Postgraduate School, and with the addition of the Data Processing Management Association, and in cooperation with the Association for Women in Computing, and the Software Maintenance Association.

The next conference, Conference on Software Maintenance–1987, will be held in Austin, TX, September 21–24, 1987. For information contact:

Roger J. Martin, General Chair
National Bureau of Standards
Bldg. 225, Rm B266
Gaithersburg, MD 20899
(301) 921-3545

B. Special Interest Group

The Software Maintenance Association (SMA) is a special interest group in the field. For information about this organization and a maintenance newsletter, contact:

Nicholas Zvegintzov
141 Marks Place, #5F
Staten Island, NY 10301
(718) 981-7842.

References

[1] *An American National Standard IEEE Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Standard 729, 1983.
[2] J. Martin and C. McClure, *Software Maintenance: The Problem and Its Solutions*. Englewood Cliffs, NJ: Prentice-Hall, 1983.
[3] D. P. Freedman and G. M. Weinberg, "A checklist for potential side effects of a maintenance change," in *Techniques of Program and System Maintenance*, Girish Parikh, Ed. Ethotech., Inc., 1980, pp. 61–68.
[4] N. Zvegintzov, "Nanotrends," *Datamation*, pp. 106–116, Aug. 1983.
[5] B. P. Lientz and B. E. Swanson, "Problems in application software maintenance," *Commun. ACM*, vol. 24, no. 11, pp. 763–769, Nov. 1981.
[6] N. F. Schneidewind, "Quality metrics standards applied to software maintenance" (Abstract), in *Proc. Comput. Standards Conf. 1986* (Addendum), IEEE Comput. Soc., May 113–15, 1986.
[7] M. B. Kline and N. F. Schneidewind, "Life cycle comparisons of hardware and software maintainability," in *Proc. Third Nat. Rel. Conf.*, Birmingham, England, Apr./May 1981, p. 4A/3/1–4A/3/14.
[8] E. Bush, "The automatic restructuring of COBOL," in *Proc. Conf. Software Maintenance-1985*. Washington, DC: IEEE Comput. Soc. Press, Nov. 1985, pp. 35–41.
[9] L. A. Belady, "Evolved software for the 80's," *Computer*, vol. 12, no. 2, pp. 79–82, Feb. 1979.
[10] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proc. IEEE*, vol. 68, no. 9, Sept. 1980.

[11] ——, "Program evolution," Dep. Computing, Imperial College of Science and Technology, London SW7 2BZ, England, Res. Rep. DoC 82/1, Dec. 1982.

[12] B. P. Lientz and E. B. Swanson, *Software Maintenance Management*. Reading, MA: Addison-Wesley, 1980.

[13] J. R. McKee, "Maintenance as a function of design," in *AFIPS Conf. Proc.*, vol. 53, 1984 Nat. Comput. Conf., pp. 187-193.

[14] C. L. McClure, *Managing Software Development and Maintenance*. New York: Van Nostrand, 1981.

[15] J. Silverman, N. Giddings, and J. Beane, "An approach to design-for-maintenance," in *Proc. Software Maintenance Workshop*, R. S. Arnold, Ed. Washington, DC: IEEE Comput. Soc. Press, Dec. 1983, pp. 106-110.

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

[17] J. B. Munson, "Software maintainability: A practical concern for life-cycle costs," *Computer*, vol. 14, no. 11, pp. 103-109, Nov. 1981.

[18] E. Yourdon, "Structured maintenance," in *Techniques of Program and System Maintenance*, Girish Parikh, Ed. Ethotech, Inc., 1980, pp. 211-213.

[19] S. Letovsky and E. Soloway, "Delocalized plans and program comprehension," *IEEE Software*, vol. 3, no. 3, pp. 41-49, May 1986.

[20] B. Boehm, "The economics of software maintenance," in *Proc. Software Maintenance Workshop*, R. S. Arnold, Ed. Washington, DC: IEEE Comput. Soc. Press, Dec. 1983, pp. 9-37.

[21] Z. Kishimoto, "Testing in software maintenance and software maintenance from the testing perspective," in *Proc. Software Maintenance Workshop*, R. S. Arnold, Ed. Washington, DC: IEEE Comput. Soc. Press, Dec. 1983, pp. 166-117.

[22] M. J. Lyons, "Salvaging your software asset (tools based maintenance)," in *AFIPS Conf. Proc.*, 1981 Nat. Comput. Conf., pp. 337-341.

[23] S. S. Yau and J. S. Collofello, "Some stability measures for software maintenance," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 545-552, Nov. 1980.

[24] ——, "Design stability measures for software maintenance," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 849-856, Sept. 1985.

[25] G. M. Berns, "Assessing software maintainability," *Commun. ACM*, vol. 27, no. 1, pp. 14-23.

[26] H. Schafer, "Metrics for optimal maintenance management," in *Proc. Conf. Software Maintenance-1985*. Washington, DC: IEEE Comput. Soc. Press, 1985, pp. 114-119.

[27] R. S. Arnold and D. A. Parker, "The dimensions of healthy maintenance," in *Proc. 6th Int. Conf. Software Eng.* Washington, DC: IEEE Comput. Soc. Press, Sept. 1982, pp. 10-27.

[28] J. S. Collofello and J. W. Blaylock, "Syntactic information useful for software maintenance," in *AFIPS Conf. Proc.*, vol. 54, 1985 Nat. Comput. Conf., pp. 547-553.

[29] N. F. Schneidewind, "Usability of military standards for the maintenance of embedded computer software," in *Advisory Group for Aerospace Research & Development Conf. Proc. 330, Software for Avionics*, North Altantic Treaty Organization, The Hague, Netherlands, Sept. 6-10, 1982, pp. 21-1-21-6.

[30] J. A. McCall, M. A. Herdon, and W. M. Osborne, "Software Maintenance Management," Nat. Bureau Standards, NBS Special Publ. 500-129, Oct. 1985.

[31] R. J. Martin and W. M. Osborne, "Guidance of software maintenance," Nat. Bureau Standards, NBS Special Publ. 500-106, Dec. 1983.

[32] N. F. Schneidewind, "Evaluation of maintainability enhancement for the TCP/TSP Revision 6.0 Update .20," Naval Postgraduate School, Rep. NPS54-82-004, Feb. 1982.

[33] G. Arango, "TMM: Software maintenance by transformation," *IEEE Software*, vol. 3, no. 3, pp. 27-39, May 1986.

[34] R. N. Britcher and J. J. Craig, "Using modern design practices to upgrade aging software systems," *IEEE Software*, vol. 3, no. 3, pp. 16-24, May 1986.

[35] N. Chapin, "Software maintenance with fourth-generation languages," *ACM Software Engineering Notes*, vol. 9, no. 1, pp. 41-42, Jan. 1984.

[36] ——, "Software maintenance: A different view," in *AFIPS Conf. Proc. 54*, Nat. Comput. Conf., 1985, pp. 509-513.

[37] R. S. Arnold, N. F. Schneidewind, and N. Zvegintzov, "A software maintenance workshop," *Commun. ACM*, vol. 27, no. 11, pp. 1120-1121, 1158.

Norman F. Schneidewind (A'54-M'59-M'72-SM'77), for a photograph and biography, see this issue, p. 301.